BC - ABAP Programming

HELP.BCABA

Release 4.6C

Copyright

Icons

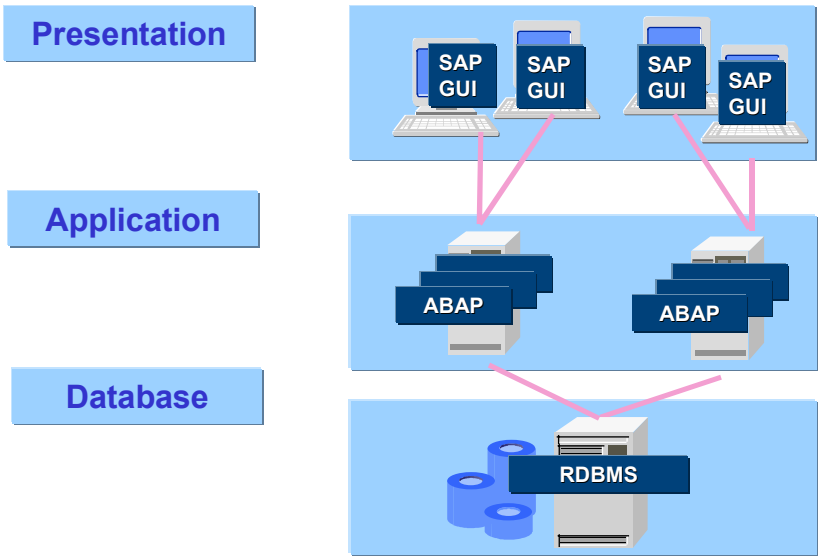
Icon	Meaning
	Caution
	Example
	Note
	Recommendation
	Syntax

Contents

BC - ABAP Programming

BC - ABAP Programming

This documentation describes how to write application programs within the three-tier client/server architecture of the R/3 System.



R/3 applications are written in the ABAP programming language, and run within the application layer of the R/3 System.

ABAP programs communicate with the database management system of the central relational database (RDBMS), and with the graphical user interface (SAPgui) at presentation level.

Contents

The documentation is divided into five sections:

Introduction to ABAP [Page 19]

This contains the basics of application programming in the R/3 System. This information is essential for an understanding of ABAP programming. Following an overview of the R/3 Basis system, it introduces the essential features of application programs and the ABAP programming language. Finally, it gives a short introduction to how you can create an application program in the ABAP Workbench.

The ABAP Programming Language [Page 82]

This section describes the statements in the ABAP programming language. Beginning with simple statements for data declarations, data processing, and program flow control, it progresses to topics such as modularization and special techniques, explaining which ABAP statements can be used for which purposes.

ABAP User Dialogs [Page 523]

BC - ABAP Programming

This section describes the different user dialogs that you can use in ABAP programs, and shows how you can program and control the interaction between program and user.

Running ABAP Programs [Page 937]

This section explains how ABAP programs are executed in the R/3 System. It shows how you can start programs, the conditions under which you must start them, and the different kinds of program execution.

ABAP Database Access [Page 1037]

This section explains how to work with the database in the R/3 System. It describes the parts of the programming language that are converted into SQL statements in the database, and shows how you can program database updates.

ABAP Objects [Page 1291]

This is an introduction to ABAP Objects, the object-oriented extension of ABAP. The section introduces objects, classes, and interfaces - the basic elements of ABAP Objects - and shows how you can define classes on their own, or using interfaces or inheritance. It then goes on to introduce further components of classes, namely methods and events.

Appendix [Page 1360]

The appendix contains summary descriptions and overviews, including a reference of ABAP statements and a glossary.

Further Reading

SAP Style Guide [Ext.]

Changing the SAP Standard [Ext.]

ABAP Workbench Tools [Ext.]

ABAP Dictionary [Ext.]

Remote Communications [Ext.]

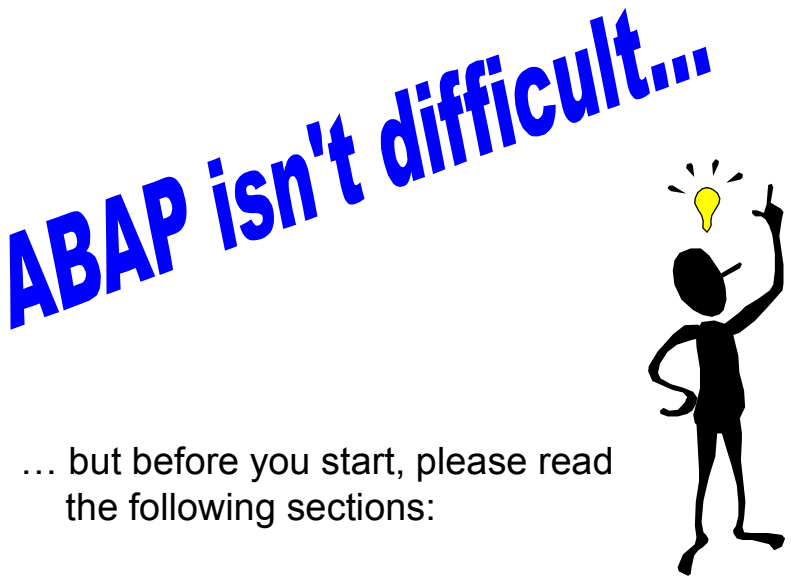
RFC Programming in ABAP [Ext.]

ABAP as an OLE Automation Controller [Ext.]

Basis Programming Interfaces [Ext.]

ABAP Query [Ext.]

Introduction to ABAP



The R/3 Basis System: Overview [Page 20]
Overview of the Components of Application Programs [Page 37]
Creating and Changing ABAP Programs [Page 68]

The R/3 Basis System: Overview

The R/3 Basis system is the platform for all other applications (financial accounting, logistics, human resources management) in the R/3 System.

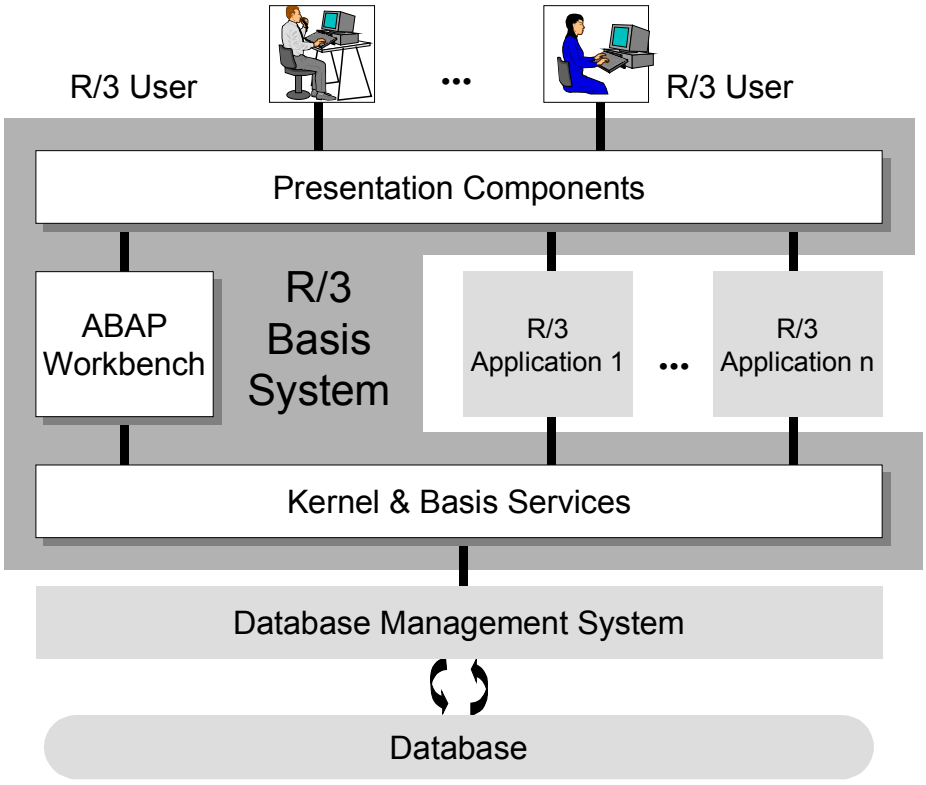
This documentation explains just what the Basis system is, and how it ties in with the R/3 System as a whole. It starts by introducing the Basis system in general. The second part concentrates on one central component - the application server. Finally, it will explain about work processes, which are components of the application server.

Position of the Basis System Within the R/3 System

The following sections describe three different views of the R/3 System, which show the role of the Basis system.

Logical View

The following illustration represents a logical view of the R/3 System.



The difference between the logical view and a hardware- or software-based view is that not all of the above components can be assigned to a particular hardware or software unit. The above diagram shows how the R/3 Basis system forms a central platform within the R/3 System. Below are listed the tasks of the three logical components of the R/3 Basis system.

Kernel and Basis Services

The kernel and basis services component is a runtime environment for all R/3 applications that is hardware-, operating system- and database-specific. The runtime environment is written principally in C and C++. However, some parts are also written in ABAP. The tasks of the kernel and basis services component are as follows:

- Running applications
  All R/3 applications run on software processors (virtual machines) within this component.
- User and process administration
  An R/3 System is a multi-user environment, and each user can run several independent applications. In short, this component is responsible for the tasks that usually belong to an

operating system. Users log onto the R/3 System and run applications within it. In this way, they do not come into contact with the actual operating system of the host. The R/3 System is the only user of the host operating system.

- Database access
  Each R/3 System is linked to a database system, consisting of a database management system (DBMS) and the database itself. The applications do not communicate directly with the database. Instead, they use Basis services.
- Communication
  R/3 applications can communicate with other R/3 Systems and with non-SAP systems. It is also possible to access R/3 applications from external systems using a BAPI interface. The services required for communication are all part of the kernel and basis services component.
- System Monitoring and Administration
  The component contains programs that allow you to monitor and control the R/3 System while it is running, and to change its runtime parameters.

ABAP Workbench

The ABAP Workbench component is a fully-fledged **development environment** for applications in the ABAP language. With it, you can create, edit, test, and organize application developments. It is fully integrated in the R/3 Basis system and, like other R/3 applications, is itself written in ABAP.

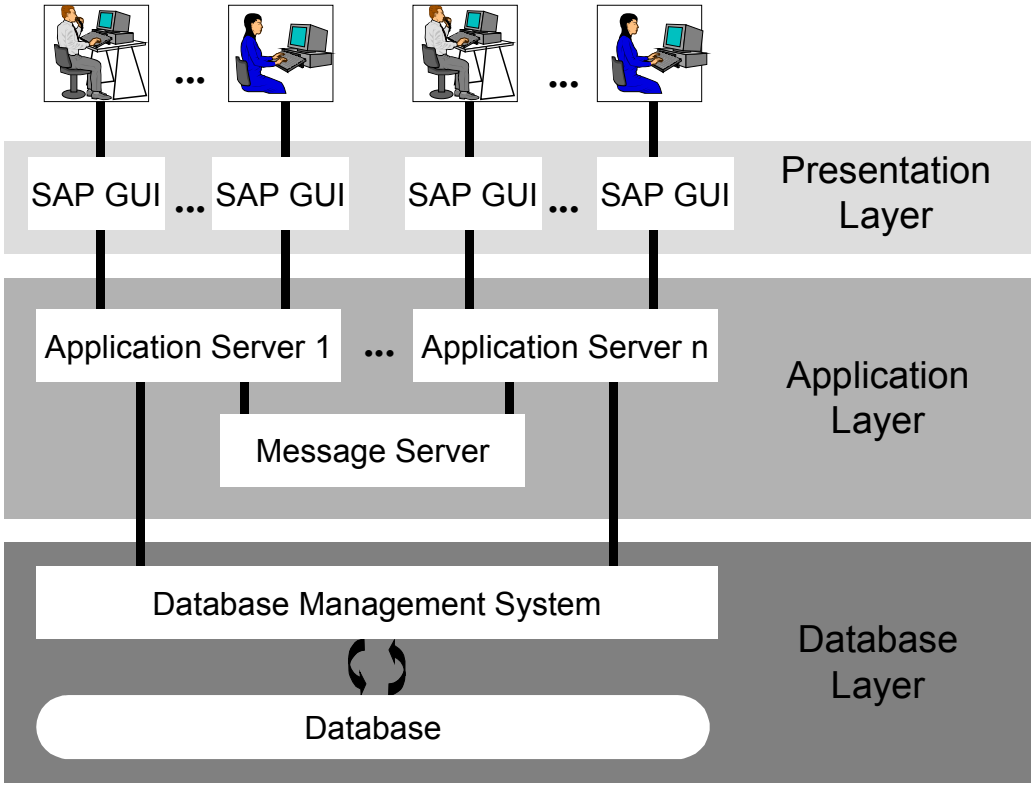
Presentation Components

The presentation components are responsible for the **interaction** between the R/3 System and the user, and for **desktop component integration** (such as word processing and spreadsheets).

Software-oriented View

The following illustration represents a software-oriented view of the R/3 System. The software-oriented view describes the various software components that make up the R/3 System. In the software-oriented view, all of the SAPgui components and application servers in the R/3 System make up the R/3 Basis system.

Position of the Basis System Within the R/3 System



The R/3 Basis system is a multi-tier client/server system. The individual software components are arranged in tiers and function, depending on their position, as a client for the components below them or a server for the components above them. The classic configuration of an R/3 System contains the following software layers:

Database Layer

The database layer consists of a central database system containing **all** of the data in the R/3 System. The database system has two components - the database management system (DBMS), and the databse itself. SAP does not manufacture its own database. Instead, the R/3 System supports the following database systems from other suppliers: ADABAS D, DB2/400 (on AS/400), DB2/Common Server, DB2/MVS,INFORMIX, Microsoft SQL Server, ORACLE, and ORACLE Parallel Server.

The database does not only contain the master data and transaction data from your business applications, **all** data for the **entire** R/3 System is stored there. For example, the database contains the control and Customizing data that determine how your R/3 System runs. It also contains the program code for your applications. Applications consist of program code, screen definitions, menus, function modules, and various other components. These are stored in a special section of the database called the R/3 Repository, and are accordingly called Repository objects. You work with them in the ABAP Workbench.

Application Layer

The application layer consists of one or more application servers and a message server. Each application server contains a set of services used to run the R/3 System. Theoretically, you only need one application server to run an R/3 System. In practice, the services are distributed across more than one application server. This means that not all application servers will provide the full

range of services. The message server is responsible for communication between the application servers. It passes requests from one application server to another within the system. It also contains information about application server groups and the current load balancing within them. It uses this information to choose an appropriate server when a user logs onto the system.

Presentation Layer

The presentation layer contains the software components that make up the SAPgui (graphical user interface). This layer is the interface between the R/3 System and its users. The R/3 System uses the SAPgui to provide an intuitive graphical user interface for entering and displaying data. The presentation layer sends the user's input to the application server, and receives data for display from it. While a SAPgui component is running, it remains linked to a user's terminal session in the R/3 System.

This software-oriented view can be expanded to include further layers, such as an Intenet Transaction Server (ITS).

Software-oriented and Hardware-oritented View

The software-oriented view has nothing to do with the hardware configuration of the system. There are many different hardware configuration possibilities for both layers and components. When distributing the layers, for example, you can have all layers on a single host, or, at the other extreme, you could have at least one host for each layer. When dealing with components, the distribution of the database components depends on the database sytsem you are using. The application layer and presentation layer components can be distributed across any number of hosts. It is also possible to install more than one application server on a single host. A common configuration is to run the database system and a single application server (containing special database services) on one host, and to run each further application server on its own host. The presentation layer components usually run on the desktop computers of the users.

Advantages of the Multi-tier Architecture

The distribution of the R/3 software over three layers means that the system load is also distributed. This leads to better system performance.

Since the database system contains all of the data for the entire R/3 System, it is subject to a very heavy load when the sytsem is running. It is therefore a good idea not to run application programs on the same host. The architecture of the R/3 System, in which the application layer and database layer are separate, allows you to install them on separate hosts and let them communicate using the network.

It also makes sense to separate program execution from the tasks of processing user input and formatting data output. This is made possible by separating the presentation layer and the application layer. SAPgui and the application servers are designed so that the minimum amount of data has to be transported between the two layers. This means that the presentation layer components can even be used on hosts that have slow connections to application servers a long way away.

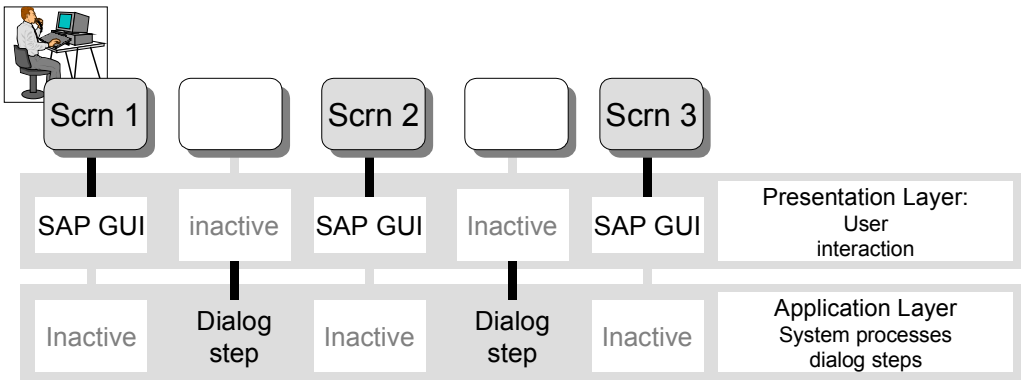
The system is highly scalable, due to the fact that the software components of an R/3 System can be distributed in almost any configuration across various hosts. This is particularly valuable in the application layer, where you can easily adapt your R/3 System to meet increasing demand by installing further application servers.

Consequences for Application Programming

The fact that the application and presentation layers are separate carries an important consequence for application programmers. When you run an application program that requires

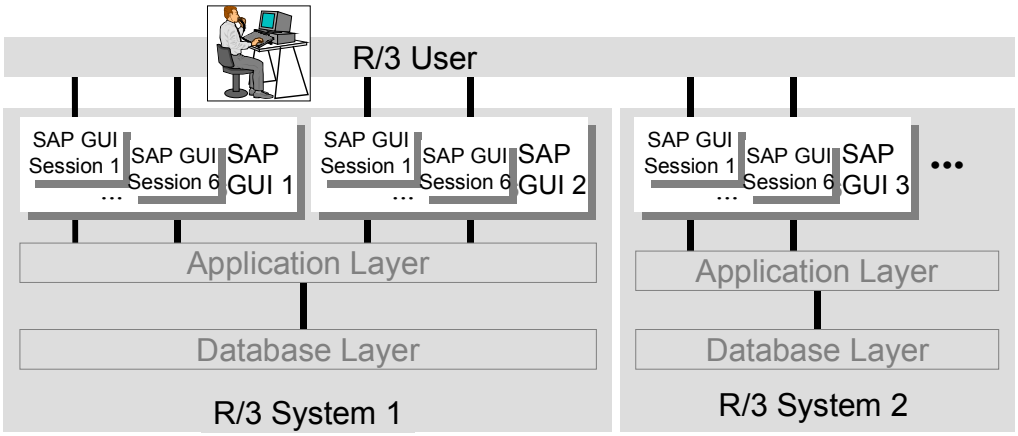
user interaction, control of the program is continually passed backwards and forwards between the layers. When a screen is ready for user input, the presentation layer is active, and the application server is inactive with regard to that particular program, but free for other tasks. Once the user has entered data on the screen, program control passes back to the application layer. Now, the presentation layer is inactive. The SAPgui is still visible to the user during this time, and it is still displaying the screen, but it **cannot accept user input** The SAPgui does not become active again until the application program has called a new screen and sent it to the presentation server.

As a consequence, the program logic in an application program that occurs between two screens is known as a **dialog step**.



User-oriented View

The following illustration represents a user-oriented view of the R/3 System:



For the user, the visible components of the R/3 System are those that appear as a window on the screen. The windows are generated by the presentation layer of the R/3 System, and form a part of the R/3 Basis system.

Before the user logs onto the R/3 System, he or she must start a utility called SAP Logon, which is installed at the front end. In SAP Logon, the user chooses one of the available R/3 Systems. The program then connects to the message server of that system and obtains the address of a suitable (most lightly-used) application server. It then starts a SAPgui, connected to that application server. The SAP Logon program is then no longer required for this connection.

SAPgui starts the logon screen. Once the user has successfully logged on, it displays the initial screen of the R/3 System in an R/3 window on the screen. Within SAPgui, the R/3 window is represented as a session. After logging on, the user can open up to five further sessions (R/3 windows) **within** the single SAPgui. These behave almost like independent SAPguis. The different sessions allow you to run different applications in parallel, independently of one another.

Within a session, the user can run applications that themselves call further windows (such as dialog boxes and graphic windows). These windows are not independent - they belong to the session from which they were called. These windows can be either modal (the original window is not ready for input) or amodal (both windows are ready for input).

The user can open other SAPguis, using SAP Logon, to log onto the same system or another R/3 System. The individual SAPguis and corresponding R/3 terminal sessions are totally independent. This means that you can have SAPguis representing the presentation layers of several R/3 Systems open on your desktop computer.

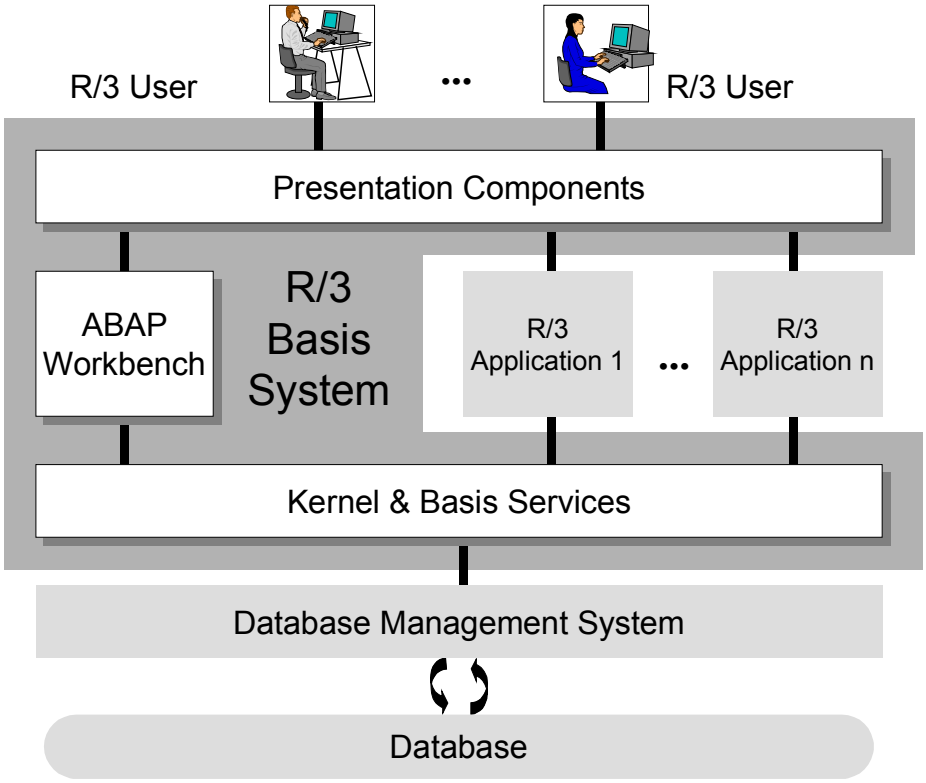
Application Servers

R/3 programs run on application servers. They are an important component of the R/3 System. The following sections describe application servers in more detail.

Structure of an Application Server

The application layer of an R/3 System is made up of the application servers and the message server. Application programs in an R/3 System are run on application servers. The application servers communicate with the presentation components, the database, and also with each other, using the message server.

The following diagram shows the structure of an application server:



The individual components are:

Work Processes

An application server contains work processes, which are components that can run an application. Each work process is linked to a memory area containing the context of the application being run. The context contains the current data for the application program. This needs to be available in each dialog step. Further information about the different types of work process is contained later on in this documentation.

Application Servers

Dispatcher

Each application server contains a dispatcher. The dispatcher is the link between the work processes and the users logged onto the application server. Its task is to receive requests for dialog steps from the SAPgui and direct them to a free work process. In the same way, it directs screen output resulting from the dialog step back to the appropriate user.

Gateway

Each application server contains a gateway. This is the interface for the R/3 communication protocols (RFC, CPI/C). It can communicate with other application servers in the same R/3 System, with other R/3 Systems, with R/2 Systems, or with non-SAP systems.

The application server structure as described here aids the performance and scalability of the entire R/3 System. The fixed number of work processes and dispatching of dialog steps leads to optimal memory use, since it means that certain components and the memory areas of a work process are application-independent and reusable. The fact that the individual work processes work independently makes them suitable for a multi-procecssor architecuture. The methods used in the dispatcher to distribute tasks to work processes are discussed more closely in the section *Dispatching Dialog Steps*.

Shared Memory

All of the work processes on an application server use a common main memory area called shared memory to save contexts or to buffer constant data locally.

The resources that all work processes use (such as programs and table contents) are contained in shared memory. Memory management in the R/3 System ensres that the work processes always address the correct context, that is the data relevant to the current state of the program that is running. A mapping process projects the required context for a dialog step from shared memory into the address of the relevant work process. This reduces the actual copying to a minimum.

Local buffering of data in the shared memory of the application server reduces the number of database reads required. This reduces access times for application programs considerably. For optimal use of the buffer, you can concentrate individual applications (financial accounting, logistics, human resources) into separate application server groups.

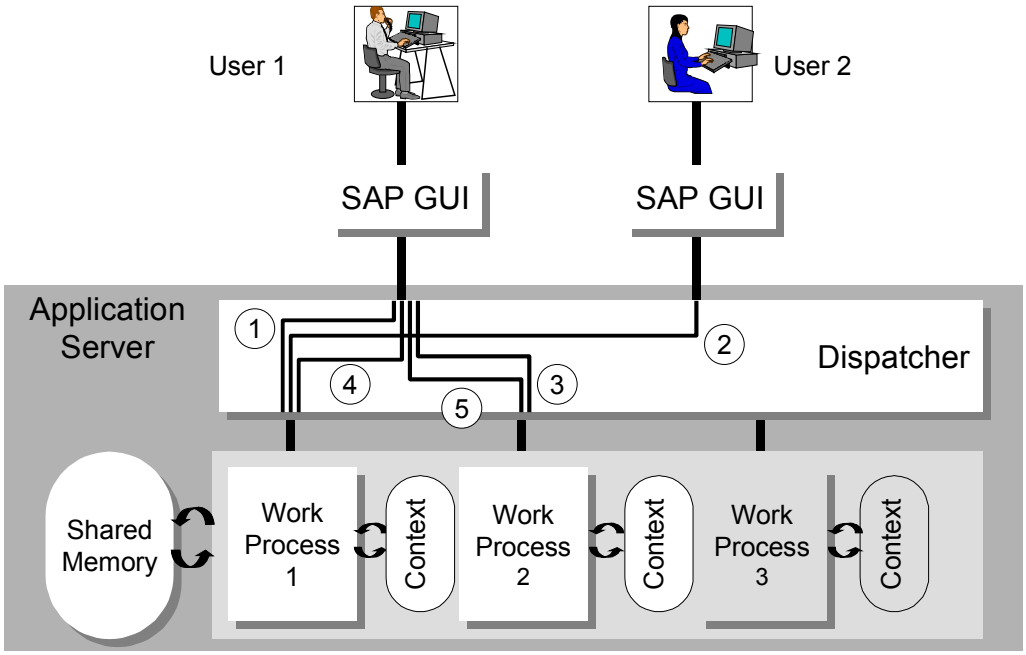
Database Connection

When you start up an R/3 System, each application server registers its work proceses with the database layer, and receives a single dedicated channel for each. While the system is running, each work process is a user (client) of the database system (server). You cannot change the work process registration while the system is running. Neither can you reassign a database channel from one work process to another. For this reason, a work process can only make database changes within a **single** database logical unit of work (LUW). A database LUW is an inseparable sequence of database operations. This has important consequences for the programming model explained below.

Dispatching Dialog Steps

The number of users logged onto an application server is often many times greater than the number of available work processes. Furthermore, it is not restricted by the R/3 system architecture. Furthermore, each user can run several applications at once. The dispatcher has the important task of distributing all dialog steps among the work processes on the application server.

The following diagram is an example of how this might happen:



1. The dispatcher receives the request to execute a dialog step from user 1 and directs it to work process 1, which happens to be free. The work process addresses the context of the application program (in shared memory) and executes the dialog step. It then becomes free again.
2. The dispatcher receives the request to execute a dialog step from user 2 and directs it to work process 1, which is now free again. The work process executes the dialog step as in step 1.
3. While work process 1 is still working, the dispatcher receives a further request from user 1 and directs it to work process 2, which is free.
4. After work processes 1 and 2 have finished processing their dialog steps, the dispatcher receives another request from user 1 and directs it to work process 1, which is free again.
5. While work process 1 is still working, the dispatcher receives a further request from user 2 and directs it to work process 2, which is free.

From this example, we can see that:

- A dialog step from a program is assigned to a single work process for execution.
- The individual dialog steps of a program can be executed on different work processes, and the program context must be addressed for each new work process.
- A work process can execute dialog steps of different programs from different users.

The example does not show that the dispatcher tries to distribute the requests to the work processes such that the same work process is used as often as possible for the successive dialog steps in an application. This is useful, since it saves the program context having to be addressed each time a dialog step is executed.

Dispatching and the Programming Model

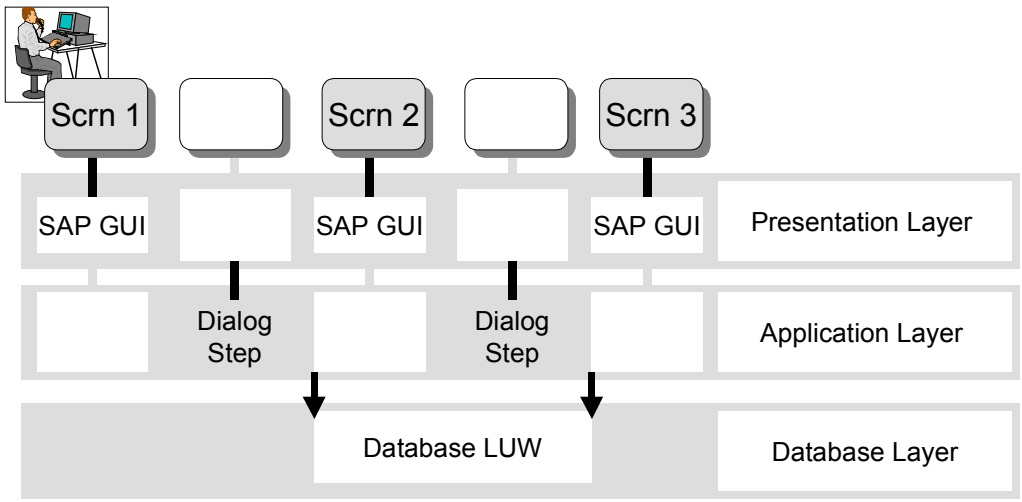
The separation of application and presentation layer made it necessary to split up application programs into dialog steps. This, and the fact that dialog steps are dispatched to individual work processes, has had important consequences for the programming model.

As mentioned above, a work process can only make database changes within a **single** database logical unit of work (LUW). A database LUW is an inseparable sequence of database operations. The contents of the database must be consistent at its beginning and end. The beginning and end of a database LUW are defined by a commit command to the database system (database commit). During a database LUW, that is, between two database commits, the database system itself ensures consistency within the database. In other words, it takes over tasks such as locking database entries while they are being edited, or restoring the old data (rollback) if a step terminates in an error.

A typical SAP application program extends over several screens and the corresponding dialog steps. The user requests database changes on the individual screens that should lead to the database being consistent once the screens have all been processed. However, the individual dialog steps run on different work processes, and a single work process can process dialog steps from other applications. It is clear that two or more independent applications whose dialog steps happen to be processed on the same work process cannot be allowed to work with the same database LUW.

Consequently, a work process must open a **separate** datables LUW for **each** dialog step. The work process sends a commit command (database commit) to the database at the end of each dialog step in which it makes database changes. These commit commands are called implicit database commits, since they are not explicitly written into the application program.

These implicit database commits mean that a database LUW can be kept open for a maximum of one dialog step. This leads to a considerable reduction in database load, serialization, and deadlocks, and enables a large number of users to use the same system.



However, the question now arises of how this method (1 dialog step = 1 database LUW) can be reconciled with the demand to make commits and rollbacks dependent on the logical flow of the application program instead of the technical distribution of dialog steps. Database update requests that depend on one another form logical units in the program that extend over more

than one dialog step. The database changes associated with these logical units must be executed together and must also be able to be undone together.

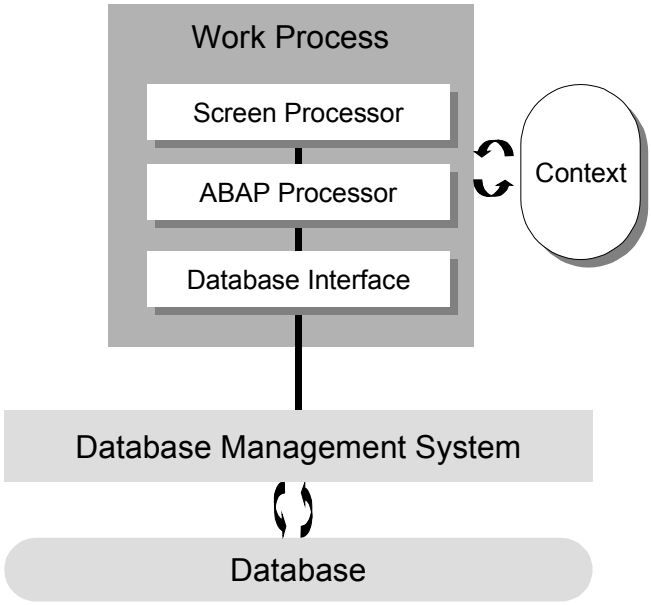
The SAP programming model contains a seies of bundling techniques that allow you to group database updates together in logical units. The section of an R/3 application program that bundles a set of logically-associated database operations is called an **SAP LUW**. Unlike a database LUW, a SAP LUW includes all of the dialog steps in a logical unit, including the database update.

Work Processes

Work processes execute the individual dialog steps in R/3 applications. The next two sections describe firstly the structure of a work process, and secondly the different types of work process in the R/3 System.

Structure of a Work Process

Work processes execute the dialog steps of application programs. They are components of an application server. The following diagram shows the components of a work process:



Each work process contains two software processors and a database interface.

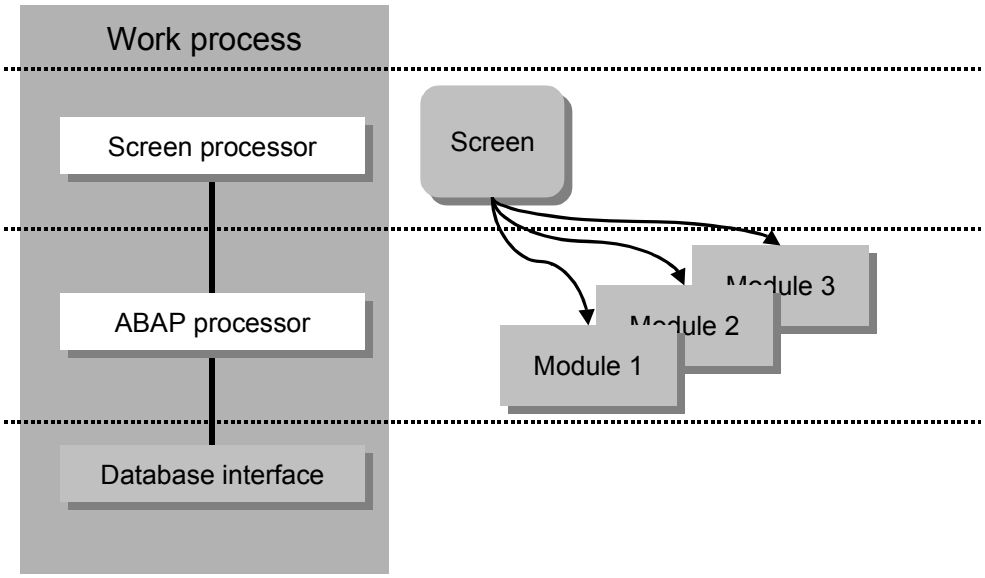
Screen Processor

In R/3 application programming, there is a difference between **user interaction** and **processing logic**. From a programming point of view, user interaction is controlled by **screens**. As well as the actual input mask, a screen also consists of flow logic. The screen flow logic controls a large part of the user interaction. The R/3 Basis system contains a special language for programming screen flow logic. The screen processor executes the screen flow logic. Via the dispatcher, it takes over the responsibility for communication between the work process and the SAPgui, calls modules in the flow logic, and ensures that the field contents are transferred from the screen to the flow logic.

ABAP-Prozessor

The actual processing logic of an application program is written in **ABAP** - SAP's own programing language. The ABAP processor executes the **processing logic** of the application program, and communicates with the database interface. The screen processor tells the ABAP processor which module of the **screen flow logic** should be processed next. The following

screen illustrates the interaction between the screen and the ABAP processors when an application program is running.



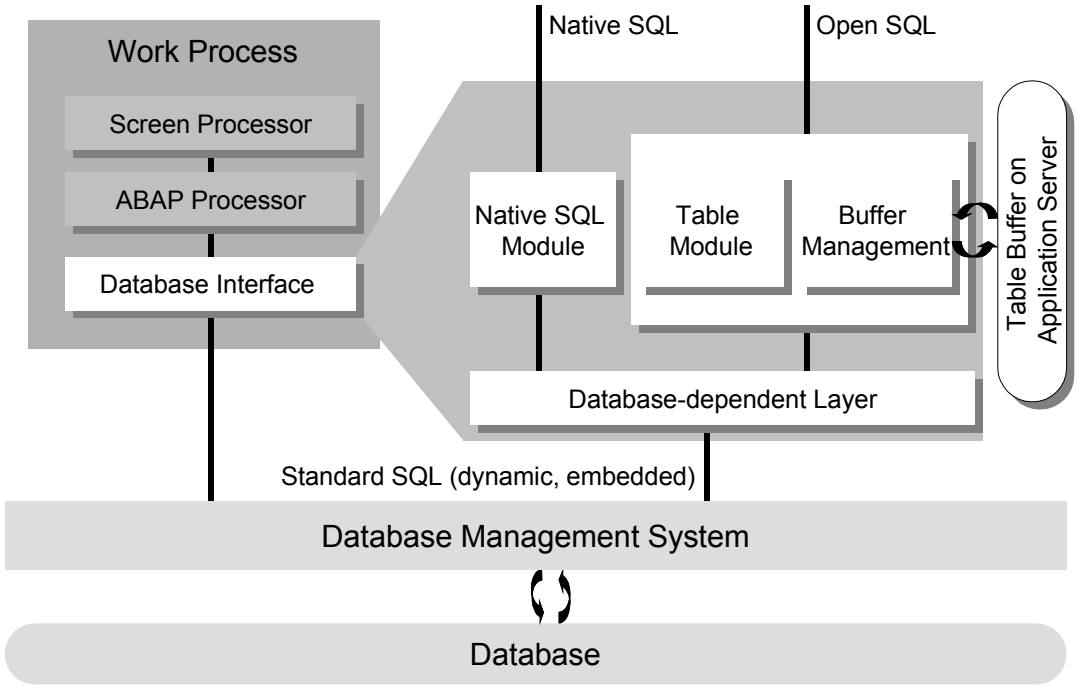
Database Interface

The database interface provides the following services:

- Establishing and terminating connections between the work process and the database.
- Access to database tables
- Access to R/3 Repository objects (ABAP programs, screens and so on)
- Access to catalog information (ABAP Dictionary)
- Controlling transactions (commit and rollback handling)
- Table buffer administration on the application server.

The following diagram shows the individual components of the database interface:

Work Processes



The diagram shows that there are two different ways of accessing databases: Open SQL and Native SQL.

Open SQL statements are a subset of Standard SQL that is fully integrated in ABAP. They allow you to access data irrespective of the database system that the R/3 installation is using. Open SQL consists of the Data Manipulation Language (DML) part of Standard SQL; in other words, it allows you to read (SELECT) and change (INSERT, UPDATE, DELETE) data. The tasks of the Data Definition Language (DDL) and Data Control Language (DCL) parts of Standard SQL are performed in the R/3 System by the ABAP Dictionary and the authorization system. These provide a unified range of functions, irrespective of database, and also contain functions beyond those offered by the various database systems.

Open SQL also goes beyond Standard SQL to provide statements that, in conjunction with other ABAP constructions, can simplify or speed up database access. It also allows you to buffer certain tables on the application server, saving excessive database access. In this case, the database interface is responsible for comparing the buffer with the database. Buffers are partly stored in the working memory of the current work process, and partly in the shared memory for all work processes on an application server. Where an R/3 System is distributed across more than one application server, the data in the various buffers is synchronized at set intervals by the buffer management. When buffering the database, you must remember that data in the buffer is not always up to date. For this reason, you should only use the buffer for data which does not often change.

Native SQL is only loosely integrated into ABAP, and allows access to all of the functions contained in the programming interface of the respective database system. Unlike Open SQL statements, Native SQL statements are not checked and converted, but instead are sent directly to the database system. Programs that use Native SQL are specific to the database system for which they were written. R/3 applications contain as little Native SQL as possible. In fact, it is only used in a few Basis components (for example, to create or change table definitions in the ABAP Dictionary).

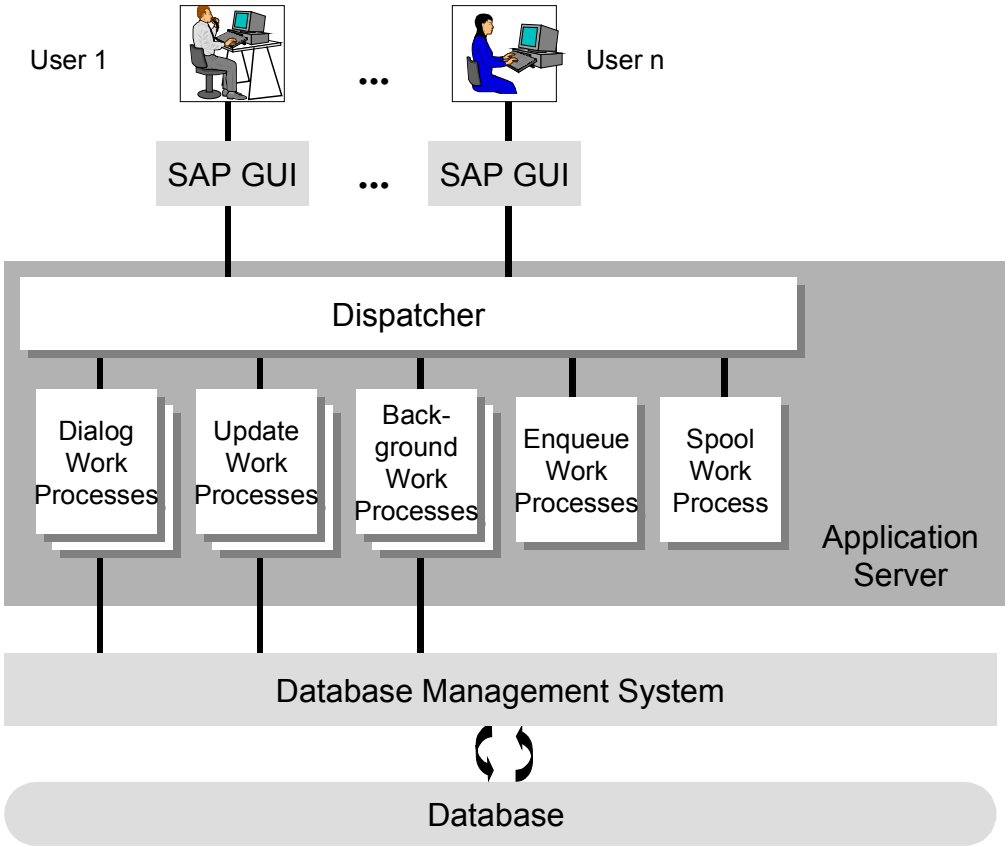
The database-dependent layer in the diagram serves to hide the differences between database systems from the rest of the database interface. You choose the appropriate layer when you install the Basis system. Thanks to the standardization of SQL, the differences in the syntax of statements are very slight. However, the semantics and behavior of the statements have not been fully standardized, and the differences in these areas can be greater. When you use Native SQL, the function of the database-dependent layer is minimal.

Types of Work Process

Although all work processes contain the components described above, they can still be divided into different types. The type of a work process determines the kind of task for which it is responsible in the application server. It does not specify a particular set of technical attributes. The individual tasks are distributed to the work processes by the dispatcher.

Before you start your R/3 System, you determine how many work processes it will have, and what their types will be. The dispatcher starts the work processes and only assigns them tasks that correspond to their type. This means that you can distribute work process types to optimize the use of the resources on your application servers.

The following diagram shows again the structure of an application server, but this time, includes the various possible work process types:



Work Processes

The various work processes are described briefly below. Other parts of this documentation describe the individual components of the application server and the R/3 System in more detail.

Dialog Work Process

Dialog work processes deal with requests from an active user to execute dialog steps.

Update Work Process

Update work processes execute database update requests. Update requests are part of an SAP LUW that bundle the database operations resulting from the dialog in a database LUW for processing in the background.

Background Work Process

Background work processes process programs that can be executed without user interaction (background jobs).

Enqueue Work Process

The enqueue work process administers a lock table in the shared memory area. The lock table contains the logical database locks for the R/3 System and is an important part of the SAP LUW concept. In an R/3 System, you may only have one lock table. You may therefore also only have one application server with enqueue work processes.

Spool Work Process

The spool work process passes sequential datasets to a printer or to optical archiving. Each application server may contain only one spool work process.

The services offered by an application server are determined by the types of its work processes. One application server may, of course, have more than one function. For example, it may be both a dialog server and the enqueue server, if it has several dialog work processes and an enqueue work process.

You can use the system administration functions to switch a work process between dialog and background modes while the system is still running. This allows you, for example, to switch an R/3 System between day and night operation, where you have more dialog than background work processes during the day, and the other way around during the night.

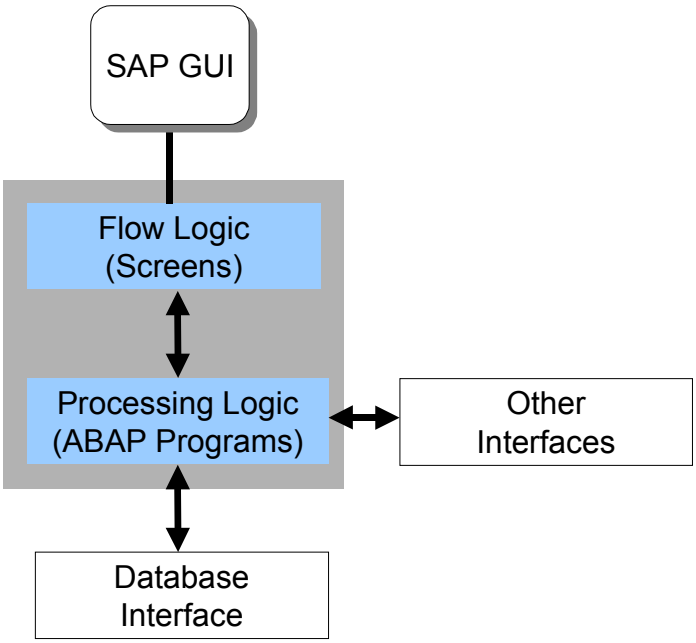
Overview of the Components of Application Programs

This overview describes application programming in the R/3 System. All application programs, along with parts of the R/3 Basis system, are written in the ABAP Workbench using ABAP, SAP's programming language. The individual components of application programs are stored in a special section of the database called the R/3 Repository. The R/3 Repository serves as a central store for all of the development objects in the R/3 System. The following sections of this documentation cover the basics and characteristics of application programming.

Structure of an Application Program

R/3 application programs run within the R/3 Basis system on the work processes of application servers. This makes them independent of the hardware and operating system that you are using. However, it also means that you cannot run them outside the R/3 System.

As described in the Overview of the R/3 Basis System [Page 20], a work process contains a screen processor for processing user input, an ABAP processor for processing the program logic, and a database interface for communicating with the database. These components of a work process determine the following structure of an applciation program:



An application program consists of two components, each of which has a different task:

Flow Logic

Interaction between application programs and the user is implemented using screens. Screens are processed by the screen processor of a work process. As well as the input mask, they consist of flow logic. This is coding, written using a special set of keywords called the screen language. The input mask is displayed by the SAPgui, which also transfers the user action on the screen back to the flow logic. In the program flow, screens react to the user actions and call program modules. These program modules form the processing logic.

Processing logic

The components of application programs that are responsible for data processing in the R/3 System are ABAP programs. ABAP stands for ‘Advanced Business Application Programming’. ABAP programs run on the ABAP processor of a work process. They receive screen input from the screen processor and send it to the screen processor. You access the database using the database interface. ABAP contains a special set of commands called OPEN SQL. This allows you to read from and write to the database regardless of the database you are using. The

database interface converts the OPEN SQL commands into commands of the relevant database. You can also use native SQL commands, which are passed to the database without first being converted. There is a range of further interfaces such as memory, sequential files and external interfaces. These provide other means of sending data to and from ABAP programs. When working together with screens, ABAP programs play a more passive role, acting as a container for a set of modules that can be called from the flow logic.

Screens

Each screen that a user sees in the R/3 System belongs to an application program. Screens send data to, receive data from, and react to the user's interaction with the input mask. There are three ways to organize screen input and output. These differ in the way in which they are programmed, and in how the user typically interacts with them.

Screens

In the most general case, you create an entire screen and its flow logic by hand using the **Screen Painter** in the ABAP Workbench.

Selection Screens and Lists

There are two special kinds of screen in the R/3 System that you will often use - selection screens and lists. These screens and their flow logic are generated from ABAP statements in the processing logic. In this case, you do not have to work with the **Screen Painter**. Instead, you define the entire screen in the **processing logic**.

Screens in the R/3 System also contain a menu bar, a standard toolbar, and an application toolbar. Together, these three objects form the **status** of the screen. Each status is an object of its corresponding application program. It is a standalone part of the screen . Since it does not expressly belong to one screen, it can be reused in any number of screens. You define statuses using the **Menu Painter** in the ABAP Workbench. You can define your own statuses for screens and lists. Selection screens have a fixed status.

The following sections describe the different types of screen in more detail.

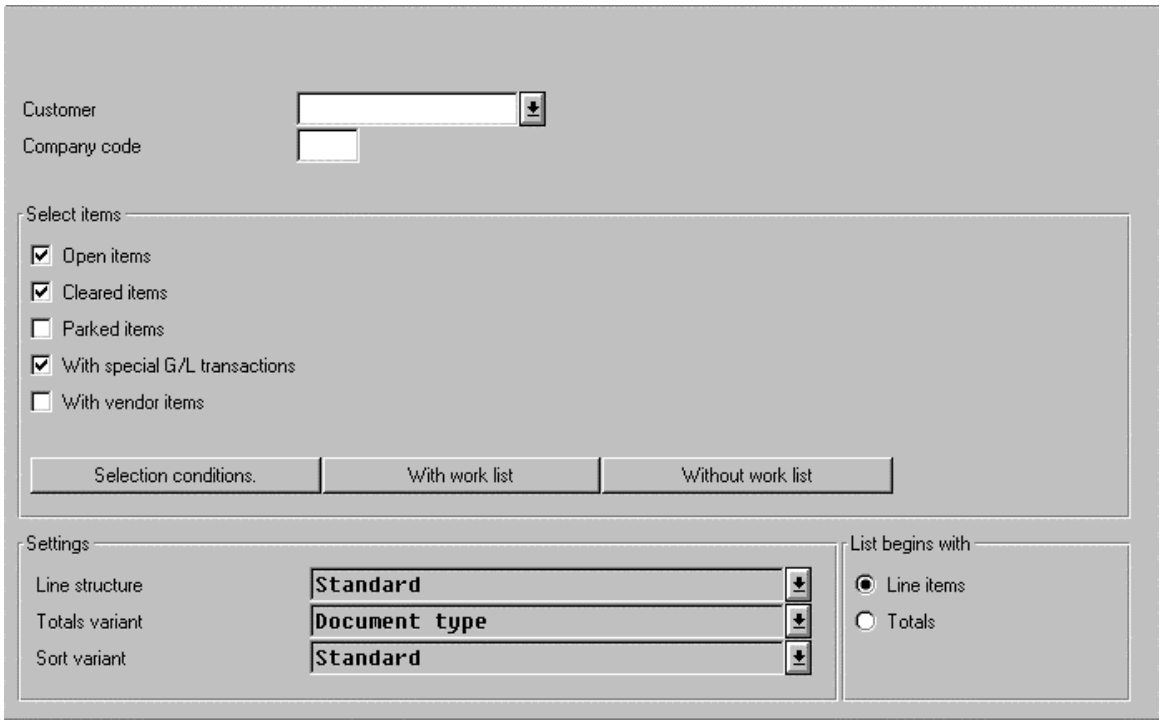
Screens

Each screen contains an input mask that you can use for data input and output. You can design the mask yourself. When the screen mask is displayed by the SAPgui, two events are triggered: Before the screen is displayed, the Process Before Output (PBO) event is processed. When the user interacts with the screen, the Process After Input (PAI) event is processed.

Each screen is linked to a single PBO processing block and a single PAI processing block. The PAI of a screen and the PBO of the subsequent screen together form a **dialog step** in the application program.

The screen language is a special subset of ABAP, and contains only a few keywords. The statements are syntactically similar to the other ABAP statements, but you may not use screen statements in ABAP programs or ABAP statements in the screen flow logic. The most important screen keywords are MODULE, FIELD, CHAIN, and LOOP. Their only funciton is to link the processing logic to the flow logic, that is, to call modules in the processing logic, and control data transfer between the screen and the ABAP program, for example, by checking fields.

The input/output mask of a screen contains all of the normal graphical user interface elements, such as input/output fields, pushbuttons, and radio buttons. The following diagram shows a typical screen mask:



All of the active elements on a screen have a field name and are linked to screen fields in shared memory. You can link screen fields to the ABAP Dictionary. This provides you with automatic field and value help, as well as consistency checks.

When a user action changes the element, the value of the screen field is changed accordingly. Values are transported between screens and the processing logic in the PAI event, where the contents of the screen fields are transported to program fields with the same name.

Each screen calls modules in its associated ABAP program which prepare the screen (PBO) and process the entries made by the user (PAI). Although there are ABAP statements for changing the attributes of screen elements (such as making them visible or invisible), but there are none for defining them.

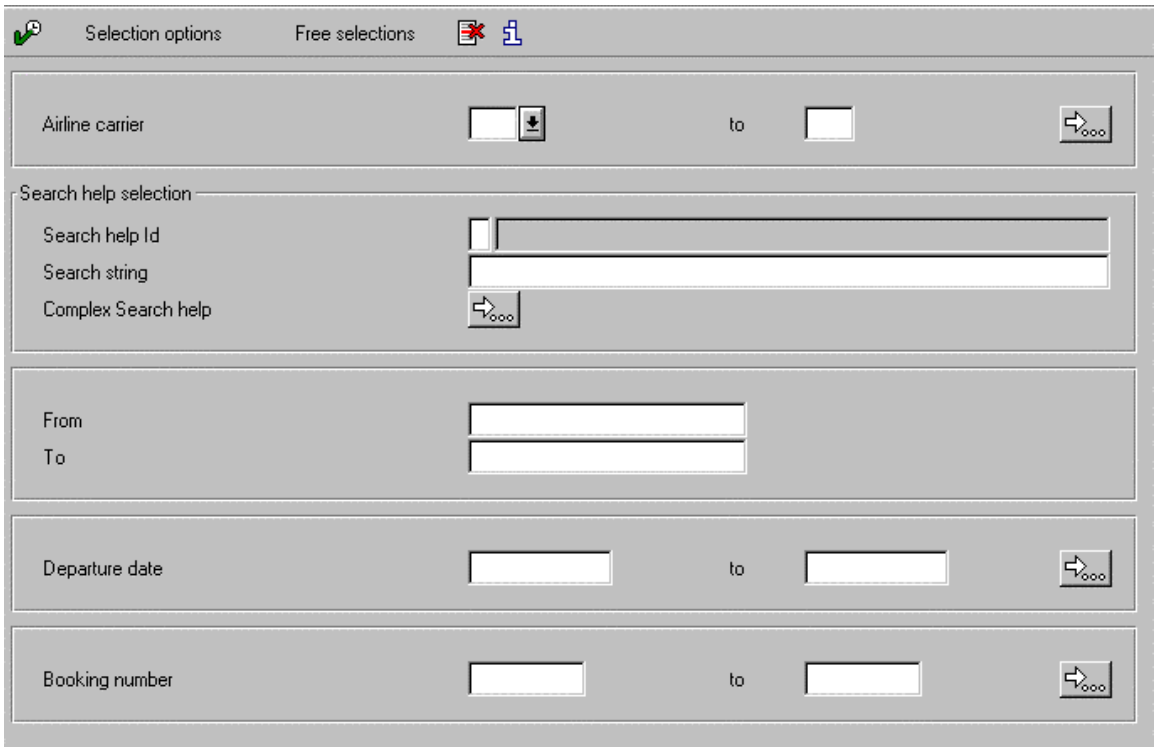
Dialog screens enable the user and application programs to communicate with each other. They are used in dialog-oriented programs such as transactions, where the program consists mainly of processing a sequence of screens. Essentially, you use screens when you need more flexibility than is offered by selection screens or lists.

Selection Screens

Selection screens are special screens used to enter values in ABAP programs. Instead of using the Screen Painter, you create them using ABAP statements in the processing logic of your program. The selection screen is then generated according to these statements. The screen flow logic is supplied by the system, and remains invisible to you as the application programmer.

You define selection screens in the declaration part of an ABAP program using the special declaration statements PARAMETERS, SELECT-OPTIONS and SELECTION-SCREEN). These statements declare and format the input fields of each selection screen. The following is a typical selection screen:

Screens



The most important elements on a selection screen are input fields for single values and for selection tables. Selection tables allow you to enter more complicated selection criteria. Selection tables are easy to use in ABAP because the system automatically processes them itself. As on other screens, field and possible values help is provided for input fields which refer to an ABAP Dictionary field. Users can use pre-configured sets of input values for selection screens. These are called variants.

You call a selection screen from an ABAP program using the CALL SELECTION-SCREEN statement. If the program is an executable (report) with type 1, the ABAP runtime environment automatically calls the selection screen defined in the declaration part of the program. Selection screens trigger events, and can therefore call event blocks in ABAP programs.

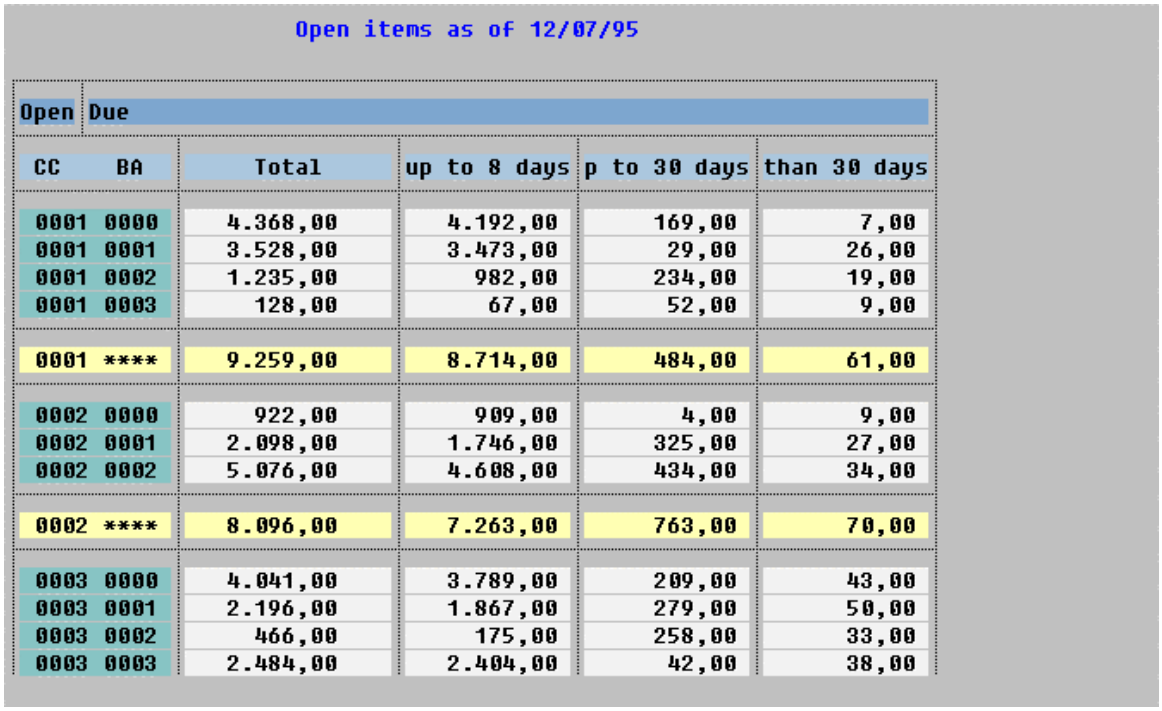
Since selection screens contain principally input fields, selection screen dialogs are more input-oriented than the screens you define using the Screen Painter. Dialog screens can contain both input and output fields. Selection screens, however, are appropriate when the program requires data from the user before it can continue processing. For example, you would use a selection screen before accessing the database, to restrict the amount of data read.

Lists

Lists are output-oriented screens which display formatted, structured data. They are defined, formatted, and filled using ABAP commands. The system displays lists defined in ABAP on a special list screen. As with selection screens, the flow logic is supplied by the system and remains hidden from the application programmer.

The most important task of a list is to display data. However, users can also interact with them. Lists can react to mouse clicks and contain input fields. Despite these similarities with other types of screen, lists are displayed using a completely different technique. Input fields on lists cannot be compared with those on normal screens, since the method of transferring data between the list and the ABAP program is completely different in each case. If input fields on lists are linked to

the ABAP Dictionary, the usual automatic field and possible values help is available. The following is a typical list:



Open items as of 12/07/95

Open	Due			
CC	BA	Total	up to 8 days	p to 30 days	than 30 days
0001	0000	4.368,00	4.192,00	169,00	7,00
0001	0001	3.528,00	3.473,00	29,00	26,00
0001	0002	1.235,00	982,00	234,00	19,00
0001	0003	128,00	67,00	52,00	9,00
0001	****	9.259,00	8.714,00	484,00	61,00
0002	0000	922,00	909,00	4,00	9,00
0002	0001	2.098,00	1.746,00	325,00	27,00
0002	0002	5.076,00	4.608,00	434,00	34,00
0002	****	8.096,00	7.263,00	763,00	70,00
0003	0000	4.041,00	3.789,00	209,00	43,00
0003	0001	2.196,00	1.867,00	279,00	50,00
0003	0002	466,00	175,00	258,00	33,00
0003	0003	2.484,00	2.404,00	42,00	38,00

You define lists using a special set of statements (the list statements WRITE, SKIP, ULINE, NEW-PAGE and so on) in the processing blocks of ABAP programs. When these statements are executed, a list is composed within the system. An internal system program called the list processor is responsible for displaying lists and for interpreting user actions in the list. Lists are important because only data in list format can be sent to the R/3 spool system for printing.

In an ABAP program, you can use the LEAVE TO LIST-PROCESSING statement to define the next screen as a list. If the ABAP program is an executable (report) with type 1, the ABAP runtime environment automatically calls the list defined in your program. A single program can be responsible for a stack of up to 21 lists. From one basic list, you can create up to 20 details lists. User actions on a list screen trigger events, and can thus call event blocks in the ABAP program.

Lists are output-oriented. When users carry out actions on a list screen, it is normally to use part of the list contents in the next part of the program, and not to input values directly. Using lists is appropriate when you want to work with output data, to print data or when the user’s next action depends on output data.

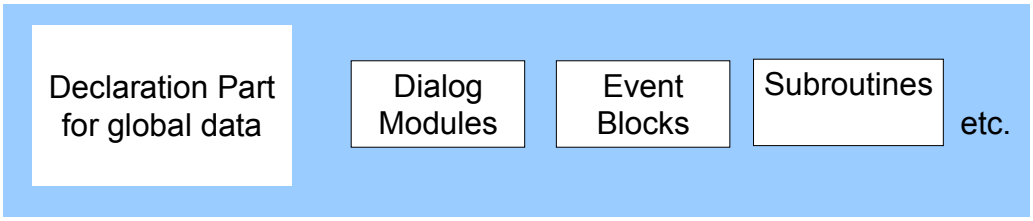
Structure of ABAP Programs

ABAP processing logic is responsible for processing data in R/3 application programs. ABAP was designed specifically for dialog-oriented database applications. The following sections deal with how an ABAP program is structured and executed.

Program Structure

ABAP programs are responsible for data processing within the individual **dialog steps** of an application program. This means that the program cannot be constructed as a single sequential unit, but must be divided into sections that can be assigned to the individual dialog steps. To meet this requirement, ABAP programs have a modular structure. Each module is called a **processing block**. A processing block consists of a set of ABAP statements. When you run a program, you effectively call a series of processing blocks. They cannot be nested.

The following diagram shows the structure of an ABAP program:



Each ABAP program consists of the following two parts:

Declaration Part for Global Data, Classes and Selection Screens

The first part of an ABAP program is the declaration part for global data, classes, and selection screens. This consists of:

- All declaration statements for global data. Global data is visible in all internal processing blocks. You define it using declarative statements that appear before the first processing block, in dialog modules, or in event blocks. You cannot declare local data in dialog modules or event blocks.
- All selection screen definitions.
- All local class definitions (CLASS DEFINITION statement). Local classes are part of ABAP Objects, the object-oriented extension of ABAP.

Declaration statements which occur in procedures (methods, subroutines, function modules) form the declaration part for local data in those processing blocks. This data is only visible within the procedure in which it is declared.

Container for Processing Blocks

The second part of an ABAP program contains all of the processing blocks for the program. The following types of processing blocks are allowed:

- Dialog modules (no local data area)
- Event blocks (no local data area)

- Procedures (methods, subroutines and function modules with their own local data area).

Whereas dialog modules and procedures are enclosed in the ABAP keywords which define them, event blocks are introduced with event keywords and concluded implicitly by the beginning of the next processing block.

All ABAP statements (except declarative statements in the declaration part of the program) are part of a processing block. Non-declarative ABAP statements, which occur between the declaration of global data and a processing block are automatically assigned to the START-OF-SELECTION processing block.

Calling Processing Blocks

You can call processing blocks either from outside the ABAP program or using ABAP commands which are themselves part of a processing block. Dialog modules and event blocks are called from outside the ABAP program. Procedures are called using ABAP statements in ABAP programs.

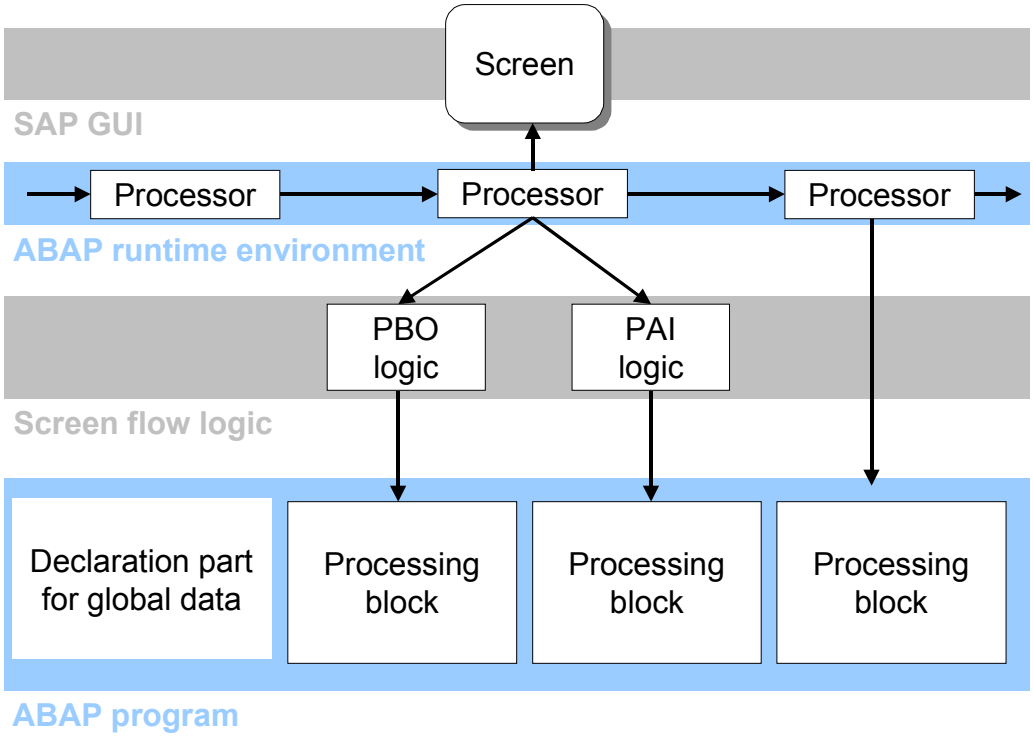
Calling event blocks is different from calling other processing blocks for the following reasons:

An event block call is triggered by an event. User actions on selection screens and lists, and the runtime environment trigger events that can be processed in ABAP programs. You only have to define event blocks for the events to which you want the program to react (whereas a subroutine call, for example, must have a corresponding subroutine). This ensures that while an ABAP program may react to a particular event, it is not forced to do so.

Program Types and Execution

When you run an ABAP program, you call its processing blocks. ABAP programs are controlled from outside the program itself by the processors in the current work process. For the purposes of program flow, we can summarize the screen processor and ABAP processor into the ABAP runtime environment. The runtime environment controls screens and ABAP processing blocks. It contains a range of special control patterns that call screens and processing blocks in certain orders. These sections are also called processors. When you run an ABAP program, the control passes between various processors.

Structure of ABAP Programs



In the R/3 System, there are various types of ABAP program. The program type determines the basic technical attributes of the program, and you must set it when you create it. The main difference between the different program types is the way in which the runtime environment calls its processing blocks.

When you run an application program, you must call at least the first processing block from outside the program, that is, from the runtime environment. This processing block can then either call further processing blocks or return control to the runtime environment. When you start an ABAP program, the runtime environment starts a processor (dependent on the program type), which calls the first ABAP processing block. An ABAP program can be started either by the user or by the system (for example, in background processing), or through an external interface (for example, Remote Function Call).

There are two ways of allowing users to execute programs - either by entering the program name or by entering a transaction code. You can assign a transaction code to any program. Users can then start that program by entering the code in the command field. Transaction codes are also usually linked to a menu path within the R/3 System.

The following program types are relevant to application programming:

Type 1

Type 1 programs have the important characteristic that they do not have to be controlled using user-defined screens. Instead, they are controlled by the runtime environment, which calls a series of processing blocks (and selection screens and lists where necessary) in a fixed sequence. User actions on screens can then trigger further processing blocks.

You can start a type 1 program and the corresponding processor in the runtime environment using the SUBMIT statement in another ABAP program. There are also various ways of starting a

type1 program by entering its program name. This is why we refer to type 1 programs as executable programs.

When you run a type 1 program, a series of processors run in a particular order in the runtime environment. The process flow allows the user to enter selection parameters on a selection screen. The data is them selected from the database and processed. Finally, an output list is displayed. At no stage does the programmer have to define his or her own screens. The runtime environment also allows you to work with a logical database. A logical database is a special ABAP program which combines the contents of certain database tables. The flow of a type 1 program is oriented towards reporting, whose main tasks are to read data from the database, process it, and display the results. This is why executable programs (type 1) in the R/3 System are often referred to as **reports**, and why running an executable program is often called **reporting**.

Since it is not compulsory to define event blocks, you can yourself determine the events to which your ABAP program should react. Furthermore, you can call your own screens or processing blocks at any time, leaving the prescribed program flow. You can use this, for example, to present data in a table on a dialog screen instead of in a list. The simplest executable program (report) contains only one processing block (START-OF-SELECTION).

Executable programs do not require any user dialog. You can fill the selection screen using a variant and output data directly to the spool system instead of to a list. This makes executable programs (reports) the means of background processing in the R/3 System.

You can also assign a transaction code to an executable program. Users can then start it using the transaction code and not the program name. The reporting-oriented runtime environment is also called when you run a report using a transaction code. This kind of transaction is called a **report transaction**.

It is appropriate to use executable programs (reports) when the flow of your program corresponds either wholly or in part to the pre-defined flow of the runtime environment. Until Release 4.5A, the only way to use a logical database was to use an executable program. However, from Release 4.5A, it is also possible to call logical databases on their own.

Type M

The most important technical attribute of a type M program is that it can only be controlled using screen flow logic. You must start them using a transaction code, whcih is linked to the program and one of its screens (initial screen). Another feature of these programs is that you must define your own screens in the Screen Painter (although the intial screen can be a selection screen).

When you start a program using a transaction code, the runtime environment starts a processor that calls the initial screen. This then calls a dialog module in the corresponding ABAP program. The remainder of the program flow can take any form. For example, the dialog module can:

- return control to the screen, after which, the processing passes to a subsequent screen. Each screen has a following screen, set either statically or dynamically.
- call other sequences of screens, selection screens or lists, from which further processing blocks in the ABAP program are started.
- call other processing blocks itself, either internally or externally.
- call other application programs using CALL TRANSACTION (type M program) or SUBMIT (type 1 program).

ABAP programs with type M contain the dialog modules belonging to the various screens. They are therefore known as **module pools**. It is appropriate to use module pools when you write

dialog-oriented programs using a large number of screens whose flow logic largely determines the program flow.

Type F

Type F programs are containers for **function modules**, and cannot be started using a transaction code or by entering their name directly. Function modules are special procedures that you can call from other ABAP programs.

Type F programs are known as **function groups**. Function modules may only be programmed in function groups. The **Function Builder** is a tool in the ABAP Workbench that you can use to create function groups and function modules. Apart from function modules, function groups can contain global data declarations and subroutines. These are visible to all function modules in the group. They can also contain event blocks for screens in function modules.

Type K

You cannot start type K programs using a transaction code or by entering the program name. They are containers for **global classes** in ABAP Objects . Type K programs are known as **class definitions**. The **Class Builder** is a tool in the ABAP Workbench that you can use to create class definitions.

Type J

You cannot start type J programs using a transaction code or by entering the program name. They are containers for **global interface** in ABAP Objects . Type J programs are known as **interface definitions**. Like class definitions, you create interface definitions in the **Class Builder**.

Type S

You cannot start a type S program using a transaction code or by entering the program name. Instead, they are containers for subroutines, which you can call externally from other ABAP programs. Type S programs are known as **subroutine pools**. They cannot contain screens.

Type I

Type I programs - called **includes** - are a means of dividing up program code into smaller, more manageable units. You can insert the coding of an include program at any point in another ABAP program using the INCLUDE statement. There is no technical relationship between include programs and processing blocks. Includes are more suitable for logical programming units, such as data declarations, or sets of similar processing blocks. The ABAP Workbench has a mechanism for automatically dividing up module pools and function groups into include programs.

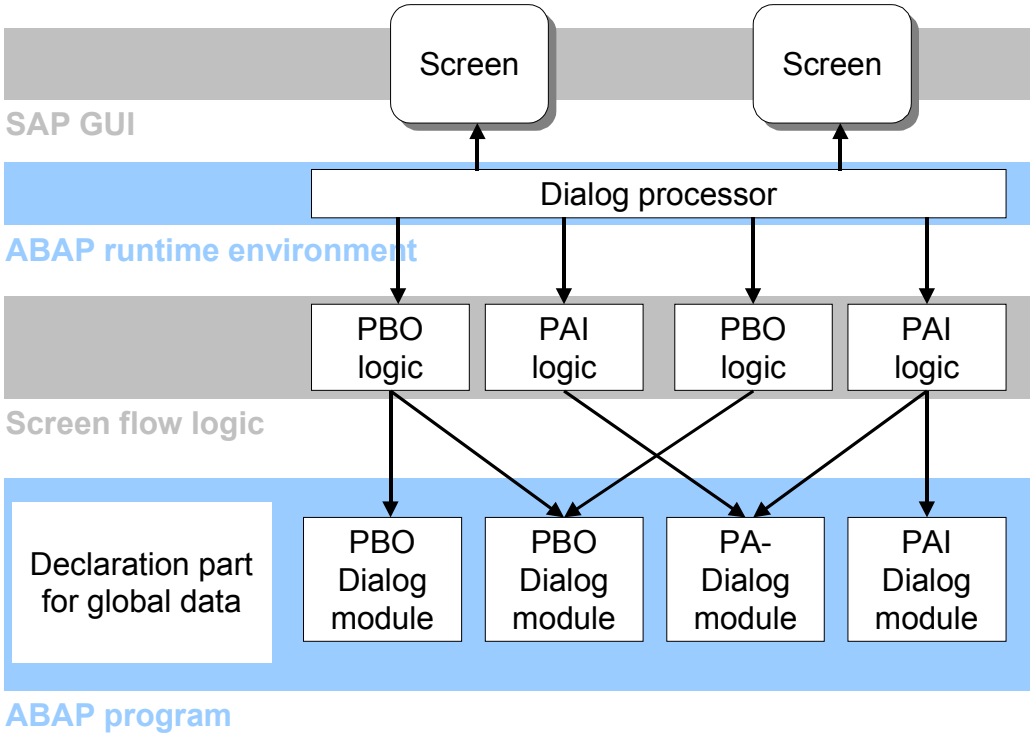
Processing Blocks in ABAP Programs

The following sections explain the different processing blocks in ABAP programs and how they are called.

Dialog Modules

You call dialog modules from the screen flow logic (screen command MODULE). You can write a dialog module in an ABAP program for each state (PBO, PAI; user input) of any of the screens belonging to it. The PAI modules of a screen together with the PBO modules of the subsequent screen form a **dialog step**. The interaction between the flow logic and the screen is controlled by a dialog processor.

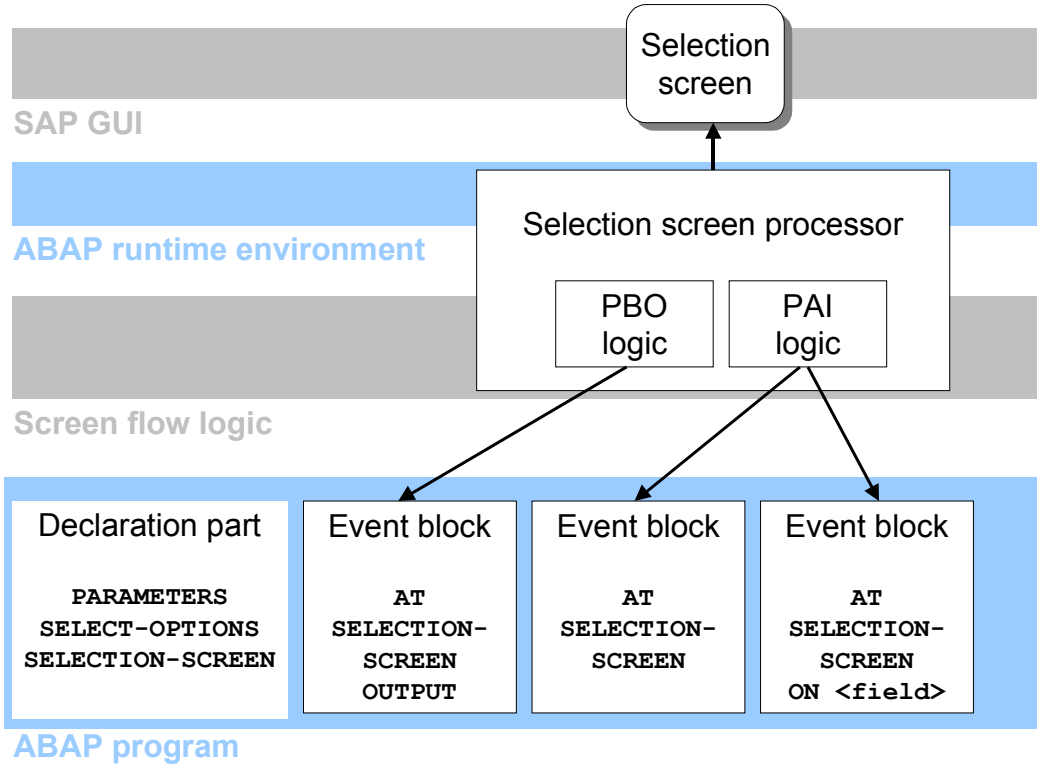
Dialog modules are introduced with the MODULE statement and concluded with the ENDMODULE statement.



Fields on a dialog screen have the same name as a corresponding field in the ABAP program. Data is passed between identically-named fields in the program. You do notdefine dialog screens in the ABAP programs.

Event Blocks for Selection Screens

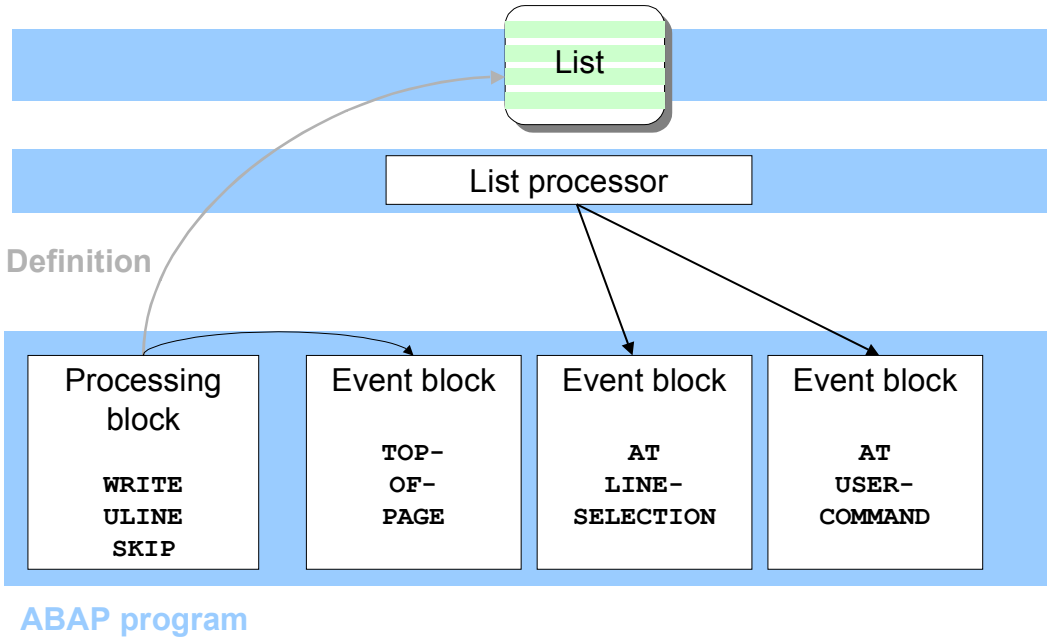
A selection screen is a special kind of dialog screen that you create using ABAP commands in the declaration part of the program. The different events in a selection screen (PAI, PBO, user input), are controlled by a selection screen processor. You can program processing logic for these events in your program. The selection screen processor controls the flow logic of the selection screen.



You do not have to create the screen flow logic yourself, neither can you create your own dialog modules for the PBO or PAI events . Data is passed between selection screen and ABAP program using the fields (parameters and selection tables) which you create in the selection screen definition in the declaration part of the ABAP program.

Event Blocks for Lists

Lists are special screens which output formatted data. You can create them in any processing block in an ABAP program using a special set of commands (such as WRITE, NEW-PAGE and so on). The list processor displays the list on the screen and handles user actions within lists. The list processor controls the flow logic of the list. You do not have to create the screen flow logic yourself, neither can you create your own dialog modules for the PBO or PAI events .

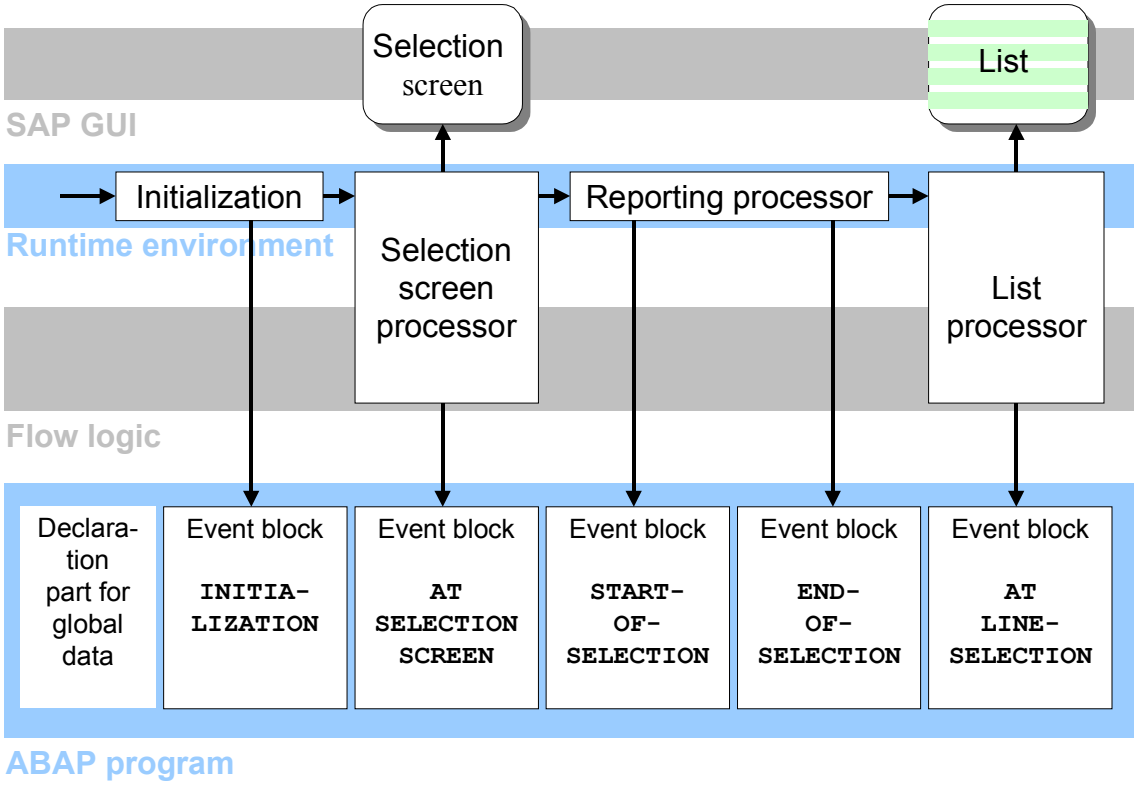


You can call various event blocks while the list is being created which are used in page formatting. The above illustration contains the event block TOP-OF-PAGE, which is called from the ABAP program itself. When the list is displayed, the user can carry out actions which trigger event blocks for interactive list events (such as AT LINE-SELECTION). You can program processing logic for the interactive list events in your program. Data is transferred from list to ABAP program via system fields or an internal memory area called the HIDE area.

Event Blocks for Executable Programs (Reports)

When you run an executable program (type 1), it is controlled by a predefined process in the runtime environment. A series of processors is called, one after the other. These processors trigger events, for which you can define event blocks. Type 1 programs are event-driven.

Processing Blocks in ABAP Programs



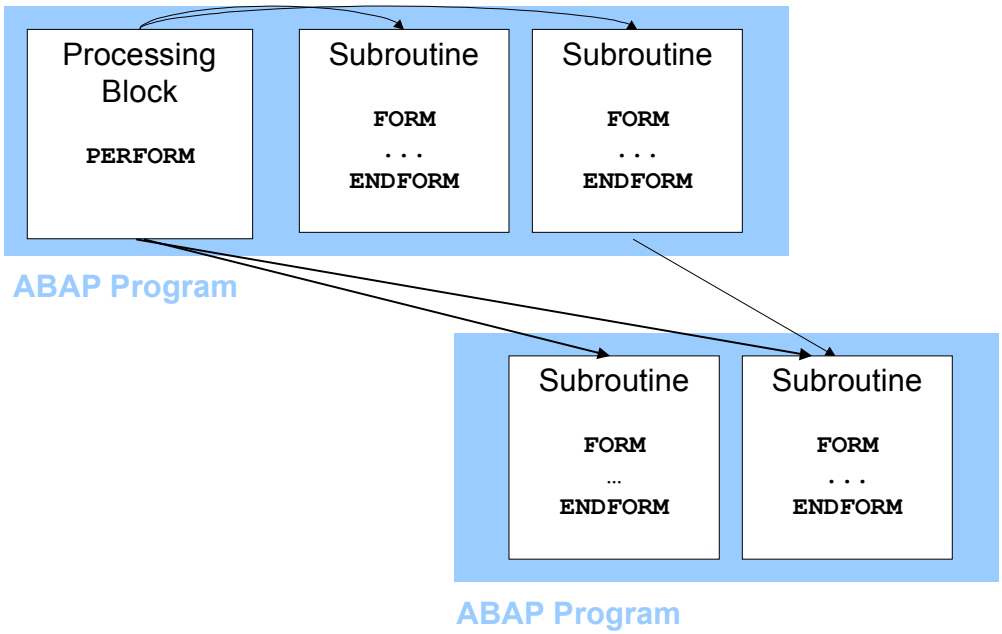
The process is as follows:

1. The runtime environment creates the INITIALIZATION event and calls the corresponding event block (if it has been defined in the ABAP program).
2. If there is a selection screen defined in the program, control returns to the selection screen processor. This generates the corresponding events and calls their event blocks.
3. Control then passes to the reporting processor. It creates the START-OF-SELECTION event and calls the corresponding event block (if it has been defined in the ABAP program).
4. The logical database, if you are using one, calls further event blocks at this point.
5. The reporting processor creates the END-OF-SELECTION event and calls the corresponding event block (if it has been defined in the ABAP program).
6. If the program contains a list description, control now passes to the list processor. The list processor displays the list defined in the ABAP program. It converts user actions on the list into events and calls the corresponding event blocks.

Subroutines

You call subroutines from ABAP programs using the PERFORM statement.

Subroutines are introduced with the FORM statement and concluded with the ENDFORM statement.

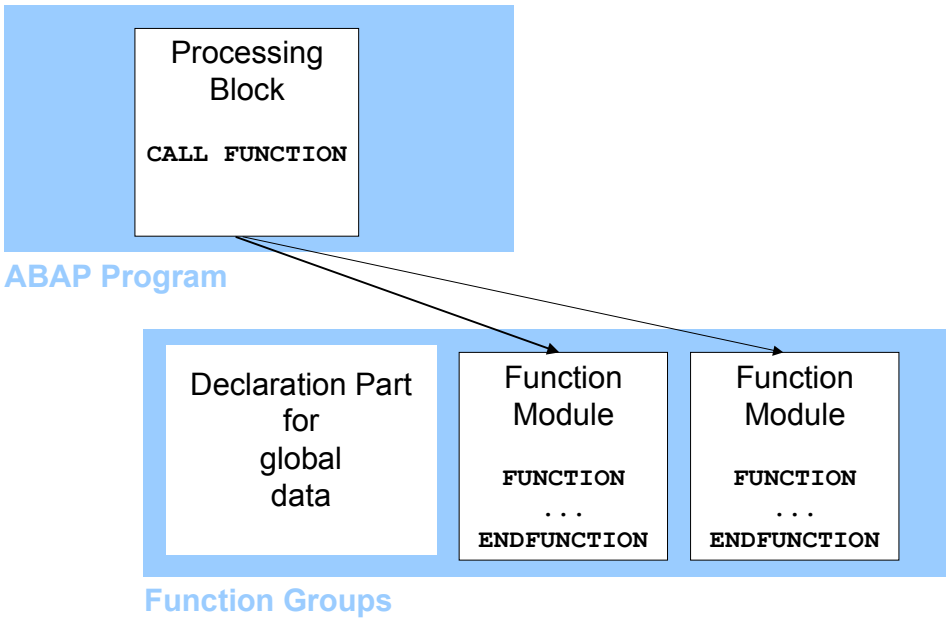


You can define subroutines in any ABAP program. You can either call a subroutine that is part of the same program or an external subroutine, that is, one that belongs to a different program. If you call an internal subroutine, you can use global data to pass values between the main program and the subroutine. When you call an external subroutine, you must pass actual parameters from the main program to formal parameters in the subroutine.

Function Modules

Function modules are external functions with a defined interface. You call function modules from ABAP programs using the CALL FUNCTION statement.

Function modules are introduced with the FUNCTION statement and concluded with the ENDFUNCTION statement.

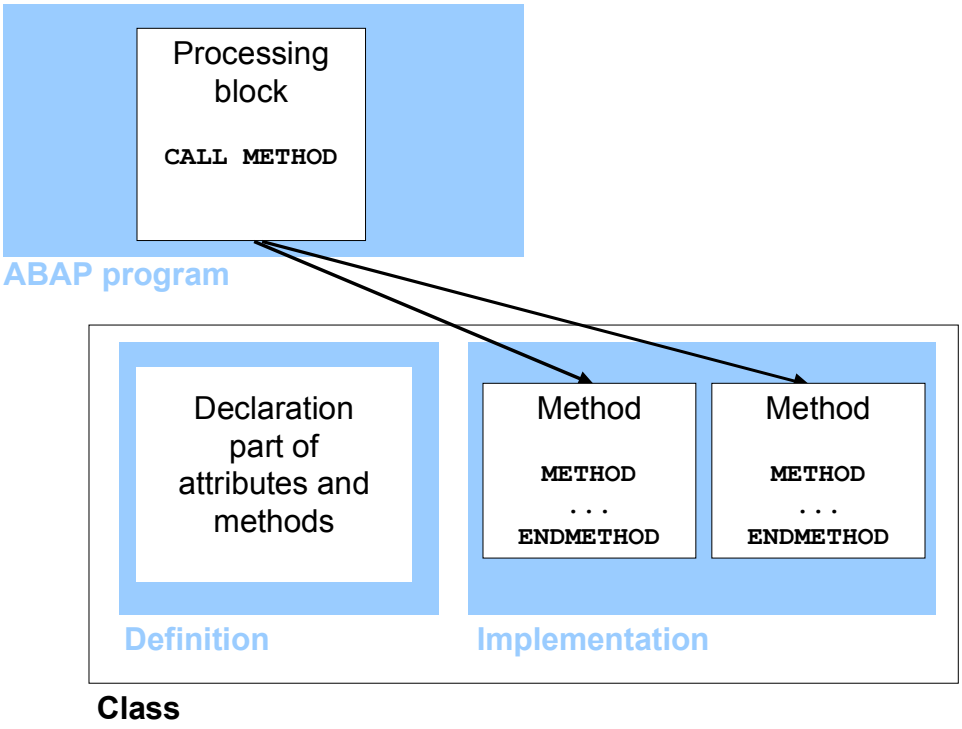


You can only create function groups within special ABAP programs called function groups using the Function Builder. This means that you can only call them externally from other ABAP programs. Unlike subroutines, you always pass data to function modules using an explicit parameter interface.

Methods

Methods describe the functions of classes in ABAP Objects. Like function modules, they have a defined interface. You call methods from ABAP programs using the CALL METHOD statement.

Methods are introduced with the METHOD statement and concluded with the ENDMETHOD statement.



Methods can only be defined in the implementation parts of classes. You can either do this globally in the Class Builder, or locally within ABAP programs. Methods can work with the attributes of their class and with data that you pass to them using their explicit parameter interface.

ABAP Statements

The source code of an ABAP program consists of comments and ABAP statements.

Comments are distinguished by the preceding signs * (at the beginning of a line) and “ (at any position in a line).

ABAP statements always begin with an ABAP keyword and are always concluded with a period (.) . Statements can be several lines long; conversely, a line may contain more than one statement.

ABAP statements use ABAP data types and objects.

Statements and Keywords

The first element of an ABAP statement is the ABAP keyword. This determines the category of the statements. The different statement categories are as follows:

Declarative Statements

These statements define data types or declare data objects which are used by the other statements in a program or routine. The collected declarative statements in a program or routine make up its declaration part.

Examples of declarative keywords:

TYPES, DATA, TABLES

Modularization Statements

These statements define the processing blocks in an ABAP program.

The modularization keywords can be further divided into:

- Event Keywords

  You use statements containing these keywords to define event blocks. There are no special statements to conclude processing blocks - they end when the next processing block is introduced.

  Examples of event keywords are:

  AT SELECTION SCREEN, START-OF-SELECTION, AT USER-COMMAND

- Defining keywords

  You use statements containing these keywords to define subroutines, function modules, dialog modules and methods. You conclude these processing blocks using the END- statements.

  Examples of definitive keywords:

  FORM ..... ENDFORM, FUNCTION ... ENDFUNCTION,
  MODULE ... ENDMODULE.

Control Statements

You use these statements to control the flow of an ABAP program within a processing block according to certain conditions.

Examples of control keywords:

IF, WHILE, CASE

Call Statements

You use these statements to call processing blocks that you have already defined using modularization statements. The blocks you call can either be in the same ABAP program or in a different program.

Examples of call keywords:

PERFORM, CALL, SET USER-COMMAND, SUBMIT, LEAVE TO

Operational Statements

These keywords process the data that you have defined using declarative statements.

Examples of operational keywords:

WRITE, MOVE, ADD

Database Statements

These statements use the database interface to access the tables in the central database system. There are two kinds of database statement in ABAP: Open SQL and Native SQL.

Open SQL

Open SQL is a subset of the standard SQL92 language. It contains only Data Manipulation Language (DML) statements, such as SELECT, IINSERT, and DELETE. It does not contain any Data Definition Language (DDL) statements (such as CREATE TABLE or CREATE INDEX). Functions of this type are contained in the ABAP Dictionary. Open SQL contains all of the DML functions from SQL92 that are common to all of the database systems supported by SAP. It also contains a few SAP-specific functions. ABAP programs that use only Open SQL statements to access the database are fully portable. The database interface converts the OPEN SQL commands into commands of the relevant database.

Native SQL

Native SQL statements are passed directly from the database interface to the database without first being converted. It allows you to take advantage of all of your database's characteristics in your programs. In particular, it allows you to use DDL operations. The ABAP Dictionary uses Native SQL for tasks such as creating database tables. In ordinary ABAP programs, it is not worth using DDL statements, since you cannot then take advantage of the central administration functions of thie ABAP Dictionary. ABAP programs that use Native SQL statements are database-specific, because there is no standardized programming interface for SQL92.

Data Types and Objects

The physical units with which ABAP statements work at runtime are called internal program data objects. The contents of a data object occupy memory space in the program. ABAP statements access these contents by addressing the name of the data object. For example, statements can write the contents of data objects in lists or in the database, they can pass them to and receive them from routines, they can change them by assigning new values, and they can compare them in logical expressions.

Each ABAP data object has a set of technical attributes, which are fully defined at all times when an ABAP program is running. The technical attributes of a data object are: Data type, field length, and number of decimal places.

ABAP Statements

The data type determines how the contents of a data object are interpreted by ABAP statements. As well as occurring as attributes of a data object, data types can also be defined independently. You can then use them later on in conjunction with a data object. You can define data types independently either in the declaration part of an ABAP program (using the TYPES statement), or in the ABAP Dictionary.

The data types you will use in a program depend on how you will use your data objects. The task of an ABAP program can range from passing simple input data to the database to processing and outputting a large quantity of structured data from the database. ABAP contains the following data types:

Predefined and User-defined Elementary Types

There are five predefined non-numeric data types:

Character string (C), Numeric character string (N), Date (D), Time (T) and Hexadecimal (X)

and three predefined numeric types:

Integer (I), Floating-point number (F) and Packed number (P).

The **field length** for data types D, F, I, and T is fixed. The field length determines the number of bytes that the data object occupies in memory. In types C, N ,X and P, the length is not part of the type definition. Instead, you define it when you declare the data object in your program.

Data type P is particularly useful for exact calculations in a business context. When you define an object with type P, you also specify a number of **decimal places**.

You can also define your own elementary data types in ABAP using the TYPES statement. You base these on the predefined data types. This determines all of the technical attribtues of the new data type. For example, you could define a data type P_2 with two decimal places, based on the predefined data type P. You could then use this new type in your data declarations.

You use elementary data types to define individual elementary data objects. You use these object to transfer input and output values, as auxiliary fields in calculations, to store intermediate results, and so on. The aggregated data types described below are made up of elementary data types.

An aggregated data type can be a **structure** or an **internal table**.

Structures

A structure is a user-defined sequence of data types. It fully defines the data object. You can either access the entire data object, or its individual components. ABAP has no predefined structures. You therefore need to define your own structures, either in the ABAP program in which you want to use it, or in the ABAP Dictionary.

You use structures in ABAP programs to group work areas that logically belong together. Since the individual elements within a structure can be of any type, and can also themselves be structures or internal tables, the possible uses of structures are very wide-ranging. For example, you can use a structure with elementary data types to display lines from a database table within a program. You can also use structures containing aggregated elements to include all of the attributes of a screen or control in a single data object.

Internal Tables

Internal tables consists of a series of lines that all have the same data type.

Internal tables are characterized by:

- Their line type.

  The line type of an internal table can be any ABAP data type - an elementary type, a structure or an internal table.

- A key

  The key of an internal table is used to identify its entries. It is made up of the elementary fields in the line. The key may be unique or non-unique.

- The access type

  The access method determines how ABAP will access individual table entries. There are three access types, namely unsorted tables, sorted index tables and hash tables.
  For index tables, the system maintains a linear index, so you can access the table either by specifying the index or the key.
  Hashed tables have no linear index. You can only access hashed tables by specifying the key. The system has its own hash algorithm for managing the table.

You should use internal tables whenever you need to use structured data within a program. One imprint use is to store data from the database within a program.

Data Types for Reference

You use references to refer to objects in ABAP Objects. References are stored in reference variables. The data type of the reference determines how it is handled in the program. There are different types for class and interface variables.

Reference variables in ABAP are treated like other elementary data objects. This means that a reference variable can occur as a component of a structure or internal table as well as on its own.

Declaring Data Objects

Apart from the interface parameters of routines, you declare all of the data objects in an ABAP program or routine in its declaration part. The declarative statements establish the data type of the object, along with any missing technical attributes, such as its length or the number of decimal places. This all takes place before the program is actually executed. The exception to this are internal tables.

When you declare an internal table, you specify the above details. However, you do not need to specify the overall size of the data object. Only the length of a row in an internal table is fixed. The number of rows (the actual length of the data object in memory) is adapted dynamically at runtime. In short, internal tables can be extended dynamically while retaining a fixed structure.

The interface parameters of routines are generated as local data objects, but not until the routine is called. You can define the technical attributes of the interface parameters in the routine itself. If you do not, they adopt the attributes of the parameters from which they receive their values.

Logical Databases and Contexts

This section introduces logical databases and contexts - two methods that make it easier to read data from database tables. Both of them encapsulate Open SQL statements in separate ABAP programs.

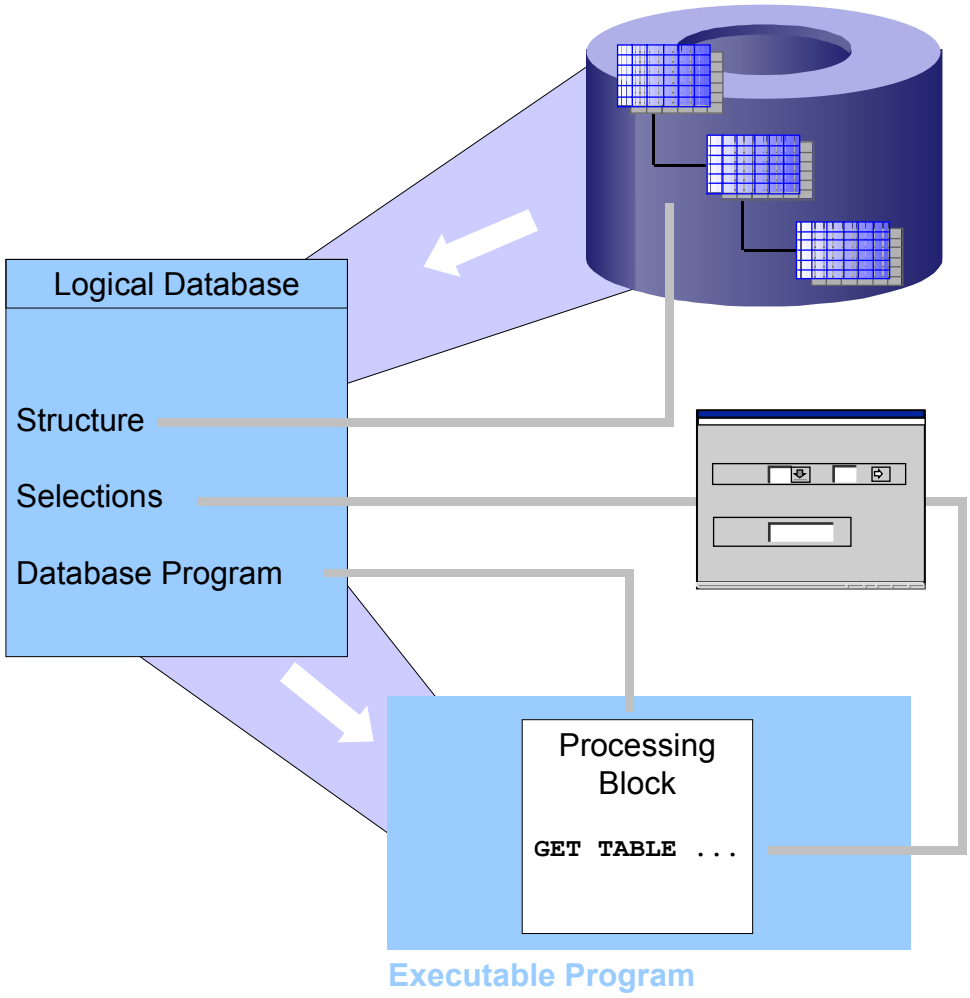
Logical Databases

Logical databases are special ABAP programs that read data from database tables. They are used by executable (type 1) programs. At runtime, you can regard the logical database and the executable program (reports) as a single ABAP program, whose processing blocks are called by the runtime environment in a particular, pre-defined sequence.

You edit logical databases using a tool within the ABAP Workbench, and link them to executable programs (reports) when you enter the program attributes. You can use a logical database with any number of executable programs (reports). From Release 4.5A, it is also possible to call logical databases on their own.

Structure of a Logical Database

The following diagram shows the structure of a logical database, which can be divided into three sections:



Structure

The structure of a logical database determines the database tables which it can access. It adopts the hierarchy of the database tables defined by their foreign key relationships. This also controls the sequence in which the tables are accessed.

Selection Part

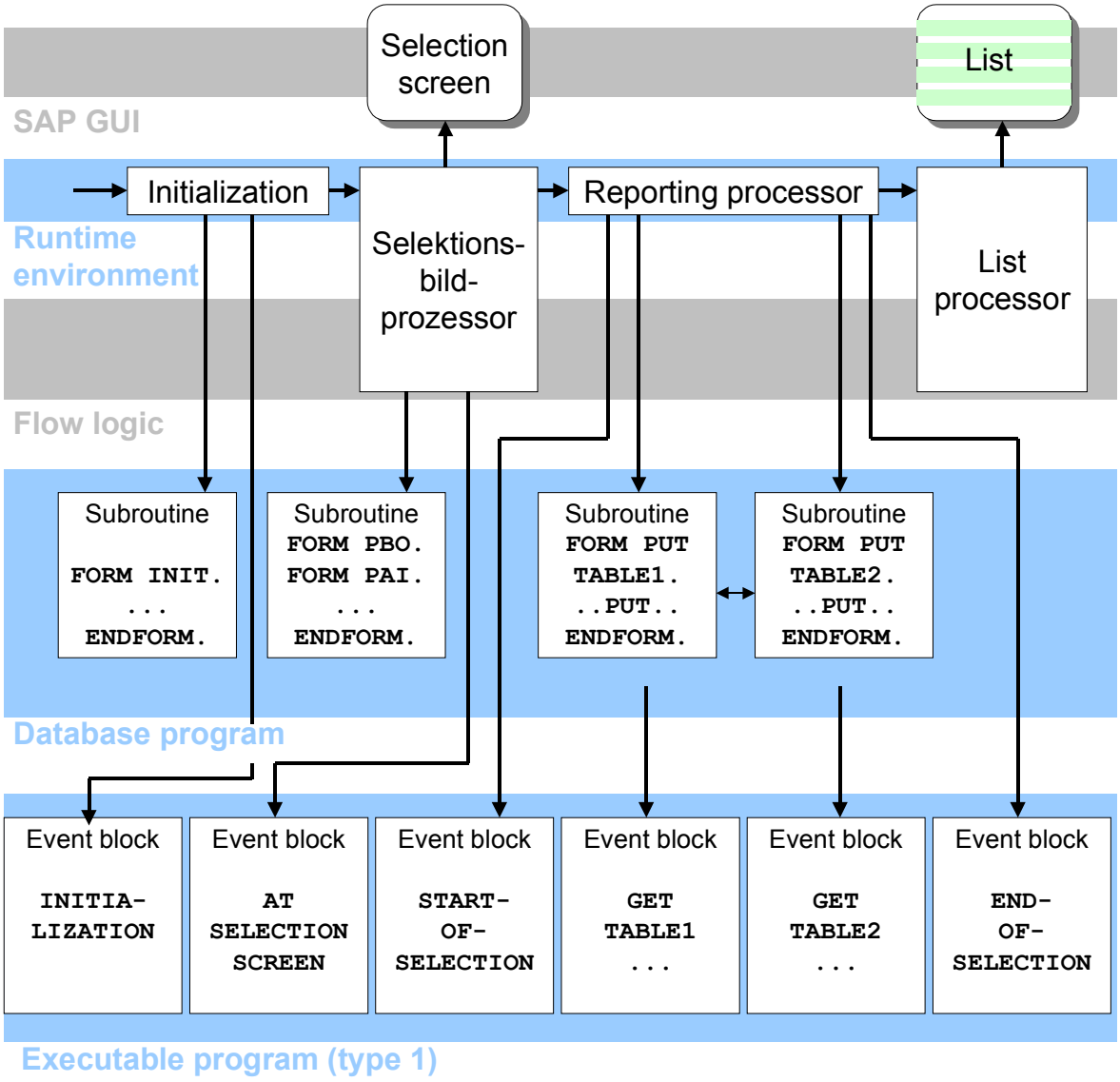
The selection part of the logical database defines input fields for selecting data. The runtime environment displays these on the selection screen when you run an executable program linked to the logical database. The corresponding fields are also available in the ABAP program, allowing you, for example, to change their values to insert default values on the selection screen.

Database Program

The database program of a logical database is a container for special subroutines, in which the data is read from the database tables. These subrotuines are called by the reporting processor in the runtime environment in a predefined sequence.

Running Type 1 Programs with a Logical Database

The following diagram shows the principal processing blocks that are called when you run an executable program linked to a logical database:



The runtime environment calls depend both on the structure of the logical database and on the definition of the executable program. The structure of the logical database determines the sequence in which the processing blocks of the logical database are called. These in turn call GET event blocks in the executable program. These GET event blocks determine the read depth in the structure of the logical database. TABLES or NODES statements in the declaration part of the executable program determine which of the input fields defined in the logical database are included in the selection screen. They also define interface work areas for passing data between the logical database and the executable program.

The actual access to the R/3 System database is made using OPEN SQL statements in the PUT_<TABLE> subroutines. The data that is read is passed to the executable program using the interface work areas (defined using the TABLES statement). Once the data has been read in the logical database program, the executable program (report) can process the data in the GET event blocks. This technique separates data reading and data processing.

Uses of Logical Databases

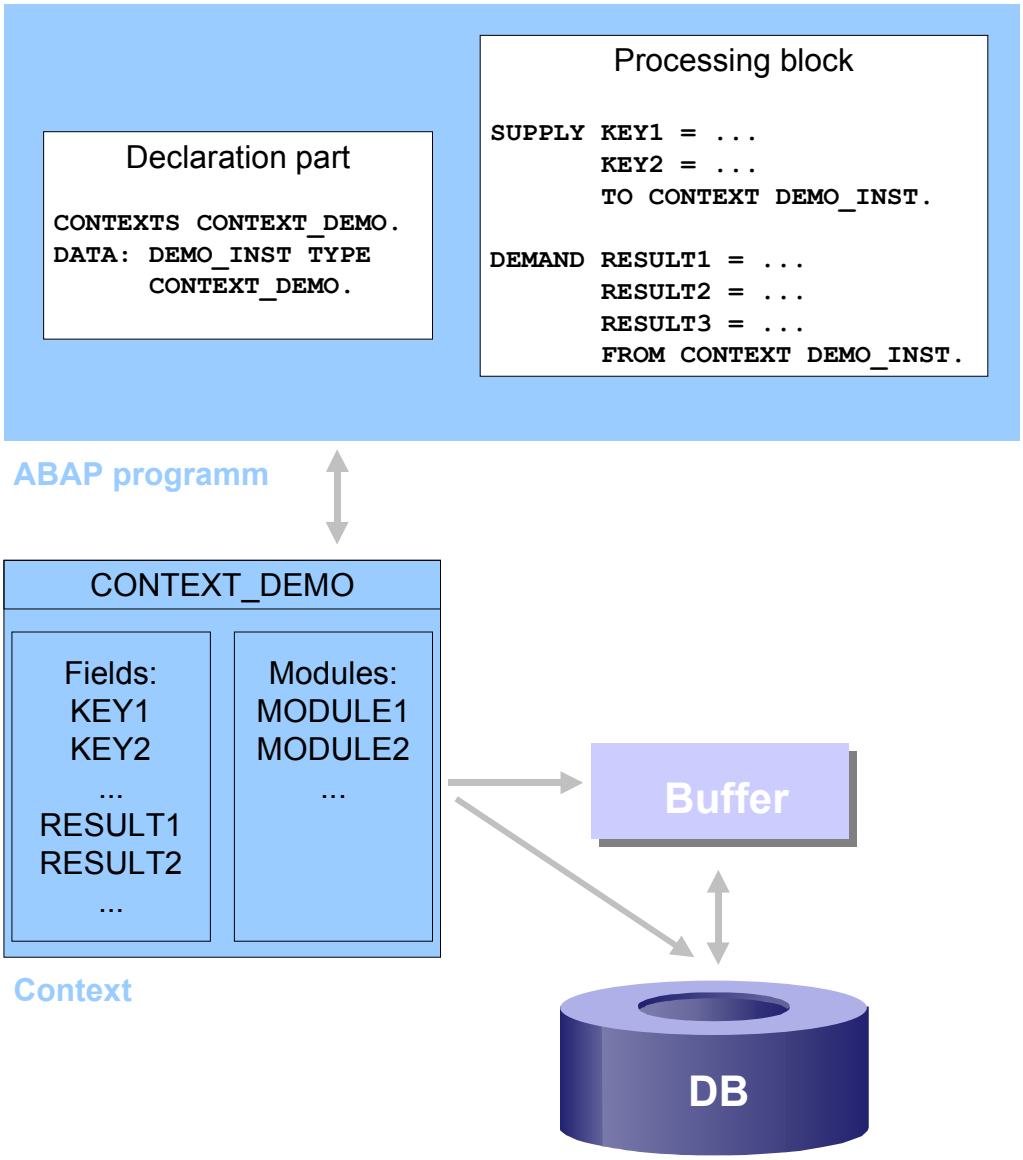
The main use of logical databases is to make the code that accesses data in database tables reusable. SAP supplies logical databases for all applications. These have been configured for optimal performance, and contain further functions such as authorization checks and search helps. It is appropriate to use logical databases whenever the database tables you want to read correspond largely to the structure of the logical database and where the flow of the system program (select - read - process - display) meets the requirements of the application.

Contexts

In application programming, you often use a relatively small set of basic data to derive further data. This basic data might, for example, be the data that the user enters on the screen. The relational links in the database are often used to read further data on the basis of this basic data, or further values are calculated from it using ABAP statements.

It is often the case that certain relationships between data are always used in the same form to get further data, either within a single program or in a whole range of programs. This means that a particular set of database accesses or calculations is repeatedly executed, despite the fact that the result already exists in the system. This causes unnecessary system load, which can be alleviated by using contexts.

You define contexts in the Context Builder, which is part of the ABAP Workbench. They consist of key input fields, the relationships between the fields, and the other fields and values that you can derive from them. Contexts can link these derived fields by foreign key relationships between tables, by function modules, or by other contexts.



In application programs, you work with instances of a context. You can use more than one instance of the same context. The application program supplies input values for the key fields in the context using the SUPPLY statement, and can query the derived fields from the instance using the DEMAND statement.

Each context has a cross-transaction buffer on the application server. When you query an instance for values, the context program searches first of all for a data record containing the corresponding key fields in the appropriate buffer. If one exists, the data is copied to the instance. If one does not exist, the context program derives the data from the key field values supplied and writes the resulting data record to the buffer.

Memory Structures of an ABAP Program

In the Overview of the R/3 Basis System [Page 20] you have seen that each user can open up to six R/3 windows in a single SAPgui session. Each of these windows corresponds to a session on the application server with its own area of shared memory.

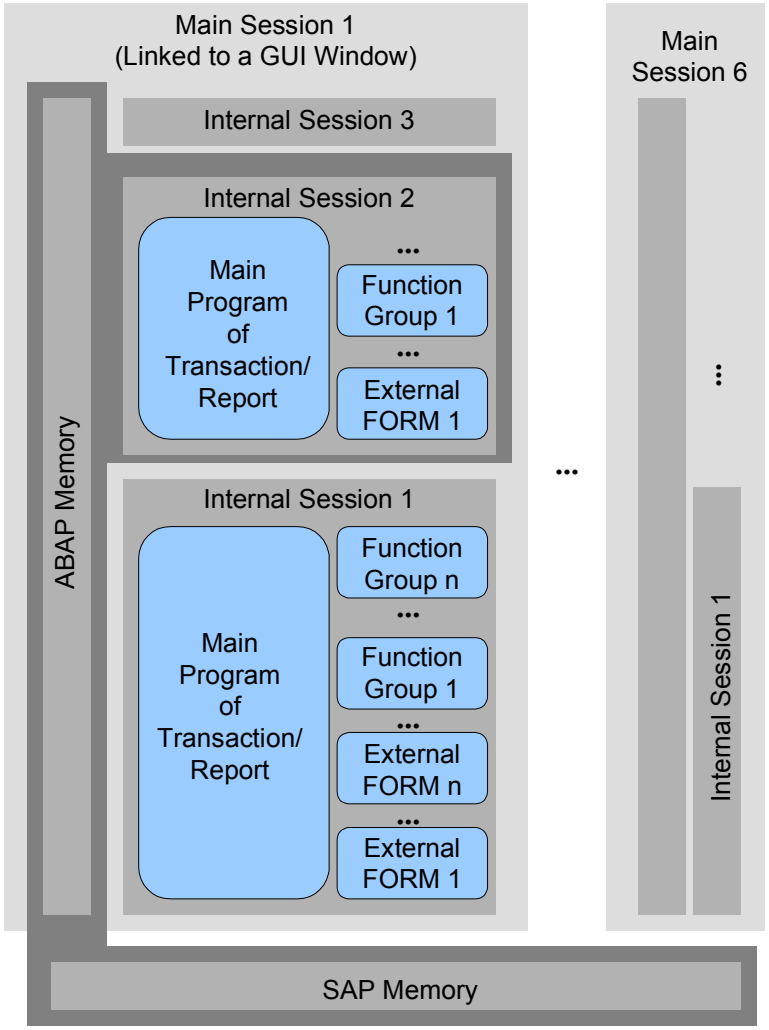
The first application program that you start in a session opens an internal session within the main session. The internal session has a memory area that contains the ABAP program and its associated data. When the program calls external routines (methods, subroutines or function modules) their main program and working data are also loaded into the memory area of the internal session.

Only one internal session is ever active. If the active application program calls a further application program, the system opens another internal session. Here, there are two possible cases: If the second program does not return control to the calling program when it has finished running, the called program replaces the calling program in the internal session. The contents of the memory of the calling program are deleted. If the second program does return control to the calling program when it has finished running, the session of the called program is not deleted. Instead, it becomes inactive, and its memory contents are placed on a stack.

The memory area of each session contains an area called ABAP memory. ABAP memory is available to all internal sessions. ABAP programs can use the EXPORT and IMPORT statements to access it. Data within this area remains intact during a whole sequence of program calls. To pass data to a program which you are calling, the data needs to be placed in ABAP memory before the call is made. The internal session of the called program then replaces that of the calling program. The program called can then read from the ABAP memory. If control is then returned to the program which made the initial call, the same process operates in reverse.

All ABAP programs can also access the SAP memory. This is a memory area to which all sessions within a SAPgui have access. You can use SAP memory either to pass data from one program to another within a session, or to pass data from one session to another. Application programs that use SAP memory must do so using SPA/GPA parameters (also known as SET/GET parameters). These parameters are often used to preassign values to input fields. You can set them individually for users, or globally according to the flow of an application program. SAP memory is the only connection between the different sessions within a SAPgui.

The following diagram shows how an application program accesses the different areas within shared memory:



In the diagram, an ABAP program is active in the second internal session of the first main session. It can access the memory of its own internal session, ABAP memory and SAP memory. The program in the first internal session has called the program which is currently active, and its own data is currently inactive on the stack. If the program currently active calls another program but will itself carry on once that program has finished running, the new program will be activated in a third internal session.

Creating and Changing ABAP Programs

SAP Easy Access [Ext.]

ABAP programs are objects of the R/3 Repository. Like all other Repository objects, you maintain them using an ABAP Workbench [Ext.] tool - in this case, the ABAP Editor.

This section provides a brief description of the ABAP Workbench and an overview of how to create and edit ABAP programs. It describes the different ways of starting the ABAP Editor. In the text below, 'open a program' always means 'start the ABAP Editor and use it to open a program'.

Starting the ABAP Editor

To start the ABAP Editor to create or change ABAP programs, the R/3 system offers three possibilities:

- Using the Repository Browser [Page 70]

  The Repository Browser in the ABAP Workbench (transaction SE80) offers a hierarchical overview of all R/3 Repository objects, ordered by development class, user name of the programmer, object type, and so on. If you enter a program name, you can directly access all of its components, such as the main program, subroutines, and global data. If you select a program object in the Repository Browser and choose *Change*, the system automatically opens the appropriate tool; in this case, the ABAP Editor.

  This method is suitable for complex programs, since the Repository Browser always provides you with an overview of all of the program components.

- Using the ABAP Editor [Page 73]

  You can open a program directly by choosing ABAP Editor from the initial screen of the ABAP Workbench (Transaction SE38). If you want to change a program using this method, you must already know its name and environment.

  This procedure is suited for maintaining or creating relatively simple or short programs, which have few or no additional components.

- Using Forward Navigation [Page 74]

  In any of the tools in the ABAP Workbench, you can open a different Repository object by positioning the cursor on it and double-clicking. The system automatically opens the object using the correct tool. This also applies to ABAP programs.

  Forward navigation by double-clicking is possible wherever an ABAP program is called from another object, such as screen flow logic or another program.

Naming ABAP Programs

The name of an ABAP program can be between 1 and 30 characters long. It cannot contain any of the following characters: Period (.), comma (,), space (), parentheses (), apostrophe (‘), inverted commas (“), equals sign (=), asterisk (*), accented characters or German umlauts (à, é, ø, ä, ß, and so on), percentage signs (%), or underscores (_).

Program Attributes

Like many other Repository objects, ABAP programs have attributes, which are important in determining the function of the program within the R/3 System. For an overview of program attributes, refer to Maintaining Program Attributes [Page 75].

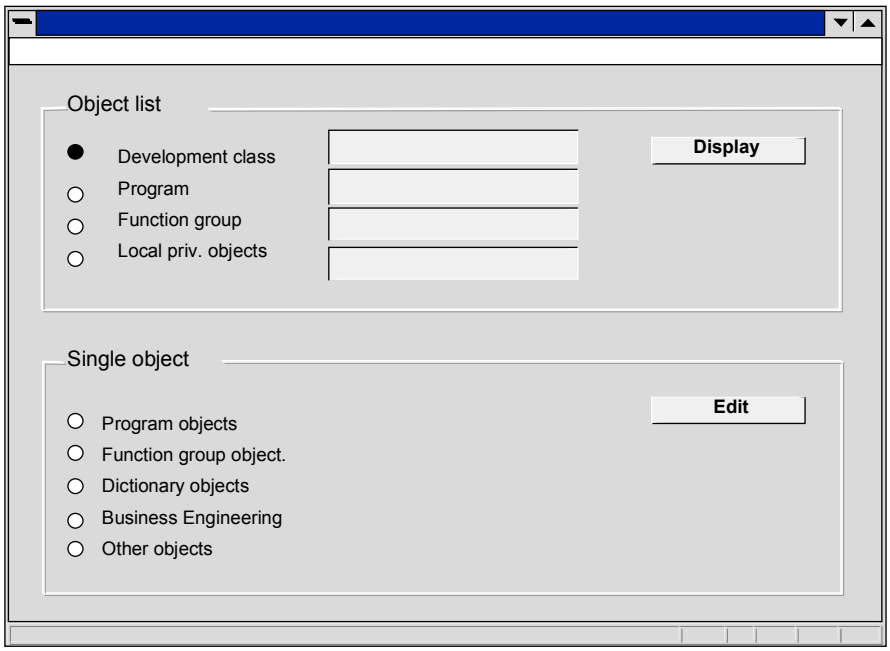
Source Code

ABAP source code defines the processing logic of R/3 application programs. For an introduction to writing source code, refer to Editing Programs [Page 79].

Opening a Program from the Repository Browser

SAP Easy Access [Ext.]

To open ABAP programs from the Repository Browser, choose *Repository Browser* from the *ABAP Workbench* screen, or start Transaction SE80.



Here you can enter a program name directly or display a list of all programs of a certain development class.

Opening a Specific Program

Enter a program name in the object list and choose *Display*.

If the program does not yet exist, a dialog screen appears, asking you whether to create the program. Otherwise, the Repository Browser opens the specified program.

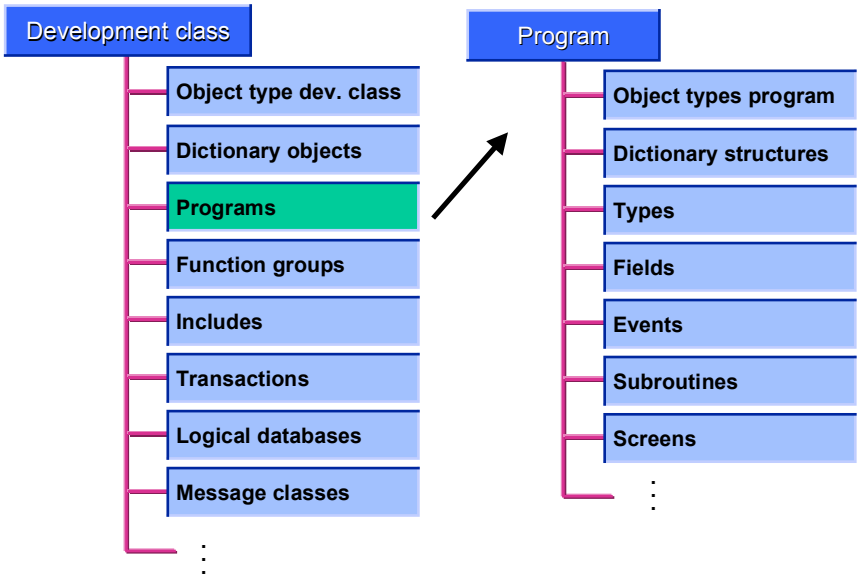
Creating a New Program

1. The dialog box *Create Program* appears, which allows you to create a *TOP INCL* (top include program). A Top include is a special include program in which you write your global data declarations. When you create a module pool, begin the name with 'SAPM'. This allows the system to propose a default name for the Top include that is automatically included in the ABAP Workbench navigation functions.
2. The *ABAP: Program Attributes* screen appears, on which you must enter the program attributes. Maintaining the program attributes is an important procedure when creating a program. Enter the program attributes.
3. Save the program attributes.

The program now exist in the R/3 Repository. You can now either branch directly to the ABAP Editor by choosing *Source code*, or open the program using the Repository Browser.

Displaying the Programs in a Development Class

Enter the name of an existing development class in the object list and choose *Display*. The system displays a hierarchical overview of all R/3 Repository objects belonging to the specified development class.



You now have several possibilities:

- If the *Programs* node does not exist on the screen, you can create a program by positioning the cursor on the *Object Types Development Class* node and choosing *Create*.

  A dialog box appears (*Development Objects*).Choose the entry *Program objects*. In the subsequent dialog box (*Program Objects*) choose the entry *Program.* Enter a name and choose *Create* again.

  The *Create Program* dialog box appears. From here, carry on as described in *Creating a New Program* above.

- If the *Programs* node already exists on the screen, you can position the cursor on it and choose *Create* to create a new program.

- In this case, you can also expand the node to display all of the programs belonging to the development class.

  Position the cursor on a program name and double-click or choose *Display* or *Change*. This opens the ABAP Editor for the program.

Opening a Program from the Repository Browser

The Repository Browser displays the program as a tree structure with the program as its root node and the individual program components as subnodes. Note that selecting other components of the program does not open the ABAP Editor to display the source code of the program, but always the appripriate tool for the selected component. In special cases, such as for include programs, this can also be the ABAP Editor. However, you then edit only the component and not the source code of the entire program.

Opening Programs in the ABAP Editor

SAP Easy Access [Ext.]

To open ABAP programs directly using the ABAP Editor, choose *ABAP Editor* in the *ABAP Workbench* screen or start Transaction SE38, and enter a program name.

Creating a New Program

If the program does not exist, choose *Create*. The *ABAP: Program Attributes* screen appears, on which you must enter the program attributes.

If you create a program in this way, the system does **not** offer to create a Top include. This way of creating programs is therefore best suited for reports and short test programs. Once you have entered and saved the program attributes, the new program exists in the R/3 Repository.

Maintaining an Existing Program

To edit an existing program, enter its name on the initial screen of the ABAP Editor (Transaction SE38), select one of the following components, and then choose *Display* or *Change*.

- *Source Code*

  Starts the ABAP Editor.

- *Variants*

  Starts the variant maintenance tool. Variants allow you to define fixed values for the input fields on the selection screen of a report.

- *Attributes*

  Allows you to change the program attributes (see below).

- *Documentation*

  Allows you to write documentation for a particular executable program (report). The system starts the SAP*script* Editor, where you can enter the documentation according to the predefined template. When executing the report, the user can display this documentation by choosing *System* → *Services* → *Reporting* (Transaction SA38) and *Goto* → *Documentation*. If the report is stored as node in a reporting tree (Transaction SERP), the user can choose *Goto* → *Display docu.* from the tree display to display the documentation.

- *Text elements*

  Allows you to edit the text elements of the program. Text elements are all texts that appear on the selection screen or on the output screen of a report.

You can also access any of these components by using the *Goto* menu in the ABAP Editor itself.

Opening Programs Using Forward Navigation

If you are already working in the ABAP Workbench, you can open a program by positioning the cursor on the program name and double-clicking.

Other ways of foward navigation are always available in the menus of the workbench tools, usually under *Goto* or *Environment*.

The following examples show some of the possibilites of forward navigation.

Suppose, you are editing a program and find the line

SUBMIT ZZHKTST 1.

This statement calls an executable program (report) from within another ABAP program.

If you position the cursor on the name ZZHKTST1 and double-click, there are two possibilities:

1. If the program ZZHKTST1 already exists:

   The system opens ZZHKTST1 in the ABAP Editor, and you can read or edit it.

2. If the program ZZHKTST1 does not exist:

   A dialog box appears, asking you if you want to create the program.

After editing ZZHKTST1, you can choose *Back* to return to the ABAP Editor session of the original program.

In the Repository Browser, open the hierarchy tree of a program that contains at least one screen. Position the cursor on a screen under the *Screen* node.

If you choose *Display*, *Change*, or double-click the screen name, the system opens the Screen Painter and displays the flow logic of the screen.

This usually contains a series of MODULE statements that call ABAP modules in the program.

Position the cursor on a module name and double-click. The system opens the ABAP Editor for the corresponding include program at the position in the program where the relevant module is programmed. The include program is used to modularize the source code of the main program.

If you position the cursor on the name of an include in an INCLUDE statement in the ABAP Editor and double-click, the system opens the include program in the Editor.

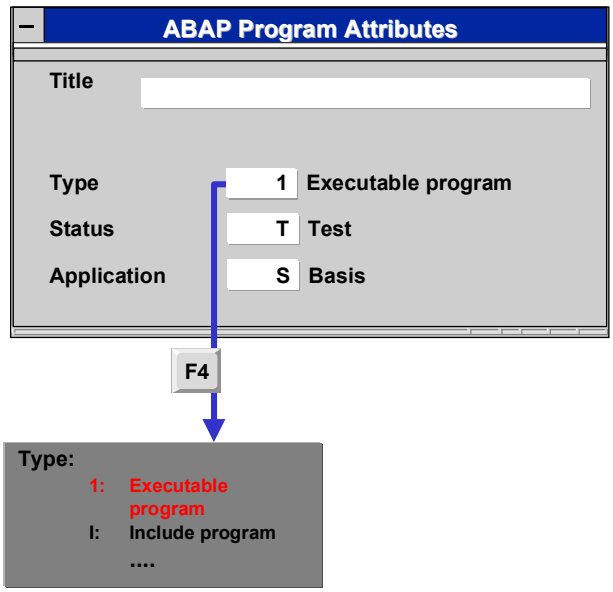
Maintaining Program Attributes

BC - Benutzer und Rollen [Ext.]

In the program attributes, you set the runtime environment of a program, and thus determine how it runs. The most important attribute is the program type. This specifies how the program can be executed.

The program attributes also tell you the application to which the program belongs, and, in the case of executable programs (reports), the name of any associated logical database.

Take care to enter the correct program attributes, otherwise the system will not be able to run the program properly. You maintain the program attributes on the *ABAP: Program Attributes* screen.



To create an executable program (report), enter 1 in the *Type* field. To create a module pool, enter M in the *Type* field. For a list of other possible types, use the possible entries button.

If you create a report (type = 1), choose `Enter`.

The system automatically displays the input fields for report-specific attributes. Only now are the additional input fields *Logical database, from application,* and *Selection screen* visible.

Overview of all program attributes

The following section provides information about program attributes. Note that some of these attributes only apply to executable programs (reports), and not to other ABAP program types. The field help and possible values help for the fields on the *ABAP: Program Attributes* screen provide further information.

Maintaining Program Attributes

Version

These fields are used for version administration. The system fills them.

Title

In the required entry field *Title* enter a program description that describes the function of the program. The system automatically includes the title into the text elements of the program. Thus, you can edit the title when maintaining the text elements.

Maintenance Language

The maintenance language is the logon language of the user who creates the program. The system fills this field automatically. You can change the maintenance language, if you maintain the program or its components in another logon language.

Type

In the *Type* field, you must specify the execution mode of your program.

Use **Type 1** (report) to declare your program as executable.. This means that the program can run on its own, and that you can start it in the R/3 system without a transaction code. You can also run executable programs (reports) in the background.

Use **Type M** to declare your program as a module pool. This means that your program **cannot** run on its own, but serves as a frame for program modules used for dialog programming. These program modules contain the application logic of a transaction and are called by a separately programmed screen flow logic (programming screens using the Screen Painter tool). The screen flow logic itself can be called via a transaction code only.

Apart form type 1 (for executable programs (reports)) and type M (for module pools), you should also know **Type I** for include programs. An inlcude program is an independent program with two main functions: On one hand, it contains program code that can be used by different programs. On the other hand, it modularizes source code, which consists of several different, logically related parts. Each of these parts is stored in a different include program. Include programs make your source code easier to read and maintain.

Status

This entry describes the status of the program development; for example, T for test program.

Application

This field contains the short form of your application, for example, **F** for Financial accounting. This required entry enables the system to allocate the program to the correct business area.

Authorization Group

In this field, you can enter the name of a program group. This allows you to group different programs together for authorization checks. The group name is a field of the two authorization objects S_DEVELOP (program development and program execution) and S_PROGRAM (program maintenance). Thus, you can assign authorizations to users according to program groups. For more information about creating function modules, refer to the Users and Authorizations documentation.

Development Class

The development class is important for transports between systems. You combine all Workbench objects assigned to one development class in one transportation request.

If you are working in a team, you may have to assign your program to an existing development class, or you may be free to create a new class. All programs assigned to the development class $TMP are private objects and cannot be transported into other systems. You can enter the development class directly into this field. Otherwise, the system prompts for it when you save the attributes.

Choosing *Local object* is equivalent to entering $TMP in the field *Development class*. You can change the development class of a program later on by choosing, for example, *Program* → *Reassign* in the *ABAP: Program Attributes* screen.

Logical Database from Application

Only for Executable Programs (Reports)

These attributes determine the logical database used by the executable program (report) to read data, and the application to which it belongs. Logical databases have unique names within their application. However, systemwide, you can have more than one logical database with the same name. This is why you also need to specify the application.

If you read data directly in your program instead of using a logical database, you should enter an application, but leave the *logical database* field empty.

Selection Screen Version

Only for Executable Programs (Reports)

If you do not specify a selection screen version, the system automatically creates a selection screen based on the selection criteria of the logical datbase and the parameters and select-options statements in the program.

If you want to use a different selection screen, enter the number here (not 1000, since this is reserved for the standard selection screen). The number must be smaller than 1000 and correspond to an additional selection screen of the logical database. The possible values help displays a list of available selection screens. You can also look in the selection include of the logical database (program DBxxxSEL, where xxx is the name of the logical database).

Upper/Lower Case

If you do not want the ABAP Editor to change the case of your code when you display the program, leave this field empty. If you select it, the program code (apart from literals and comments) is converted to uppercase. The screen display depends on the Editor mode.

Editor Lock

If you set this attribute, other users cannot change, rename, or delete your program. Only you will be able to change the program, its attributes, text elements, and documentation, or release the lock.

Fixed Point Arithmetic

If the attribute *Fixed point arithmetic* is set for a program, the system rounds type P fields according to the number of decimal places or pads them with zeros. The decimal sign in this case is always the period (.), regardless of the user's personal settings. We recommend that you **always** set the fixed point arithmetic attribute.

Start Using Variant

Only for Executable Programs (Reports)

Maintaining Program Attributes

If you set this attribute, other users can only start your program using a variant. You must then create at least one variant before the report can be started.

Editing Programs

programmeinleitende Anweisung [Page 1365]

You edit programs using the ABAP Editor [Ext.]. For detailed information, refer to the appropriate documentation. Below are a few hints to help you to get started:

Program Structure

The following gives a short overview on how to structure a program. Apart from the first statement, the sequence of statements is not obligatory, but you should keep to it for reasons of clarity and readability.

1. The first program statement

   The first statement of an ABAP program must always be the statement REPORT or PROGRAM, respectively (only exception: FUNCTION-POOL for function modules). Both statements have exactly the same function.

   The name specified in the statements REPORT and PROGRAM must not necessarily be the program name, but for documentation reasons, you should use the correct name.

   The statements REPORT or PROGRAM can have several options, such as LINE-SIZE, LINE-COUNT, or NO STANDARD PAGE HEADING. You use these options mainly in programs that which evaluate data and display the results in a list. For other options, such as the definition of a message class, see the key word documentation.

   Whenever you create a new program, the system automatically inserts the first ABAP statement, for example:

   REPORT <name>. for executable programs (reports) or

   PROGRAM <name>. for dialog programs

   As report or program name, the system enters the name you used to create the program.

2. Data declaration

   Next, insert all of your delclarations. This includes the global data definitions, selection screen definitions, and the definitions of classes and interfaces in ABAP Objects.

3. Processing logic

   After the declarations, write the processing logic. This consists of a series of processing blocks.

4. Subroutines

   At the end of your program, include its internal procedures (subroutines and methods).

Using includes to split up a program into a series of source code modules does not change this basic structure. If, for example, you follow the forward navigation of the ABAP Workbench when creating a dialog program, the system automatically creates a number of include programs, which contain the program parts described above in the correct sequence. The top include program usually contains the PROGRAM statement and the global data declaration. The subsequent include programs contain the individual dialog modules, ordered by PBO and PAI. There may also be further includes, for example, for subroutines. These include programs do not influence the program function, they only serve to make the program order easier to understand.

Program Layout

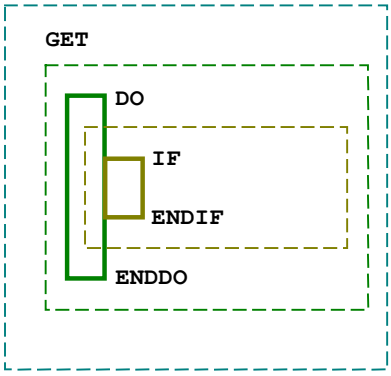
A high-quality ABAP program observes the following layout standards:

Comment Your Programs

Ensure that your program is correctly commented. For example, at the beginning of a subroutine, explain what it does, and provide any necessary information and references. The ABAP Editor [Ext.] provides predefined comment blocks. Comments within lines of code should be used sparingly, and only where they do not make the program flow more difficult to understand.

Indent Blocks of Statements

You should combine statements that belong together into a single block. Indent each block by at least two columns:



Use Modules

Make your programs easier to understand by using modules. For example, writing a large processing block as a subroutine makes the logical structure of your program easier to recognize. Subroutines may increase the overall length of programs, but you will soon find that this method greatly increases clarity, especially in the case of complex programs.

Use the Pretty Printer

If you use the Pretty Printer in the ABAP Editor [Ext.], your programs will conform to the layout guidelines. To use the Pretty Printer, choose *Program* → *Pretty Printer* from the Editor screen.

Statement Patterns

In the ABAP Editor, you can use statement patterns to help you write your program. They provide the exact syntax of a statement and follow the ABAP layout guidelines. You can insert two kinds of predefined structures into your program code when using the ABAP Editor: Keyword structures and comment lines. In the ABAP Editor, choose *Edit* → *Insert statement*. To display a list of all predefined keyword structures, place the cursor in the *Other pattern.* field and click the possible entries button to the right of the input field.

Checking Programs

When you have finished editing, or reach an intermediate stage of the program, choose *Check* to check the syntax. The system checks the program coding for syntax errors and compatibility

problems. If it finds an error, it displays a message describing it and, if possible, offers a solution or correction. The system positions the cursor on the error in the coding. Once you decide that your program is finished, run the extended program check on it by choosing *Program* → *Check* → *Ext. program check* from the Editor screen. Ensure that you correct all of the errors and warnings displayed. Although they do not prevent the program from working, they are not examples of good programming. Furthermore, some warnings might be escalated to syntax errors in future releases.

Saving and Activating Programs

Choose *Save* to save the source code.

The system stores the source code in the program library. Before you can execute the program from outside the ABAP Editor, you must generate an active version using the **Activate** function.

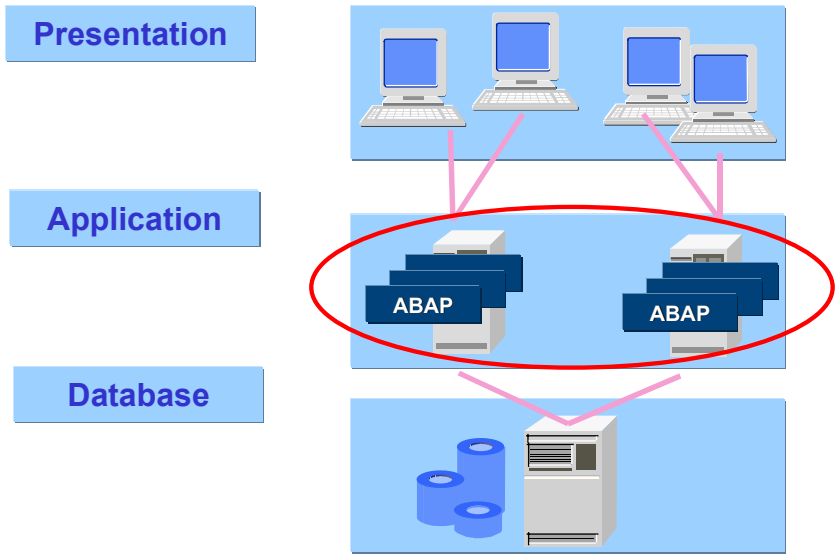
Testing Programs

You can test executable programs in the ABAP Editor. To do so, choose *Program* → *Execute*. The system then creates a temporary runtime object with a name that differs from the program name. However, the system executes the program as if started outside the ABAP Editor. If you created an ABAP module pool, you cannot test the program in the ABAP Editor. You must create a transaction code and a screen flow logic before you can execute the program.

Testing a program often involves a runtime analysis [Ext.], which shows you the amount of time your program consumes in the client/server environment of the R/3 system and what this time is used for. For more information, refer to the runtime analysis documentation.

The ABAP Programming Language

Typen und Objekte [Page 87]



ABAP Syntax [Page 83]
Basic Statements [Page 90]
Processing Large Volumes of Data [Page 250]
Saving Data Externally [Page 361]
Modularization Techniques [Page 441]
Special Techniques [Page 496]

ABAP Syntax

The syntax of the ABAP programming language consists of the following elements:

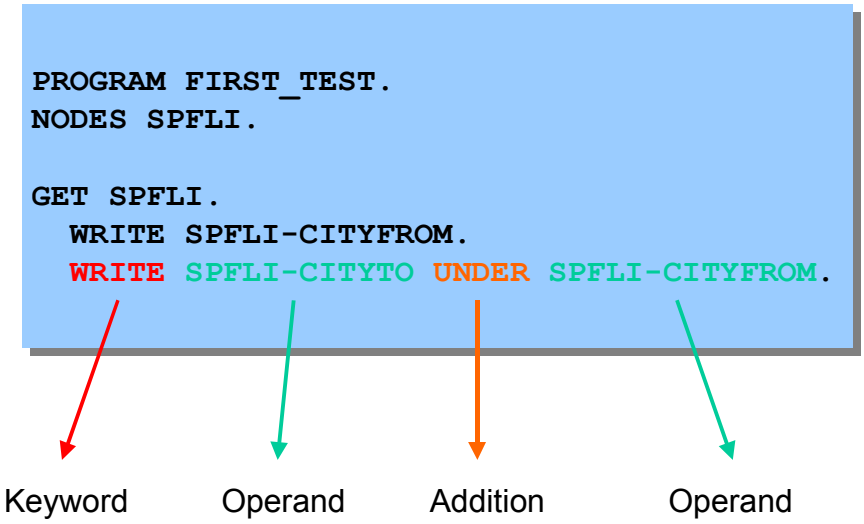
Statements

An ABAP program consists of individual ABAP statements. Each statement begins with a keyword and ends with a period.

```
PROGRAM FIRST_PROGRAM.

WRITE 'My First Program'.
```

This example contains two statements, one on each line. The keywords are PROGRAM and WRITE. The program displays a list on the screen. In this case, the list consists of the line "My First Program".

The keyword determines the category of the statement. For an overview of the different categories, refer to ABAP Statements [Page 56].



This diagram shows the structure of an ABAP statement.

Formatting ABAP Statements

ABAP has no format restrictions. You can enter statements in any format, so a statement can be indented, you can write several statements on one line, or spread a single statement over several lines.

You must separate words within a statement with at least one space. The system also interprets the end of line marker as a space.

The program fragment

```
PROGRAM TEST.
WRITE 'This is a statement'.
```

could also be written as follows:

```
PROGRAM TEST. WRITE 'This is a statement'.
```

or as follows:

```
PROGRAM
TEST.
      WRITE
     'This is a statement'.
```

Use this free formatting to make your programs easier to understand.

Special Case: Text Literals

Text literals [Page 119] are sequences of alphanumeric characters in the program code that are enclosed in quotation marks. If a text literal in an ABAP statement extends across more than one line, the following difficulties can occur:

- All spaces between the quotation marks are interpreted as belonging to the text literal.
- Letters in text literals in a line that is not concluded with quotation marks are interpreted by the editor as uppercase.

If you want to enter text literals that do not fit into a single line, you can use the '&' character to combine a succession of text literals into a single one.

The program fragment

```
PROGRAM TEST.
WRITE 'This
    is
    a statement'.
```

inserts all spaces between the quotation marks into the literal, and converts the letters to uppercase.

This program fragment

```
PROGRAM TEST.
WRITE 'This'  &
    ' is ' &
    'a statement'.
```

combines three text literals into one.

Chained Statements

The ABAP programming language allows you to concatenate consecutive statements with an identical first part into a chain statement.

To concatenate a sequence of separate statements, write the identical part only once and place a colon (:) after it. After the colon, write the remaining parts of the individual statements, separating them with commas. Ensure that you place a period (.) after the last part to inform the system where the chain ends.

Statement sequence:

```
WRITE SPFLI-CITYFROM.
WRITE SPFLI-CITYTO.
WRITE SPFLI-AIRPTO.
```

Chain statement:

```
WRITE: SPFLI-CITYFROM, SPFLI-CITYTO, SPFLI-AIRPTO.
```

In the chain, a colon separates the beginning of the statement from the variable parts. After the colon or commas, you can insert any number of spaces.

You could, for example, write the same statement like this:

```
WRITE:    SPFLI-CITYFROM,
          SPFLI-CITYTO,
          SPFLI-AIRPTO.
```

In a chain statement, the first part (before the colon) is not limited to the keyword of the statements.

Statement sequence:

```
SUM = SUM + 1.
SUM = SUM + 2.
SUM = SUM + 3.
SUM = SUM + 4.
```

Chain statement:

```
SUM = SUM + : 1, 2, 3, 4.
```

Comments

Comments are texts that you can write between the statements of your ABAP program to explain their purpose to a reader. Comments are distinguished by the preceding signs * (at the beginning of a line) and “ (at any position in a line). If you want the entire line to be a comment, enter an asterisk (*) at the beginning of the line. The system then ignores the entire line when it generates the program. If you want part of a line to be a comment, enter a double quotation mark (") before the comment. The system interprets comments indicated by double quotation marks as spaces.

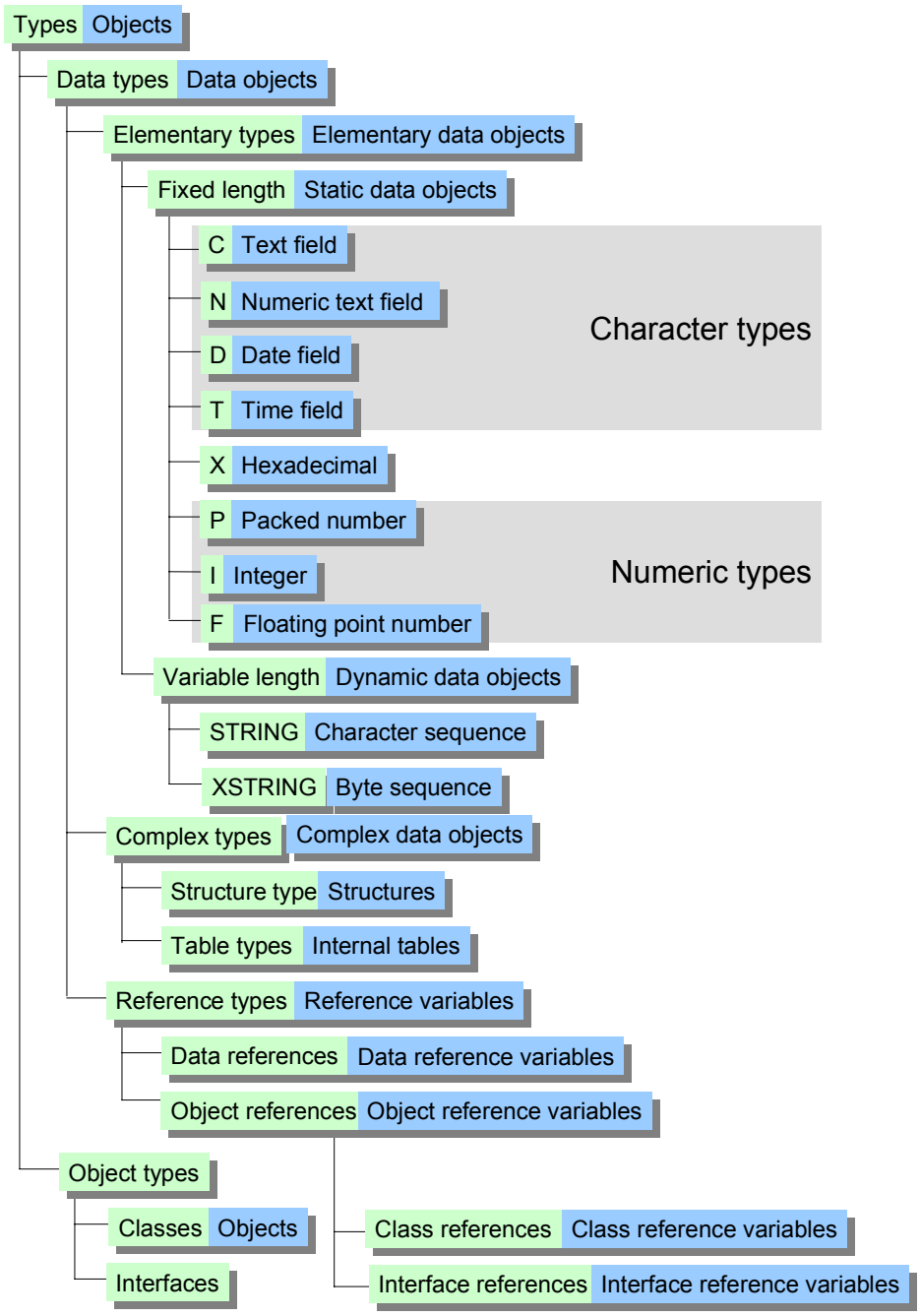
```
***********************************************
*  PROGRAM SAPMTEST                           *
*  WRITTEN BY KARL BYTE, 06/27/1995           *
*  LAST CHANGED BY RITA DIGIT, 10/01/1995     *
*  TASK:    DEMONSTRATION                     *
***********************************************

PROGRAM SAPMTEST.
```

```
**********************************************
* DECLARATIONS                               *
**********************************************

DATA: FLAG             " GLOBAL FLAG
      NUMBER TYPE I    " COUNTER

......
**********************************************
* PROCESSING BLOCKS                          *
**********************************************

......
```

Types and Objects

ABAP distinguishes between types and objects. Types are descriptions that do not occupy memory. Objects are instances of types, and do occupy their own memory space. A type describes the technical attributes of all of the objects with that type.

ABAP types form a hierarchy. Objects in ABAP reflect the same hierarchy.



ABAP has both data types and object types.

- Data types [Page 92] describe data objects [Page 118]. They can be further subdivided into elementary, reference, and complex types. There are predefined data types, but you can also declare your own, either locally in a program, or globally in the R/3 Repository.

- Object types describe objects in ABAP Objects [Page 1291]. They can be divided into classes and interfaces. Object types contain not only the data types specified above, but functions as well. There are no predefined object types. Instead, you must declare them in a program or in the R/3 Repository. A class is a full description of an object. It defines the data types and functions that an object contains. Interfaces describe an aspect of an object. The data types and functions of an interface can be implemented by several classes. Objects (instances) can only be created from classes. Object references, on the other hand, can be created with reference to either classes or interfaces.

There are two types of objects that you can create from ABAP types - data objects, and objects.

- Data objects [Page 118] are fields. They contain the data with which programs work at runtime.
- Objects [Page 1307] are real software objects in ABAP Objects [Page 1291]. They contain methods and events as well as data, and support object-oriented programming.

Basic Statements

Feldsymbole und Datenreferenzen [Page 200]

Data Types and Objects [Page 91]
Processing Data [Page 143]
Field Symbols
Logical Expressions [Page 225]
Controlling the Program Flow [Page 240]

Data Types and Data Objects

Programs work with local data. Data consists of strings of bytes in the memory area of the program. A string of related bytes is called a field. Each field has an identity (a name) and a data type. All programming languages have a concept that describes how the contents of a field are interpreted according to the data type.

In the ABAP type concept, fields are called data objects. Each data object is an instance of an abstract data type. Data types in ABAP are not just attributes of fields, but can be defined in their own right. There are separate name spaces for data objects and data types. This means that a name can at the same time be the name of a data object as well as the name of a data type.

Data Types

As well as occurring as attributes of a data object, data types can also be defined independently. The definition of a user-defined data type is based on a set of predefined elementary data types. You can define data types either locally in the declaration part of a program (using the TYPES statement) or globally in the ABAP Dictionary. You can use your own data types to declare data objects or to check the types of parameters in generic operations.

Data objects

Data objects are the physical units with which ABAP statements work at runtime. Each ABAP data object has a set of technical attributes, which are fully defined at all times when an ABAP program is running. The technical attributes of a data object are its length, number of decimal places, and data type. ABAP statements work with the contents of data objects and interpret them according to their data type. You declare data objects either statically in the declaration part of an ABAP program (the most important statement for this is DATA), or dynamically at runtime (for example, when you call procedures). As well as fields in the memory area of the program, the program also treats literals like data objects.

Data Types [Page 92]

Data Objects [Page 118]

Further information about data types and data objects:

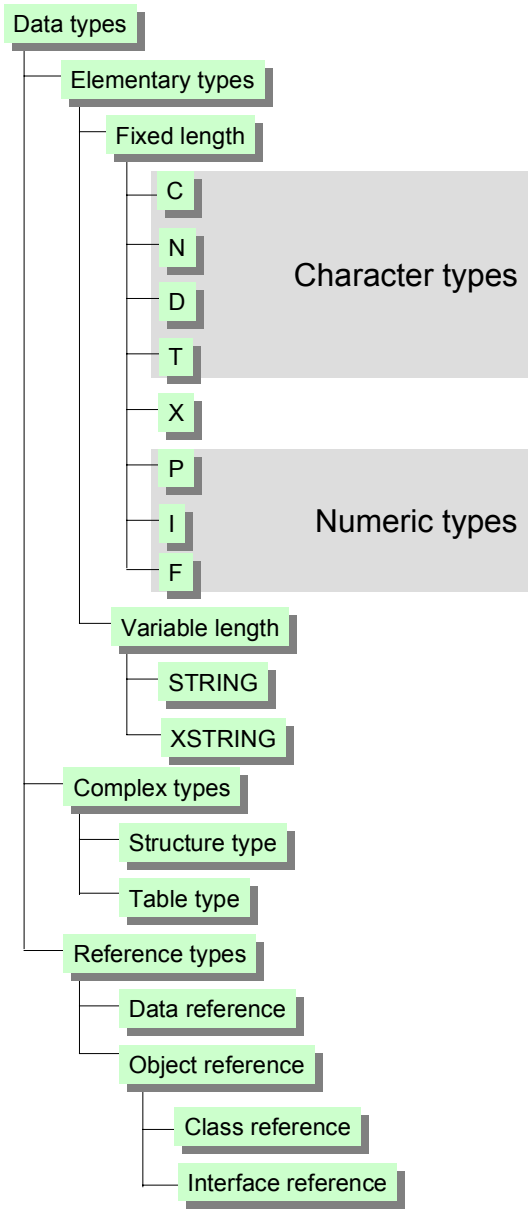
Compatibility [Page 133]

Determining the Attributes of Data Objects [Page 135]

Examples of Data Types and Objects [Page 140]

Data Types

The following graphic shows the different data types that are used in ABAP. Data types form a part of the ABAP Type Hierarchy [Page 87].



Data types can be divided into elementary, reference, and complex types.

Elementary Types

Elementary types are the smallest indivisible unit of types. They can be grouped as those with fixed length and those with variable length.

Fixed-Length Elementary Types

There are eight predefined types in ABAP with fixed length:

- Four character types:

  Character (C), Numeric character (N), Date (D), and Time (T).

- One hexadecimal type:

  Byte field (X).

- Three numeric types:

  Integer (I), Floating-point number (F) and Packed number (P).

Variable-Length Elementary Types

There are two predefined types in ABAP with variable length:

- STRING for character strings
- XSTRING for byte strings

Reference Types

Reference types describe data objects that contain references (pointers) to other objects (data objects and objects in ABAP Objects). There is a hierarchy of reference types that describes the hierarchy of objects to which the references can point. There are no predefined references - you must define them yourself in a program.

Complex Types

Complex types are made up of other types. They allow you to manage and process semantically-related data under a single name. You can access a complex data object either as a whole or by individual component. There are no predefined complex data types in ABAP. You must define them either in your ABAP programs or in the ABAP Dictionary. Structured types are divided further into structures and internal tables.

Structures

A structure is a sequence of any elementary types, reference types, or complex data types.

You use structures in ABAP programs to group work areas that logically belong together. Since the elements of a structure can have any data type, structures can have a large range of uses. For example, you can use a structure with elementary data types to display lines from a database table within a program. You can also use structures containing aggregated elements to include all of the attributes of a screen or control in a single data object.

The following terms are important when we talk about structures:

- Nested and non-nested structures
- Flat and deep structures

A nested structure is a structure that contains one or more other structures as components. Flat structures contain only elementary data types with a fixed length (no internal tables, reference types, or strings). The term deep structure can apply regardless of whether the structure is

nested or not. Nested structures are flat so long as none of the above types is contained in any nesting level.

Any structure that contains at least one internal table, reference type, or string as a component (regardless of nesting) is a deep structure. Accordingly, internal tables, references, and strings are also known as deep data types. The technical difference between deep structures and all others is as follows. When you create a deep structure, the system creates a pointer in memory that points to the real field contents or other administrative information. When you create a flat data type, the actual field contents are stored with the type in memory. Since the field contents are not stored with the field descriptions in the case of deep structures, assignments, offset and length specifications and other operations are handled differently from flat structures.

Internal Tables

Internal tables consists of a series of lines that all have the same data type. Internal tables are characterized by:

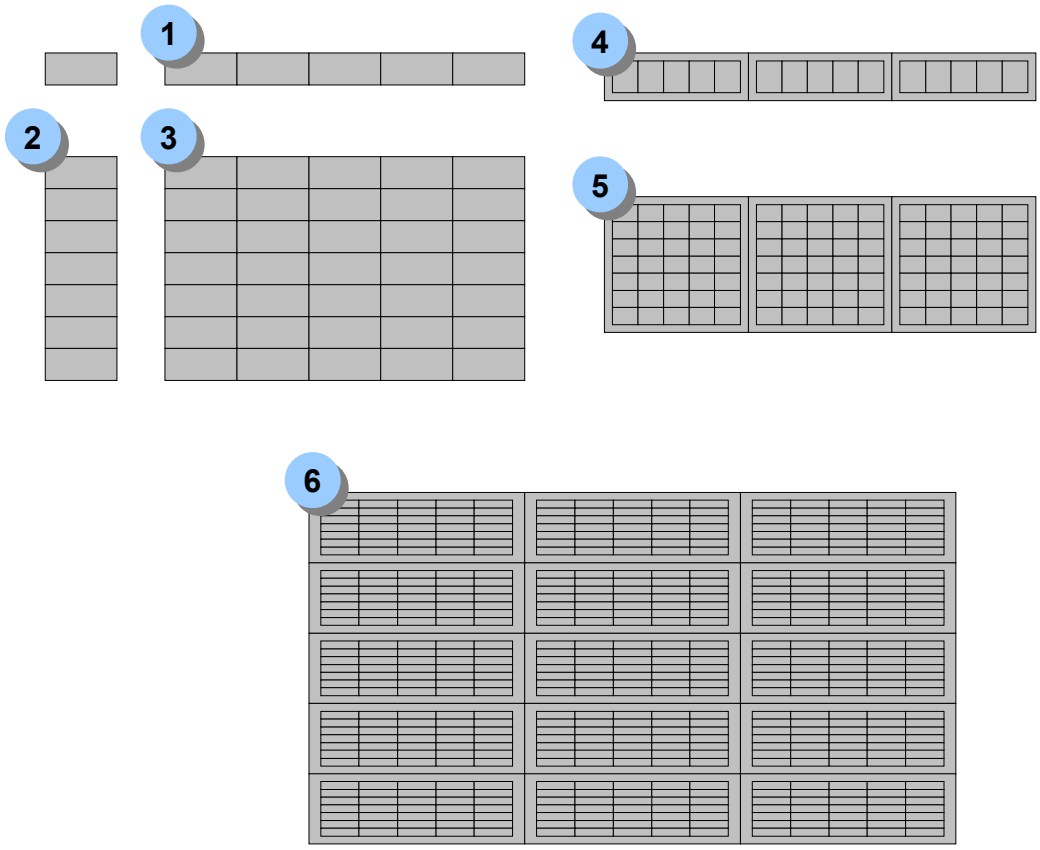
- The line type, which can be any elementary type, reference type, or complex data type.
- The key identifies table rows. It is made up of the elementary fields in the line. The key can be unique or non-unique.
- The access method determines how ABAP will access individual table entries. There are three access types, namely unsorted tables, sorted index tables and hash tables. For index tables, the system maintains a linear index, so you can access the table either by specifying the index or the key.
  Hashed tables have no linear index. You can only access hashed tables by specifying the key. The system has its own hash algorithm for managing the table.

You should use internal tables whenever you need to use structured data within a program. One imprint use is to store data from the database within a program.

Examples for Complex Data Types

The following list contains examples of complex data types in ascending order of complexity:

1. Structures consisting of a series of elementary data types of fixed length (non-nested, flat structures)
2. An internal table whose line type is an elementary type (vector).
3. Internal tables whose line type is a non-nested structure ('real' table)
4. Structures with structures as components (nested structures, flat or deep)
5. structures containing internal tables as components (deep structures)
6. Internal tables whose line type contains further internal tables.



The graphic shows how elementary fields can be combined to form complex types.

Further Information About Data Types

You can define data types at various levels in the R/3 System. For more information, refer to

Defining Data Types [Page 96].

Some ABAP statements allow you to use the TYPE addition to refer to an existing data type. The data types must be visible in the program for this to work. For more information, refer to

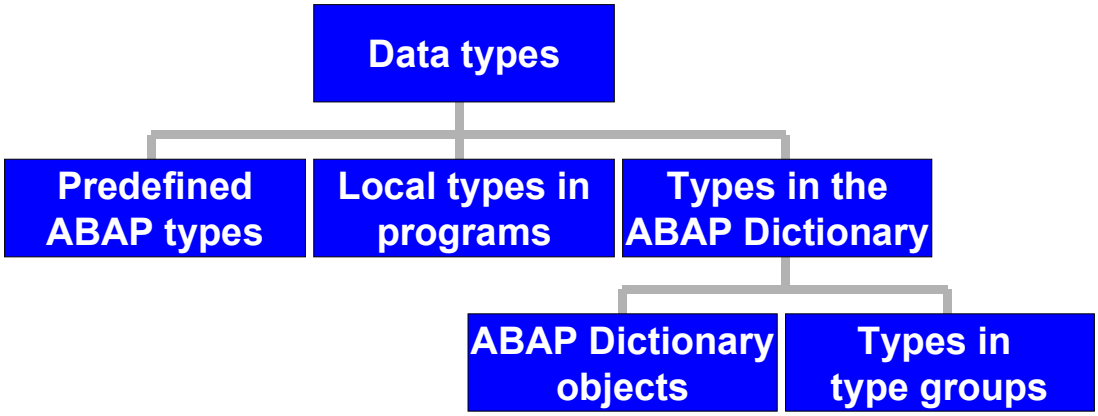
Visibility of Data Types [Page 112].

When working with data, it is important to know whether data types are compatible or not. For more information, refer to

Compatibility of Data Types [Page 133].

Defining Data Types

The following graphic shows where you can define data types in ABAP:



This differentiates between

- Predefined ABAP types [Page 97] that are built into the kernel.
- Local data types [Page 100] that you can define in ABAP programs.
- Data types in the ABAP Dictionary [Page 105] that are available to all programs in the entire system. In the ABAP Dictionary, you can define types either as Dictionary objects or in type groups.

Predefined ABAP Types

These data types are **predefined in the R/3 System kernel**, and are visible in all ABAP programs. You can use predefined types to define local data types and objects in a program and to specify the type of interface parameters and field symbols.

Predefined Elementary ABAP Types with Fixed Length

These predefined elementary data types are used to specify the types of individual fields whose lengths are always fixed at runtime. The following table shows the different fixed-length data types. All field lengths are specified in bytes.

Data Type	Initial field length	Valid field length	Initial value	Meaning
Numeric types				
I	4	4	0	Integer (whole number)
F	8	8	0	Floating point number
P	8	1 - 16	0	Packed number
Character types				
C	1	1 - 65535	' … '	Text field (alphanumeric characters)
D	8	8	'00000000'	Date field (Format: YYYYMMDD)
N	1	1 - 65535	'0 … 0'	Numeric text field (numeric characters)
T	6	6	'000000'	Time field (format: HHMMSS)
Hexadecimal type				
X	1	1 - 65535	X'0 … 0'	Hexadecimal field

Data types D, F, I, and T describe the technical attributes of a data object fully. Data types C, N, P, and X are generic. When you use a generic type to define a local data type in a program or a data objet, you must specify the field length and, in the case of type P, the number of decimal places. When you user generic types to specify the types of interface parameters of field symbols, you do not have to specify the technical attributes.

The initial value (and initial field length in the case of the generic types), are values that are used implicitly in short forms of the TYPES and DATA statements.

The fixed-length predefined types are divided into:

Numeric Types

As well as the five non-numeric types (text field (C), numeric text field (N), date field (D), time field (T), and hexadecimal field (X)), there are three numeric types, used in ABAP to display and

calculate numbers. Data type N is **not** a numeric type. Type N objects can only contain **numeric characters** (0...9), but are not represented internally as numbers. Typical type N fields are account numbers and zip codes.

- integers - type I

  The value range of type I numbers is -2**31 to 2**31-1 and includes only **whole** numbers. Non-integer results of arithmetic operations (e.g. fractions) are rounded, not truncated.

  You can use type I data for counters, numbers of items, indexes, time periods, and so on.

- Packed numbers - type P

  Type P data allows digits after the decimal point. The number of decimal places is generic, and is determined in the program. The value range of type P data depends on its size and the number of digits after the decimal point. The valid size can be any value from 1 to 16 bytes. Two decimal digits are **packed** into one byte, while the last byte contains one digit and the sign. Up to 14 digits are allowed after the decimal point. The initial value is zero. When working with type P data, it is a good idea to set the program attribute *Fixed point arithmetic*.Otherwise, type P numbers are treated as integers.

  You can use type P data for such values as distances, weights, amounts of money, and so on.

- Floating point numbers - type F

  The value range of type F numbers is 1x10**-307 to 1x10**308 for positive and negative numbers, including 0 (zero). The accuracy range is approximately 15 decimals, depending on the **floating point** arithmetic of the hardware platform. Since type F data is internally converted to a binary system, rounding errors can occur. Although the ABAP processor tries to minimize these effects, you should not use type F data if high accuracy is required. Instead, use type P data.

  You use type F fields when you need to cope with very large value ranges and rounding errors are not critical.

Using I and F fields for calculations is quicker than using P fields. Arithmetic operations using I and F fields are very similar to the actual machine code operations, while P fields require more support from the software. Nevertheless, you have to use type P data to meet accuracy or value range requirements.

Character types

Of the five **non-numeric** types, the four types C, D, N, and T are **character types**. Fields with these types are known as **character fields**. Each **position** in one of these fields takes up enough space for the code of one character. Currently, ABAP only works with single-byte codes such as ASCII and EBCDI. However, an adaptation to UNICODE is in preparation. Under UNICODE, each character occupies two or four bytes.

Hexadecimal Type

The remaining **non-numeric** type - X - **always** interprets individual bytes in memory. **One** byte is represented by a **two-digit** hexadecimal display. The fields with this type are called **hexadecimal fields**. In hexadecimal fields, you can process single bits [Page 178].

Predefined Elementary ABAP Types with Variable Length

These predefined elementary data types are used to specify the types of individual fields whose lengths are not fixed until runtime. There are two predefined ABAP data types with variable length that are generically known as strings:

- STRING for character strings

  A string is a sequence of characters with variable length. A string can contain any number of alphanumeric characters. The length of a string is the number of characters multiplied by the length required for the internal representation of a single character.

- XSTRING for byte strings

  A byte string is a hexadecimal type with variable length. It can contain any number of bytes. The length of a byte string is the same as the number of bytes.

When you create a string as a data object, only a string header is created statically. This contains administrative information. The actual data objects are created and modified dynamically at runtime by operational statements.

The initial value of a string is the empty string with length 0. A structure that contains a string is handled like a deep structure. This means that there are no conversion rules [Page 192] for structures that contain strings.

Predefined Complex Data Types

ABAP contains no predefined complex data types that you can use to define local data types or data objects in a program. All complex data types are based on elementary ABAP types, and are constructed in ABAP programs or in the ABAP Dictionary.

Local Data Types in Programs

All ABAP programs can define their own data types. Within a program, procedures [Page 449] can also define local types.

You define local data types in a program using the

```
TYPES <t> ... [TYPE <type>|LIKE <obj>] ...
```

statement. The type name <t> may be up to 30 characters long. You can use any letters, digits, and the underscore character. Do not create a name consisting entirely of numeric characters. You cannot use the special characters + . , : ( ) - < >. Other special characters are reserved for internal use. You cannot use the names of the predefined ABAP types (C, D, F, I, N, P, T, X, STRING, XSTRING) or the name of the generic type TABLE. You should not use names that are the same as an ABAP keyword or addition. You should:

- Use names that explain the meaning of the type without the need for further comments
- Use the underscore character to separate compound words
- Always use a letter as the first character of a variable name.

You declare local data types in a program either by referring to an existing data type or constructing a new type.

An existing type can be

- A predefined ABAP type to which you refer using the TYPE addition
- An existing local type in the program to which you refer using the TYPE addition
- The data type of a local object in the program to which you refer using the LIKE addition
- A data type in the ABAP Dictionary to which you refer using the TYPE addition. To ensure compatibility with earlier releases, it is still possible to use the LIKE addition to refer to database tables and **flat** structures in the ABAP Dictionary. However, you should use the TYPE addition in new programs.

Known types must be visible [Page 112] at the point where you define the new type. If the existing type is generic, you can use further additions to set the attributes of type <t> that are still undefined.

Elementary Data Types

Local elementary data types in a program are defined with reference to predefined elementary types. You use the following syntax:

```
TYPES <t>[(<length>)] [TYPE <type>|LIKE <obj>] [DECIMALS <dec>].
```

<type> is one of the predefined ABAP types C, D, F, I, N, P, T, X, STRING, or XSTRING, an existing elementary local type in the program, or a data element defined in the ABAP Dictionary. When you refer to a data element from the ABAP Dictionary, the system converts it into an elementary ABAP type. If you use a LIKE reference, <obj> can be an existing data object with an elementary data type.

If you do not use the TYPE or LIKE addition, the system uses the default predefined type C. If <type> is one of the generic elementary predefined types with fixed length (C, N, P, X), you should set a length using the <length> option. If you omit it, the field length is set to the relevant initial value in the table in the Predefined ABAP Types [Page 97] section. If <type> is P, you can

specify the number of decimal places using the DECIMALS addition. If you omit this, the number of decimal places is set to 0.

Thus the implicit statement

```
TYPES <t>.
```

defines a character variable <f> with length 1. It is a shortened form of the explicit statement

```
TYPES <t>(1) TYPE C.
```

However, you should always use the explicit statement. The short form is prohibited within ABAP Objects classes.

Elementary local data types in a program make your programs easier to read and understand. If you have used such a type to define several data objects, you can change the type of all of those objects in one place, just be changing the definition in the TYPES statement. Alternatively, if you use a set of data types regularly in different programs but do not want to store them in the ABAP Dictionary, you can create an include program [Page 447] for the type definitions, and incorporate this into the relevant programs.

```
TYPES: number TYPE i,
       length TYPE p DECIMALS 2,
       code(3) TYPE c.
...
```

In this example, a data type called NUMBER is defined. It is the same as the predefined data type I, except it has a different name to make the program easier to read.
The program defines a data types LENGTH, based on the generic ABAP type P. LENGTH is defined with a given number of decimals. If it becomes necessary to change the accuracy of length specifications, for example, you only have to change the TYPES statement in the program.
A third data type, CODE, is also defined. CODE is based on the predefined generic ABAP type C. The length is defined as 3 bytes.

```
DATA counts TYPE i.
TYPES: company    TYPE spfli-carrid,
       no_flights LIKE counts.
```

This example shows how you can use the TYPE addition to refer to a column (CARRID) of a database table (SPFLI). The LIKE addition refers to an existing data object.

Reference Types

You can define reference types locally in your programs or globally in the ABAP Dictionary. You use the following syntax:

```
TYPES <t> TYPE REF TO ...
```

After TYPE, there is no reference to an existing data type. Instead, the type constructor occurs:

- The type constructor

  ```
  REF TO DATA
  ```

declares a reference <t> to a data object. Fields with type <t> can contain references (pointers) to data objects, that is, instances of data types (see also Data References [Page 219]).

- The type constructor

  ```
  REF TO <class>|<interface>
  ```

  defines a reference type <t> to the class <class> or interface <interface>. Fields with type <t> can contain references (pointers) to instances of the class <class> or of classes that implement the interface <interface> (see also Object Handling [Page 1307]).

Complex Types

Complex local data types in programs always consist of the above elementary data types or reference types. In this case, the TYPES statement is a construction blueprint for the complex data type.

When you refer to a known complex type using

```
TYPES <t> [TYPE <type>|LIKE <obj>].
```

the new complex type is constructed using the known type as a template. When you refer to the complex ABAP Dictionary types structure or table type, which are based on ABAP Dictionary data elements, the structure from the Dictionary is used, and the data elements are converted into elementary ABAP types.

The TYPES statement allows you to define new complex data types without referring to existing structures and tables.

Structure Types

To construct a new structure type in a program, use the following **chained** TYPES statement:

```
TYPES: BEGIN OF <structure>,
                ..............
                <ti> ...,
                ..............
       END OF <structure>.
```

This chained statement creates a structure containing all of the variables between

`TYPES BEGIN OF <structure>.` and `TYPES END OF <structure>.`

that occur in the data types defined in

`TYPES <ti>... .`

The components <fi> can be elementary types, reference types, or, if you refer to known complex types, complex themselves. TYPES BEGIN OF... TYPES END OF blocks can also be nested, so you can create deep structure types.

The individual variables within a structure type are addressed in the program with a hyphen between the structure name and component name as follows: <structure>-<fi>.

```
TYPES: spfli_type TYPE spfli,
       surname(20) TYPE c,
       BEGIN OF address,
             name       TYPE surname,
```

```
            street(30) TYPE c,
            city       TYPE spfli_type-cityfrom,
          END OF address,
          town TYPE address-city.
```

This example shows the definition of two structure types in a program - SPFLI_TYPE and ADDRESS. The structure of the data type SPFLI_TYPE is taken from the database table SPFLI in the ABAP Dictionary. The components of SPFLI_TYPE are the same as the columns of SPFLI. The individual data types of the components are the ABAP equivalents of the data types of the columns of the database table. The structure type ADDRESS is newly defined. The component ADDRESS-NAME takes the data type of the previously-defined type SURNAME, the component ADDRESS-STREET is newly-defined, ADDRESS-CITY takes the data type of column CITYFROM of the structure type SPFLI_TYPE.

```
TYPES: BEGIN OF struct1,
         col1 TYPE i,
         BEGIN OF struct2,
           col1 TYPE i,
           col2 TYPE i,
         END OF struct2,
       END OF struct1.
TYPES mytype TYPE struct1-struct2-col2.
```

The example shows how you can construct a nested structure type STRUCT1 with a complex component STRUCT2 by nesting TYPES BEGIN OF ... TYPES END OF blocks, and how you can address the inner components.

```
* local types in program
* referring to predefined ABAP types:
TYPES: surname(20)  TYPE c,
       street(30)   TYPE c,
       zip_code(10) TYPE n,
       city(30)     TYPE c,
       phone(20)    TYPE n,
       date         LIKE sy-datum.
* local structure in program
* referring to the above types
TYPES: BEGIN of address,
         name TYPE surname,
         code TYPE zip_code,
         town TYPE city,
         str  TYPE street,
       END OF address.
* local nested structure in program
* referring to the above types
TYPES: BEGIN of phone_list,
         adr TYPE address,
         tel TYPE phone,
       END OF phone_list.
```

This example shows how to create complex data types from simple type definitions. After a set of simple data types are created with ABAP predefined

types, a structured type ADDRESS is defined using the data types defined earlier. Finally, a nested structure type, PHONE_LIST, is created, whose first component has the type ADDRESS.

Table types

Local tables in a program are called internal tables. To construct a new internal table type, use the syntax:

```
TYPES <t> TYPE|LIKE <tabkind> OF <linetype> [WITH <key>].
```

After TYPE, there is no reference to an existing data type. Instead, the type constructor occurs: The type constructor

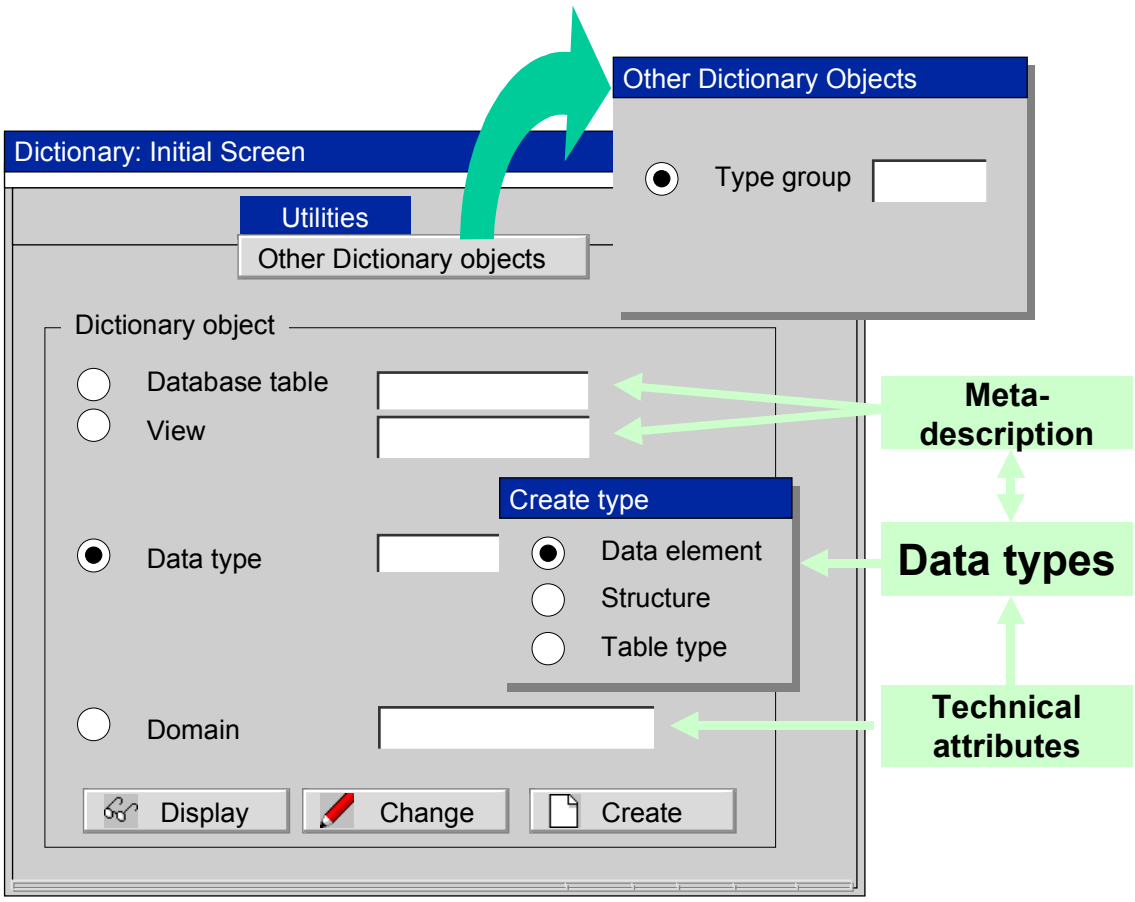
```
<tabkind> OF <linetype> [WITH <key>]
```

defines an internal table type with access type <tabkind>, line type <linetype> and key <key>. The line type <linetype> can be any known data type. Specifying the key is optional. Internal tables can thus be generic. For more information, refer to Internal Tables [Page 251].

Data Types in the ABAP Dictionary

The ABAP Dictionary allows you to define global data types. You can use the TYPE addition of an appropriate ABAP statement to refer to these data types in any ABAP program in the system.

You define these data types using the ABAP Dictionary [Ext.]. The following input fields are relevant to data types:



There are three groups on the initial screen:

Database Tables and Views

One of the most important tasks of the ABAP Dictionary [Ext.] is to administer database tables in the R/3 database. The Dictionary contains metadescriptions of the database tables, and uses these to create the physical tables in the database. A view is a "virtual table" containing fields from one or more tables.

In the description of a database table, the table lines consist of single fields or columns. An elementary data type must be assigned to each column. The elementary types in the ABAP Dictionary are data elements. Like data objects in ABAP programs, database tables and views have data types as attributes. A line of a database table or view has the data type of a **flat structure**, which consists of individual data elements.

Data Types in the ABAP Dictionary

In ABAP programs, you can use the TYPE addition with the data type of a database table or view. You may refer to the whole structure or to individual components:

```
... TYPE <dbtab> ...
```

refers to the complex data type of the structure,

```
... TYPE <dbtab>-<ci> ...
```

refers to the elementary data type of component <c_i>.

If you define a complex data type <t> as a structure using

```
TYPES <t> TYPE <dbtab>.
```

the components of the data type <t> inherit the names of the components of the database table or view, and can be addressed in the program using <t>-<c_i>.

To ensure compatibility with previous releases, you can still use the LIKE addition to refer to database tables or views, except within classes. The reason for this is that in earlier releases, the physical presence of the database tables as objects was emphasized, even though the Dictionary only contains metadescriptions and data types.

Defining program-local data types by referring to database tables and views is one of the essential techniques for processing data from database tables in ABAP. Data objects that you define in this way always have the right type to contain data from the corresponding database table. ABAP Open SQL allows you to read a single field, a range of fields, or an entire database table or view into an internal table.

```
TYPES: city type spfli-cityfrom,
       spfli_type TYPE STANDARD TABLE OF spfli WITH DEFAULT
KEY.

DATA: wa_city  TYPE city,
      wa_spfli TYPE spfli_type.

...

SELECT SINGLE cityfrom FROM spfli
                       INTO wa_city
                       WHERE carrid = 'LH' AND connid = '400'.

...

SELECT * FROM spfli INTO TABLE wa_spfli.

...
```

This example defines an elementary data type CITY that refers to a single field of the database table SPFLI and an internal table SPFLI_TYPE, whose line type is the same as the structure of the database table. The SELECT statement reads data from the database into the corresponding data objects.

Data types

Data types are the actual type definitions in the ABAP Dictionary. They allow you to define elementary types, reference types, and complex types that are visible globally in the system. The data types of database tables are a subset of all possible types, namely flat structures.

Global object types (classes and interfaces) are not stored in the ABAP Dictionary, but in the class library. You create them using the Class Builder.

For a detailed description of data types and their definitions, refer to the Types [Ext.] section of the ABAP Dictionary documentation. The following descriptions mention the types only briefly, along with how you can refer to them from ABAP programs.

Data Elements

Data elements in the ABAP Dictionary describe individual fields. They are the smallest indivisible units of the complex types described below, and are used to specify the types of columns in the database. Data elements can be elementary types or reference types.

- **Elementary Types**

  Elementary types are part of the dual-level domain concept for fields in the ABAP Dictionary. The elementary type has semantic attributes, such as texts, value tables, and documentation, and has a data type. There are two different ways to specify a data type:

  – By directly assigning an ABAP Dictionary type.

  You can assign a predefined ABAP Dictionary type and a number of characters to an elementary type. The ABAP Dictionary has considerably more predefined types than the ABAP programming language. The number of characters here is not the field length in bytes, but the number of valid characters excluding formatting characters. The data types are different because the predefined data types in the ABAP Dictionary have to be compatible with the external data types of the database tables supported by the R/3 System.

  When you refer to data types from the ABAP Dictionary in an ABAP program, the predefined Dictionary types are converted to ABAP types as follows:

Dictionary type	Meaning	Maximum length n	ABAP type
DEC	Calculation/amount field	1-31, 1-17 in tables	P((n+1)/2)
INT1	Single-byte integer	3	Internal only
INT2	Two-byte integer	5	Internal only
INT4	Four-byte integer	10	I
CURR	Currency field	1-17	P((n+1)/2)
CUKY	Currency key	5	C(5)
QUAN	Amount	1-17	P((n+1)/2)
UNIT	Unit	2-3	C(n)
PREC	Accuracy	2	X(2)
FLTP	Floating point number	16	F(8)
NUMC	Numeric text	1-255	N(n)
CHAR	Character	1-255	C(n)
LCHR	Long character	256-max	C(n)

STRING.	String of variable length	1-max	STRING.
RAWSTRING	Byte sequence of variable length	1-max	XSTRING
DATS	Date	8	D
ACCP	Accounting period YYYYMM	6	N(6)
TIMS	Time HHMMSS	6	T
RAW	Byte sequence	1-255	X(n)
LRAW	Long byte sequence	256-max	X(n)
CLNT	Client	3	C(3)
LANG	Language	internal 1, external 2	C(1)

("max" in LCHR and LRAW is the value of a preceding INT2 field. The "internal" length of a LANG field is in the Dictionary, the "external" length refers to the display on the screen.

- Assigning a domain

  The technical attributes are inherited from a domain. Domains are standalone Repository objects in the ABAP Dictionary. They can specify the technical attributes of a data element. One domain can be used by any number of data elements. When you create a domain, you must specify a Dictionary data type (see above table) and the number of characters.

- **Reference Types**

  Reference types describe single fields that can contain references to **global** classes and interfaces from the ABAP class library.

In an ABAP program, you can use the TYPE addition to refer directly to a data element. The predefined Dictionary data types of the domain are then converted into the corresponding ABAP types.

If you define a local data type in a program by referring to a data element as follows:

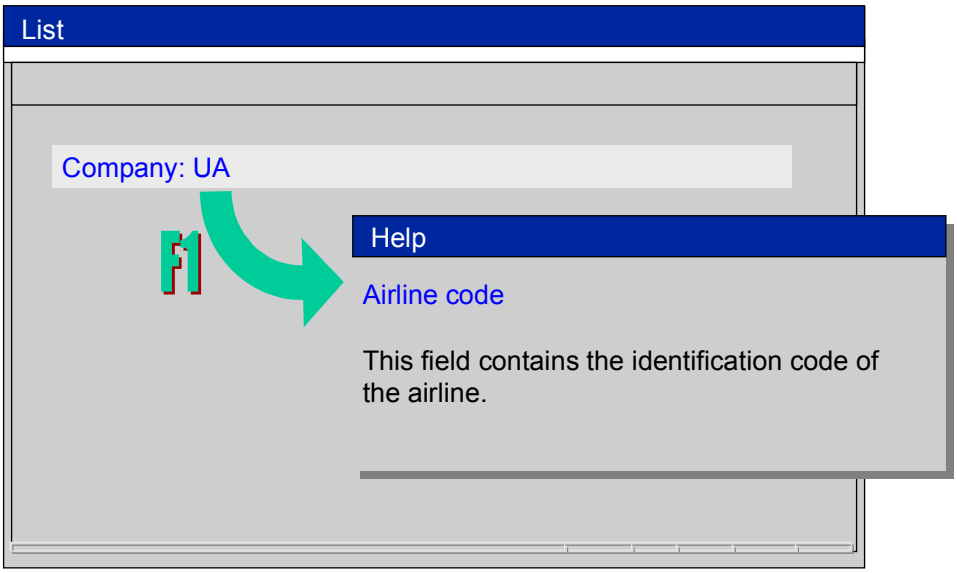
```
TYPES <t> TYPE <data element>.
```

the semantic attributes of the data element are inherited and will be used, for example, when you display a data object with type <t> on the screen. Since all data types in the ABAP Dictionary are based on data elements, they all contain the corresponding semantic attributes.

```
TYPES company TYPE s_carr_id.

DATA wa_company TYPE company.

wa_company = 'UA '.

WRITE: 'Company:', wa_company.
```

This example defines a local type COMPANY that refers to the data element S_CARR_ID. The data element is linked to the identically-named domain S_CARR_ID. The domain defines the technical attributes as data type CHAR with length 3. The local data type COMPANY in the program therefore has the ABAP type C(3). COMPANY also adopts the semantic attributes of the data element. In the

above example, we declare a data object WA_COMPANY with this type and display it on a list. If the user chooses F1 help for the output field, the help text stored in the ABAP Dictionary will appear in a dialog box.



Structures

A structure is a sequence of any other data types from the ABAP Dictionary, that is, data elements, structures, table types, or database tables. When you create a structure in the ABAP Dictionary, each component must have a name and a data type.

In an ABAP program, you can use the TYPE addition to refer directly to a structure.

If you define a local data type in a program by referring to a structure as follows:

```
TYPES <t> TYPE <structure>.
```

the construction blueprint of the structure is used to create a local structure <t> in the program. The predefined Dictionary data types of the domains used by the data elements in the structure are converted into the corresponding ABAP types. The semantic attributes of the data elements are used for the corresponding components of the structure in the program. The components of the local structure <t> have the same names as those of the structure in the ABAP Dictionary.

To ensure compatibility with previous releases, it is still possible to use the LIKE addition in an ABAP program to refer to a structure in the ABAP Dictionary (except in classes).

Suppose the structure STRUCT is defined as follows in the ABAP Dictionary:

Field name	Type name	Description
COL1	CHAR01	Character field with length 1
COL2	CHAR08	Character field with length 8
COL3	CHAR10	Character field with length 10

The types CHAR01 to CHAR10 are data elements with corresponding domains. We can refer to this structure in ABAP:

```
TYPES struct_type TYPE struct.

DATA wa TYPE struct_type.

wa-col1 = '1'.
wa-col2 = '12345678'.
wa-col3 = '1234567890'.
```

This program creates a local structure in the program - STRUCT_TYPE - and a corresponding data object WA. We can address the components using the component names from the original structure.

Table Types

Table types are construction blueprints for internal tables that are stored in the ABAP Dictionary. When you create a table type in the ABAP Dictionary, you specify the line type, access type, and key. The line type can be any data type from the ABAP Dictionary, that is, a data element, a structure, a table type, or the type of a database table. You can also enter a predefined Dictionary type directly as the line type, in the same way that you can with a domain.

In an ABAP program, you can use the TYPE addition to refer directly to a table type.

If you define a local data type in a program by referring to a table type as follows:

```
TYPES <t> TYPE <table>.
```

the construction blueprint of the table type is used to create a local internal table <t> in the program. The predefined Dictionary data types of the domains used by the data elements in the structure are converted into the corresponding ABAP types. The semantic attributes of the data elements are used for the corresponding components of the internal table in the program.

Suppose the table type STRUCT_TABLE is defined in the Dictionary with the line type STRUCT from the previous example. We can refer to this in ABAP:

```
TYPES table_type TYPE struct_table.

DATA: table_wa TYPE table_type,
      line_wa  LIKE LINE OF table_wa.

...

LOOP AT table_wa INTO line_wa.
  ...
  WRITE: line_wa-col1, line_wa-col1, line_wa-col1.
  ...
ENDLOOP.
```

This program defines an internal table type TABLE_TYPE. From it, we define data objects TABLE_WA and LINE_WA. LINE_WA corresponds to the line type of the table type in the Dictionary, and is therefore compatible with the structure STRUCT.

Type Groups

Before Release 4.5A, it was not possible to define standalone types in the ABAP Dictionary to which you could refer using a TYPE addition in an ABAP program. It was only possible to refer

to flat structures. Structures in programs corresponded to the structures of database tables or structures in the ABAP Dictionary. In ABAP programs, you could only refer to database tables and structures in the ABAP Dictionary using LIKE. It was, however, possible to refer to individual components of the Dictionary type. Complex local data types such as internal tables or deep structures had no equivalent in the ABAP Dictionary. The solution to this from Release 3.0 onwards was to use type groups. Type groups were based on the include technique, and allowed you to store any type definitions globally in the Dictionary by defining them using TYPES statements.

The definition of a type group is a fragment of ABAP code which you enter in the ABAP Editor. The first statement for the type group <pool> is always:

```
TYPE-POOL <pool>.
```

After this came the definitions of data types using the TYPES statement, as described in Local Data Types in Programs [Page 100]. It was also possible to define global constants using the CONSTANTS statement. All the names of these data types and constants must begin with the name of the type group and an underscore:

In an ABAP program, you must declare a type group as follows before you can use it:

```
TYPE-POOLS <pool>.
```

This statement allows you to use all the data types and constants defined in the type group <pool> in your program. You can use several type groups in the same program.

Let the type group HKTST be created as follows in the ABAP Dictionary:

```
TYPE-POOL hktst.

TYPES: BEGIN OF hktst_typ1,
                col1(10) TYPE c,
                col2 TYPE i,
       END OF hktst_typ1.

TYPES hktst_typ2 TYPE p DECIMALS 2.

CONSTANTS hktst_eleven TYPE i VALUE 11.
```

This type group defines two data types HKTST_TYP1 and HKTST_TYP2, as well as a constant HKTST_ELEVEN with the value 11.

Any ABAP program can use these definition by including a TYPE-POOLS statement:

```
TYPE-POOLS hktst.

DATA: dat1 TYPE hktst_typ1,
      dat2 TYPE hktst_typ2 VALUE '1.23'.

WRITE: dat2, / hktst_eleven.
```

The output is:

```
        1,23
          11
```

The data types defined in the type group are used to declare data objects with the DATA statement and the value of the constant is, as the output shows, known in the program.

The TYPE Addition

You use the TYPE addition in various ABAP statements for defining data types and specifying the types of interface parameters or field symbols. The TYPE addition can have various meanings depending on the syntax and context.

Referring to Known Data Types

You can use the addition

```
TYPE <type>
```

to refer to any data type <type> that is already **known** at this point in the program. It can be used in any of the statements listed below. The expression <obj> is either the name of the data object or the expression

```
LINE OF <table-type>
```

In this case, the TYPE addition describes the line type of a table type <table-type> that is **visible** at that point in the program.

ABAP Statements with TYPE References

- Definition of local program types using

  ```
  TYPES <t> TYPE <type>.
  ```

  The new data type <t> has the same type as <type>.

- Declaration of data objects using

  ```
  DATA <f> TYPE <type>.
  CLASS-DATA <f> TYPE <type>.
  CONSTANTS <f> TYPE <type>.
  STATICS <f> TYPE <type>.
  PARAMETERS <f> TYPE <type>.
  ```

  The data object <f> has a data type corresponding to the type <type>.

- Dynamic creation of data objects using

  ```
  CREATE DATA <dref> TYPE <type>.
  ```

- Specification of the type of a formal parameter in a subroutine using

  ```
  FORM <sub> ... USING|CHANGING <p> TYPE <type> ...
  ```

  The technical attributes of the formal parameter <p> are inherited from those of the declared data type <type>. You can then only pass actual parameters that have these attributes.

- Specification of the type of a formal parameter in a method using

  ```
  METHODS <meth> ... IMPORTING|EXPORTING|CHANGING <p> TYPE <type>
  ...
  ```

The technical attributes of the formal parameter <p> are inherited from those of the declared data type <type>. You can then only pass actual parameters that have these attributes.

- Specification of the type of a field symbol

```
FIELD-SYMBOLS <fs> TYPE <type>.
```

The technical attributes of the field symbol <FS> are inherited from those of the declared data type <type>. You can then only pass actual parameters that have these attributes.

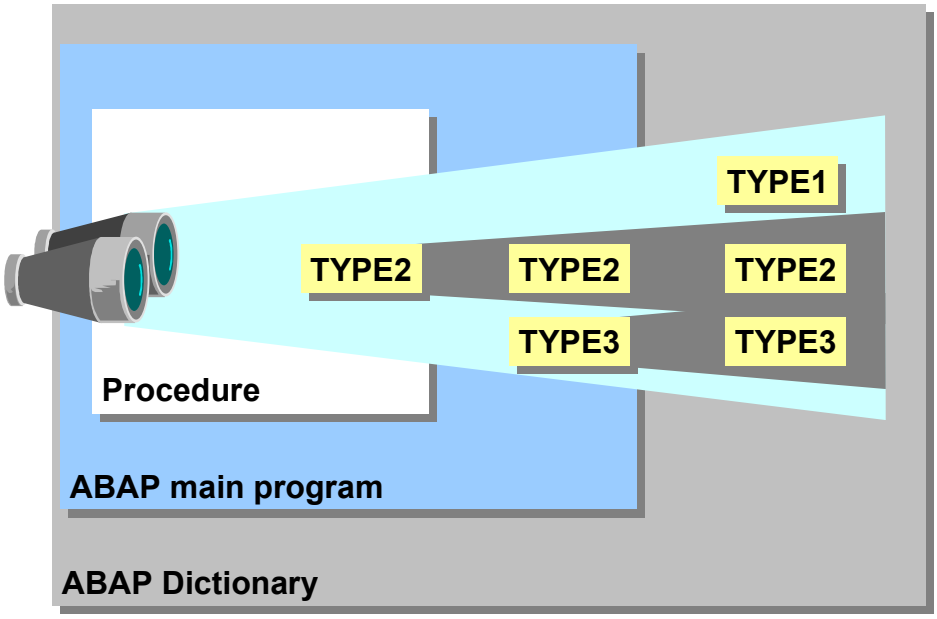
Visibility of Data Types

When you refer to known data types using the TYPE addition, the visibility of the data types is important.

- The predefined ABAP types (C, D, F, I, N, P, T, and X) are always visible. You cannot declare types with the same names as these data types, either in the program or in the ABAP Dictionary.
- When we talk about the visibility of local data types in the program, we must differentiate between local data types in procedures and global data types. Data types defined in a procedure obscure other objects with the same name that are declared in the global declarations of the program. All local data types in a program obscure data types with the same names in the ABAP Dictionary. This also applies to data types from type groups.
- In the ABAP Dictionary, different visibility rules apply to standalone data types and the data types stored in type groups. Data types in type groups obscure standalone data types with the same names. However, this should be an **exceptional situation**. All data types in the ABAP Dictionary should be in the same namespace. When you create a standalone data type, the system displays a warning if the name begins with the name of a type group followed by an underscore. Equally, you cannot create a type group if there is already a standalone data type with the same name followed by an underscore.

The graphic shows the visibility of local and ABAP Dictionary data types:

Visibility of Data Types



The system searches from the inside out. If you specify TYPE1 in a TYPE addition in the program, the system uses the ABAP Dictionary type both in the procedure and the main program. If you specify TYPE2 in the procedure, the system uses the local type from the procedure. However, if you specify TYP2 in the main program, the system uses the type from the main program. TYPE2 from the ABAP Dictionary is obscured. If you specify TYPE3 either in the procedure or the main program, the system uses the type from the main program TYPE3 from the ABAP Dictionary is obscured.

Constructing New Data Types

The TYPE addition allows you to construct new data types in the TYPES, DATA; CONSTANTS; and STATICS statements. In the TYPES statement, these are local data types in the program. In the other statements, they are attributes of new data objects.

You can use the following type constructors with the TYPE addition:

- For references

```
REF TO <class>|<interface>
```

- For structures

```
BEGIN OF <struct>.
  ...
END OF <struct>.
```

- For tables

```
<tabkind> OF <linetype> [WITH <key>]
```

These data types only exist during the runtime of the ABAP program.

Referring to Generic Types When Specifying a Type

There is a set of predefined generic types in ABAP syntax that you can use to specify the types of interface parameters and field symbols. You can only use them after the TYPE addition in the FORM, METHODS, and FIELD-SYMBOLS statements.

Generic Types for Type Specification

ANY	Fully generic type
ANY TABLE	Generic table for any internal table
INDEX TABLE	Generic type for tables with a linear index
TABLE STANDARD TABLE	Generic type for unsorted tables with a linear index
SORTED TABLE	Generic type for sorted tables with a linear index
HASHED TABLE	Generic table for tables with hash administration

You **cannot** use these types to define local data types in a program or data objects. They only allow you to check the data type of fields that you pass to procedures or of field symbols.

The LIKE Addition

You use the LIKE addition, similarly to the TYPE addition, in various ABAP statements for defining data types and specifying the types of interface parameters or field symbols. The addition

LIKE <obj>

can be used in the same ABAP statements as the TYPE addition [Page 112] to refer to any data object <obj> that is already **visible** at that point in the program. The expression <obj> is either the name of the data object or the expression

LINE OF <table-object>

In this case, the LIKE addition describes the line type of a table object that is **visible** at that point in the program.

You use LIKE to make the new object or type inherit the technical attributes of an existing data object.

ABAP Statements with LIKE References

- Definition of local types in a program using

  TYPES <t> LIKE <obj>.

  The new data type <t> inherits all of the technical attributes of the data object <obj>.

- Declaration of data objects using

  DATA <f> LIKE <obj>.

  CLASS-DATA <f> LIKE <obj>.

  CONSTANTS <f> LIKE <obj>.

  STATICS <f> LIKE <obj>.

  PARAMETERS <f> LIKE <obj>.

  The data object <f> inherits all of the technical attributes of the data object <obj>.

- Dynamic creation of data objects using

  CREATE DATA <dref> LIKE <obj>.

- Specification of the type of a formal parameter in a subroutine using

  FORM <sub> ... USING|CHANGING <p> LIKE <obj> ...

  The technical attributes of the formal parameter <p> are inherited from those of the declared data object <obj>. You can then only pass actual parameters that have these attributes.

- Specification of the type of a formal parameter in a method using

  METHODS <meth> ... IMPORTING|EXPORTING|CHANGING <p> LIKE <obj> ...

  The technical attributes of the formal parameter <p> are inherited from those of the declared data type <type>. You can then only pass actual parameters that have these attributes.

- Specification of the type of a field symbol

FIELD-SYMBOLS <fs> LIKE <obj>.

The technical attributes of the field symbol <FS> are inherited from those of the declared data object <obj>. You can then only assign data objects that have these attributes.

Visibility of Data Objects

As a rule, you can use LIKE to refer to any object that has been declared using DATA or a similar statement, and is visible in the current context. The object only has to have been declared. It is irrelevant whether the data object already exists in memory when you make the LIKE reference.

- In principle, the local data objects in the same program are visible. As with local data types, there is a difference between local data objects in procedures and global data objects. Data objects defined in a procedure obscure other objects with the same name that are declared in the global declarations of the program.
- You can also refer to the data objects of other visible ABAP programs. These might be, for example, the visible attributes of global classes in class pools. If a global class <cl_global> has a public instance attribute or static attribute <attr>, you can refer to it as follows in any ABAP program:

  ```
  DATA <ref> TYPE REF TO <cl_global>.

  DATA: f1 LIKE <cl_global>=><attr>,
        f2 LIKE <ref>-><attr>.
  ```

  You can access the technical properties of an instance attribute using the class name and a reference variable without first having to create an object. The properties of the attributes of a class are not instance-specific and belong to the static attributes of the class.
- To ensure compatibility with previous releases, you can use the LIKE addition to refer to the data types of database tables and **flat** structures in the ABAP Dictionary. The LIKE addition searches first for a data object <obj> in the program, then in the ABAP Dictionary for a database table or flat structure with the same name. You can no longer use this kind of type reference in ABAP Objects classes. You should also avoid using the LIKE addition in other ABAP programs except to refer to data objects. To refer to data types, you should use the TYPE addition instead.

Data Objects

Data objects contain the data with which ABAP programs work at runtime. They are not persistent, but only exist for the duration of the program. Before you can process persistent data (such as data from a database table or from a sequential file), you must read it into data objects first. Conversely, if you want to retain the contents of a data object beyond the end of the program, you must save it in a persistent form.

ABAP contains the following kinds of data objects:

Literals

Literals [Page 119] are not created by declarative statements. Instead, they exist in the program code. Like all data objects, they have fixed technical attributes (field length, number of decimal places, data type), but no name. They are therefore referred to as unnamed data objects.

Named Data Objects

You declare these data objects either statically or dynamically at runtime. Their technical attributes - field length, number of decimal places, and data type - are always fixed. These data objects have a **name** that you can use to address them from ABAP programs. They are therefore referred to as named data objects. ABAP contains the following kinds of named data objects:

Text symbols [Page 121] are pointers to texts in the text pool of the ABAP program. When the program starts, the corresponding data objects are generated from the texts stored in the text pool. They can be addressed using the name of the text symbol.

Variables [Page 123] are data objects whose contents can be changed using ABAP statements. You declare them using the DATA, CLASS-DATA, STATICS, PARAMETERS, SELECT-OPTIONS, and RANGES statements.

Constants [Page 129] are data objects whose contents cannot be changed. You declare them using the CONSTANTS statement.

Interface work areas [Page 130] are special variables that serve as interfaces between programs, screens, and logical databases. You declare them using the TABLES and NODES statements.

Predefined Data Objects

Predefined data objects [Page 132] do not have to be declared explicitly - they are always available at runtime.

Dynamic Data Objects

Dynamic data objects [Page 221] are not declared statically in the declaration part of a program. Instead, you create them dynamically using data references. They do not have a name.

Literals

Literals are unnamed data objects that you create within the source code of a program. They are fully defined by their value. You cannot change the value of a literal. There are two types of literals: **numeric** and **text**.

Number literals

Numeric literals are sequences of digits which may contain a plus or minus sign. They can represent any number within the valid range for the predefined ABAP type P with length 16, that is, a number of up to 31 digits plus a plus or minus sign. Numeric literals between $-2^{31}+1$ and $2^{31}-1$ have the predefined ABAP type I. All other numeric literals have type P without decimal places. Numeric literals of up to 15 digits (plus their plus or minus sign) have a field length of 8 bytes, all others have a field length of 16 bytes.

Examples of numeric literals:

```
123
-93
+456
```

Numeric literals in ABAP statements:

```
DATA number TYPE i VALUE -1234.

WRITE 6789.

MOVE 100 TO number.
```

If you want to use non-integer values or a longer number, you must use a **text literal** (data type C). The text literal in this case must have the format

```
'[<mantissa>][E][<exponent>]'
```

for floating point numbers. When you use text literals of this kind in ABAP statements, the system converts [Page 186] them into the corresponding numeric data type.

Examples of text literals that can be converted into numeric types:

```
'12345678901234567890'
'+0.58498'
'-8473.67'
'-12.34567'
'-765E-04'
'1234E5'
'+12E+23'
'+12.3E-4'
'1E160'
```

Text literals

Text literals are sequences of alphanumeric characters in the source code of an ABAP program enclosed in single quotation marks. They always have the predefined ABAP type C. The field length is determined by the number of characters.

Examples of text literals

```
'Antony Smith'
'69190 Walldorf'
```

Text literals can be up to 255 characters long. A text literal is **always** at least one character long. Entering '' is the equivalent of ' '. If you want to enter a text literal in the ABAP Editor that is longer than a single editor line, ABAP syntax [Page 83] allows you to enter several literals and link them using the & character. If a text literal contains a quotation mark, you must repeat it to enable the system to recognize the contents as a text literal and not as the end of the literal.

```
WRITE: / 'This is John''s bicycle'.
```

This statement generates the following output:

```
This is John's bicycle
```

Text Symbols

A text symbol is a named data object that is generated when you start the program from the texts in the text pool of the ABAP program. It always has the data type C. Its field length is that of the text in the text pool.

Text symbols, along with the program title, list headings, and selection texts, belong to the **text elements** of a program. Text elements allow you to create language-independent programs. Any text that the program sends to the screen can be stored as a text element in a text pool. Different text pools can be created for different languages. When a text element is changed or translated, there is no need to change the actual program code. Text elements in an ABAP program are stored in the ABAP Editor (see Text Element Maintenance [Ext.]).

In the text pool, each text symbol is identified by a three-character ID. Text symbols have a content, an occupied length, and a maximum length.

Examples for text symbols in an ABAP program:

ID	Contents	Occupied length	Maximum length
010	Text symbol 010	15	132
030	Text symbol 030	15	100
AAA	Text symbol AAA	15	15

In the program, you can address text symbols using the following form:

TEXT-<idt>

This data object contains the text of the text symbol with ID <idt> in the logon language of the user. Its field length is the same as the maximum length of the text symbol. Unfilled characters are filled up with spaces. You can address text symbols anywhere in a program where it is also possible to address a variable.

If there is no text symbol <idt> in the text pool for the logon language, the name TEXT-<idt> addresses the predefined data object SPACE instead.

You can also address text symbols as follows:

... '<textliteral>'(<idt>) ...

If the text symbol <idt> exists in the text pool for the logon language, this is the same as using TEXT-<idt>. Otherwise, the literal '<textliteral>' is used as the contents of the text symbol. This is only possible at positions in the program where a variable can occur. You can create a text symbol for any text literal by double-clicking the literal in the ABAP Editor and replacing the literal with the text symbol.

You should use text symbols in your program whenever they need to be language-specific - for example, in a WRITE statement.

If you program a list whose layout depends on field lengths, you should be careful, since the field length of text symbols will be different in different languages. You should therefore set the maximum field length of the field symbol so that there is enough space to translate it into other languages. For example, the English word 'program' has seven letters, but its equivalent German translation 'Programm' has eight.

Text Symbols

The following example shows the use of text symbols in WRITE statements.

```
SET BLANK LINES ON.

WRITE:   text-010,
       / text-aaa,
       / text-020,
       / 'Default Text 030'(030),
       / 'Default Text 040'(040).
```

If the text symbols of the above screen shots are linked to this program, the output looks as follows:

```
Text Symbol AAA
Text Symbol 010

Text Symbol 030
Default Text 040
```

Text symbols 020 and 040 have no text symbols. For text symbol 020, the system displays a space. This is only displayed in this case because the blank line suppression has been turned off (see Creating Blank Lines [Page 799]). For text symbol 040, the literal specified in the program code is displayed.

Variables

Variables are named data objects that you can declare statically using declarative statements, or dynamically while a program is running. They allow you to store changeable data under a particular name within the memory area of a program.

You can declare variables **statically** using the following statements:

- DATA: To declare variables whose lifetime is linked to the context of the declaration
- STATICS: To declare variables with static validity in procedures
- CLASS-DATA: To declare static variables within classes
- PARAMETERS: To declare elementary data objects that are also linked to an input field on a selection screen
- SELECT-OPTIONS: To declare an internal table that is also linked to input fields on a selection screen
- RANGES: To declare an internal table with the same structure as in SELECT-OPTIONS, but without linking it to a selection screen.

This section explains the DATA and STATICS statements. For further information about CLASS-DATA; refer to Classes [Page 1300] . For further information about PARAMETERS, SELECT-OPTIONS, and RANGES, refer to Selection Screens [Page 681] .

Variables are **declared dynamically** when you add characters or bytes to a string, or lines to an internal table [Page 251]. After you have declared a string, only its type is defined. When you declare an internal table, the line type, access type, and key are defined. The actual data objects - the characters and bytes for a string, or the lines of an internal table - are created dynamically at runtime.

You can also create data objects dynamically when you call procedures [Page 449] . These data objects are the formal parameters of the interface definition, which only have technical attributes when they inherit them from the actual parameters passed to them.

The DATA Statement

You use the DATA statement to declare variables in an ABAP program or instance attributes in a class. Within the program or class, you can also declare local variables within procedures [Page 449]. The same rules of visibility apply to variables as to types (see The TYPE Addition [Page 112]). Local variables in procedures obscure identically-named variables in the main program or class.

The DATA statement has a similar syntax to the TYPES statement:

```
DATA <f> ... [TYPE <type>|LIKE <obj>]... [VALUE <val>].
```

The variable name <f> may be up to 30 characters long. You can use all characters except for + . , : ( ). A name may also not consist entirely of digits. Names of predefined data objects [Page 132] cannot be changed. You should not use names that are the same as an ABAP keyword or addition. You should:

- Use names that explain the meaning of the variable without the need for further comments
- Do not use hyphens - these are reserved for addressing the components of structures

Variables

- Use the underscore character to separate compound words
- Avoid using any special characters
- Always use a letter as the first character of a variable name.

When you declare a variable statically, you define all of its technical attributes, that is, its length, data type, and number of decimal places. You can do this in the following ways:

Referring to Existing Technical Attributes

You can create a variable that inherits exactly the same technical attributes as an existing data type or data object as follows:

```
DATA <f> [TYPE <type>|LIKE <obj>]...
```

If you use the TYPE addition [Page 112], <type> is any data type with fully-specified technical attributes. This can be a:

- Non-generic predefined ABAP type [Page 97] (D, F, I, T, STRING, XSTRING)
- Any existing local data type [Page 100] in the program.
- Any ABAP Dictionary data type [Page 105]

If you use the LIKE addition, <obj> is a data object that has already been declared. This can also be a predefined data object [Page 132]. The variable <f> adopts the same technical attributes as the data object <obj>. You can also use LIKE to refer to a line of an internal table that has already been declared as a data object:

```
DATA <f> LIKE LINE OF <itab>.
```

To ensure compatibility with previous releases, <obj> can also be a database table, a view, a structure, or a component of a structure from the ABAP Dictionary.

The data types [Page 92] to which you refer can be elementary types, reference types, or complex types (structures or tables). For elementary field types, the variables are a single field in memory. When you declare a data type with fixed length (D, F, I, T) the system fixes the amount of memory that will be assigned. When you declare an object with a variable length (STRING, XSRTING), the system only assigns enough memory to administer the object. The length of the data object is managed dynamically at runtime. For structures, the variables are a sequence of variables, which may themselves also be included in further complex structures. The individual components take their name <c_i> from the type <type> or object <obj>, and can be addressed using <f>-<c_i> For tables, the memory contains administration entries that can be filled dynamically at runtime.

```
TYPES: BEGIN OF struct,
         number_1 TYPE i,
         number_2 TYPE p DECIMALS 2,
       END OF struct.

DATA:  wa_struct TYPE struct,
       number    LIKE wa_struct-number_2,
       date      LIKE sy-datum,
       time      TYPE t,
       text      TYPE string,
       company   TYPE s_carr_id.
```

This example declares variables with reference to the internal type STRUCT in the program, a component of an existing data object WA_STRUCT, the predefined data object SY-DATUM, the predefined ABAP type T and STRING, and the data element S_CARR_ID from the ABAP Dictionary.

Referring to Data Types with Generic Attributes

If you refer to a generic predefined ABAP type in the DATA statement (types C, N, P, or X), you must specify the undefined technical attributes in the statement. The DATA statement has the same syntax as the TYPES statement for this purpose:

```
DATA <f>[(<length>)] TYPE <type> [DECIMALS <d>]...
```

The <length> option sets the field length. If you omit it, the field length is set to the relevant initial value in the table in the Predefined ABAP Types [Page 97] section. If <type> is P, you can specify the number of decimal places using the DECIMALS <d> addition. If you omit this, the number of decimal places is set to 0.

If you do not use the TYPE or LIKE addition, the system uses the default predefined generic type C. Thus the implicit statement

```
DATA <f>.
```

defines a character variable <f> with length 1. It is a shortened form of the explicit statement DATA <f>(1) TYPE C.

```
DATA <f>(1) TYPE C.
```

The maximum number of decimal places for a packed number is 14. To use the decimal places, the program attribute *Fixed point arithmetic* must be set, otherwise, the variables will be treated like integers. When you assign a value to a packed number, non-significant figures are rounded.

It is not currently possible to refer to generic data types in the DATA statement other than to generic standard tables [Page 261] without a defined key.

```
DATA: text1,
      text2(2),
      text3(3) TYPE c,
      pack TYPE P DECIMALS 2 VALUE '1.225'.
```

This example creates three character variables with lengths one, two, and three bytes respectively, and a packed number variable with length 8 bytes and two decimal places. If the attribute *Fixed point arithmetic* is set, the value of PACK is 1.23.

Creating Variables With Their Own Data Type

We have so far seen how you can define variables by referring to existing data types. However, the DATA statement also allows you to use the same **type constructors** as in the TYPES statement to assign a data type to a variable when you declare it. This data type then does not exist in its own right, but only as an attribute of the corresponding data object. The data type is linked to the data object. You can refer to it using the LIKE addition, but not using TYPE.

References

The syntax for a direct reference variable declaration is the same as the definition using the TYPES statement:

Variables

```
DATA <f> TYPE REF TO ...
```

After TYPE, there is no reference to an existing data type. Instead, the type constructor occurs:

- The type constructor

  ```
  REF TO DATA
  ```

  defines a field <f> with the data type reference to a data object. The reference variable <f> can contain references (pointers) to data objects, that is, instances of data types (see also Data References [Page 219]).

- The type constructor

  ```
  REF TO <class>|<interface>
  ```

  defines a field <f> with the data type reference to an object in ABAP Objects. The reference variable <f> can contain references (pointers) to instances of the class <class> or its subclasses, or to classes that implement the interface <interface> respectively. See also Object Handling [Page 1307].

structures

The syntax for declaring a structure directly is the same as you would use to define a structure using the TYPES statement:

```
DATA: BEGIN OF <structure>,
                .............
<fi>...,
                .............
END OF <structure>.
```

This chained statement creates a structure containing all of the variables between

DATA BEGIN OF <structure>. and DATA END OF <structure>.

that occur in the

```
DATA <fi>....
```

statements. As in the TYPES statement, the DATA BEGIN OF - DATA END OF statement blocks can be nested. The components <f_i> can be elementary fields, reference variables, or, if you refer to known complex types, complex themselves. Since fields of type I or F are aligned (see Aligning Data Objects [Page 195]), the system inserts empty filler fields between the components, if necessary.

The individual variables within a structure are addressed in the program with a hyphen between the structure name and component name as follows: <structure>-<f_i>.

```
DATA: BEGIN OF address,
         name(20)    TYPE c,
         street(20)  TYPE c,
         number      TYPE p,
         postcode(5) TYPE n,
         city(20)    TYPE c,
       END OF address.
```

This example defines a structure called ADDRESS. The components can be addressed using ADDRESS-NAME, ADDRESS-STREET, and so on.

Internal tables

The syntax for declaring an internal table directly as a data type of a variable is the same as you would use to define one using the TYPES statement:

```
DATA <f> TYPE|LIKE <tabkind> OF <linetype> WITH <key>.
```

After TYPE, there is no reference to an existing data type. Instead, the type constructor occurs:

```
      <tabkind> OF <linetype> WITH <key>.
```

The variable <f> is declared as an internal table with access type <tabkind>, line type <linetype>, and key <key>. The line type <linetype> can be any known data type. For more information, refer to Internal Tables [Page 251].

Specifying a Start Value

When you declare an elementary fixed-length variable, the DATA statement automatically fills it with the type-specific initial value as listed in the table in the Predefined ABAP Types [Page 97] section.

However, you can also specify a starting value of a fixed-length **elementary** variable (also within a structure declaration) using the VALUE addition in the DATA statement:

```
DATA <f>..... VALUE <val>.
```

The start value of the field <f> in the program is set to <val>, where <val> can be

- a literal [Page 119]
- a constant [Page 129] that has already been declared
- the explicit addition IS INITIAL

You cannot use other variables or text symbols [Page 121] in the VALUE addition. You cannot assign a starting value to a variable-length data object (STRING, XSTRING), an internal table, or a reference variable.

Examples of start value specifications:

```
DATA: counter TYPE p VALUE 1,
      date    TYPE d VALUE '19980601',
      flag    TYPE n VALUE IS INITIAL.
```

After this data declaration, the variable FLAG contains its type specific initial value ‘0’.

Static Variables in Procedures

Variables that you declare with the DATA statement live for as long as the context in which they are defined. So variables in an ABAP main program exist for the entire runtime of the program, and local variables in procedures [Page 449] only exist for as long as the procedure is running.

To retain the value of a local variable beyond the runtime of the procedure, you can declare it using the STATICS statement. This declares a variable with the lifetime of the context of the main program, but which is only visible within the procedure.

The first time you call a subroutine or function module, the corresponding main program is always loaded into the internal session of the calling program. It is **not** deleted when the

procedure ends. This enables variables defined using STATICS to retain their values beyond the runtime of the procedure, allowing them to be reused the next time the procedure is called (see the example in the Local Data in Subroutines [Page 455] section).

In methods, variables defined with STATICS are static attributes that are only visible in the corresponding method, but for all instances of a class (see Classes [Page 1300]).

The syntax of the STATICS statement is identical to that of the DATA statement.

Constants

Constants are named data objects that you create statically using a declarative statement. They allow you to store data under a particular name within the memory area of a program. The value of a constant must be defined when you declare it. It cannot subsequently be changed. The value of a constant cannot be changed during the execution of the program. If you try to change the value of a constant, a syntax error or runtime error occurs.

You declare them using the CONSTANTS statement. Within the program, you can also declare local variables within procedures [Page 449] using CONSTANTS. The same rules of visibility apply to constants as to types (see The TYPE Addition [Page 112]). Local constants in procedures obscure identically-named variables in the main program. Constants live for as long as the context in which they are declared.

The syntax of the CONSTANTS statement is exactly the same as that of the DATA statement, but with the following exceptions:

- You must use the VALUE addition in the CONSTANTS statement. The start value specified in the VALUE addition cannot be changed during the execution of the program.
- You cannot define constants for strings, references, internal tables, or structures containing internal tables.

Elementary constants:

```
CONSTANTS: pi TYPE P DECIMALS 10 VALUE '3.1415926536'.

           ref_c1 TYPE REF TO C1 VALUE IS INITIAL.
```

The last line shows how you can use the IS INITIAL argument in the VALUE addition. Since you must use the VALUE addition in the CONSTANTS statement, this is the only way to assign an initial value to a constant when you declare it.

Complex constants:

```
CONSTANTS: BEGIN OF myaddress,
           name(20)    TYPE c  VALUE 'Fred Flintstone',
           street(20)  TYPE c  VALUE 'Cave Avenue',
           number      TYPE p  VALUE  11,
           postcode(5) TYPE n  VALUE  98765,
           city(20)    TYPE c  VALUE  'Bedrock',
           END OF myaddress.
```

This declares a constant structure MYADDRESS. The components can be addressed by MYADDRESS-NAME, MYADDRESS-STREET, and so on, but they cannot be changed.

Interface Work Areas

Interface work areas are special named data objects that are used to pass data between

- Screens and ABAP programs
- Logical databases and ABAP programs
- ABAP programs and external subroutines.

Interface work areas are created in a shared data area of the programs between which the data is to be exchanged. All of the programs and procedures involved access this work area. To find out how to protect shared work areas against changes in procedures, refer to Global Data of the Main Program [Page 453].

Screen Interfaces

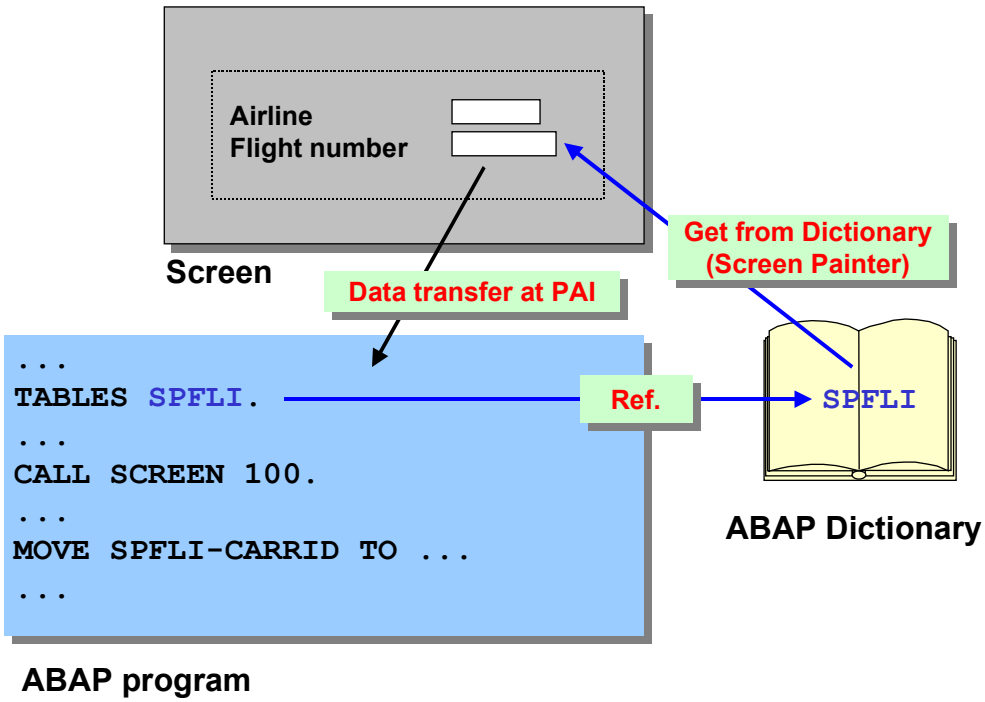
One common kind of interface work area are table work areas, which you declare using the

```
TABLES <dbtab>.
```

statement. This statement creates a **structure** with the same data type and the **same name** as a database table, a view, or a structure from the ABAP Dictionary.

Before Release 4.0, it was necessary to declare a database table <dbtab> to an ABAP program using the statement TABLES <dbtab> before you could access the table using Open SQL statements. This restriction no longer applies to Open SQL statements.

However, you still need to use the TABLES statement if you want to define input/output fields on a screen with reference to database tables, views, or ABAP Dictionary structures (*Get from Dict.* function in the layout editor of the Screen Painter [Ext.]). In the PBO event, the ABAP program sends the contents of the fields to the screen fields. In the PAI event, it receives the contents of the screen fields back into the corresponding components of the table work area.



Logical Database Interfaces

Logical databases are special ABAP programs that read data for application programs. When a logical database is linked with an executable program, it passes data to the interface work area declared with the

```
NODES <node>.
```

statement. This statement creates a variable <node> with reference to an ABAP Dictionary data type of the same name. If a node of the logical database refers to the same data type, the variable serves as an interface between the logical database and the executable program. If a node of a logical database refers to a database table, a view, or a flat structure from the ABAP Dictionary, you can use the TABLES statement instead of NODES.

For further information refer to Logical Databases [Page 1163].

External Subroutine Interfaces

If you have similar variables in different programs, you can store them in a shared data area. To do this, use the COMMON PART [Page 477] addition of the DATA statement. Common data areas of this kind can serve as interfaces to external subroutines. You can also use interface work areas defined with the TABLES and NODES statement as a common data area for external subroutines [Page 494].

Predefined Data Objects

At runtime, the following data objects are always present, and do not have to be declared:

SPACE

The data object SPACE is a constant with type C. It is one byte long, and contains a space character. SPACE is a constant, and as such, cannot be changed.

System Fields From Structure SY

SY is a structure with the ABAP Dictionary data type SYST. The components of SY are known as system fields. System fields contain values that provide information about the current state of the system. They are automatically filled and updated by the ABAP runtime environment. Examples of system fields:

- SY-SUBRC: Return code for ABAP statements
  (zero if a statement is executed successfully)
- SY-UNAME: logon name of the user
- SY-REPID: Current ABAP program
- SY-TCODE: current transaction
- SY-INDEX: Number of the current loop pass

System fields are variables and can be changed by the program. However, you should only do so where the system documentation recommends it for a specific field (for example, you can change the value of SY-LSIND to navigate between detail lists). In all other cases, you should treat system fields as though they were constants, and only read them. If you change their values, important information for the flow of the program may be lost.

For a complete list of all system fields with notes on their use, refer to

ABAP System Fields [Page 1444]

Compatibility

Two data types or data objects are compatible when **all of their technical attributes** (field length, number of decimal places, type) **are exactly the same**.

Do not confuse the two terms **compatible** and **convertible**. When working with data objects, you will often make assignments between data objects that have **different technical attributes**. In this case, the data types are **converted**. In order for non-compatible data types to be converted, a conversion rule [Page 186] must exist. Only **compatible** data objects can be assigned to one another **without a conversion** being necessary.

Compatibility applies to fully-specified data types, since all data objects are fully typed at runtime. This compatibility is symmetrical. There is also an asymmetrical compatibility, which is used to test the types of interface parameters in procedures and of field symbols. In this case, generic types are compared with fully-specified data types.

Consequences of definition for the compatibility of data types:

Elementary Data Types

Elementary data types are compatible with other elementary data types if they are identical in type, field length, and the number of decimal places.

Elementary data types are not compatible with references or aggregated types.

References

References are compatible with other references when their construction rules (type constructor REF TO <class>|<interface>) are the same.

References are not compatible with elementary data types or aggregated types.

Aggregated Data Types

Aggregated data types are compatible with other aggregated data types if their construction rules are the same.

Aggregated data types are not compatible with elementary data types or references.

structures

Structures are compatible with other structures if their constructions are identical and their components are compatible. This means that the way in which field strings are constructed from elementary fields to form the overall structure from sub-structures must be the same and their elementary components must be compatible with each other. If two structures consist of the same sequence of elementary fields, but these fields are combined differently to substructures, the structures are **not** compatible.

Structures are not compatible with internal tables.

Internal tables

Internal tables are compatible with other internal tables if their line types are compatible and all other attributes are also the same for both tables. The compatibility of internal tables does not depend on the number of lines.

Internal tables are not compatible with structures.

Determining the Attributes of Data Objects

You may sometimes need to find out the attributes of a data object at runtime that were not statically available. For example, you may need to find out the type of a generic interface parameter in a subroutine. To do this, you would use the statement:

```
DESCRIBE FIELD <f> [LENGTH <l>] [TYPE <t> [COMPONENTS <n>]]
                   [OUTPUT-LENGTH <o>] [DECIMALS <d>]
                   [EDIT MASK <m>] [HELP-ID <h>].
```

The attributes of the data object <f> specified by the parameters of the statement are written to the variables following the parameters. You can use any number of the additions in the same statement.

```
DESCRIBE FIELD DISTANCE BETWEEN <f1> AND <f2> INTO <d>.
```

This statement returns the distance between the data objects <f_1> and <f_2>.

The next sections describe the additions in detail:

Determining the Field Length

To find out the field length of a data objet, use the LENGTH addition:

```
DESCRIBE FIELD <f> LENGTH <l>.
```

The statement writes the length of the field <f> into the variable <l>.

```
DATA: text(8) TYPE c,
      len TYPE i.

DESCRIBE FIELD text LENGTH len.
```

Field LEN contains the value 8.

Determining the Data Type

To find out the data type of a data objet, use the TYPE addition:

```
DESCRIBE FIELD <f> TYPE <t>.
```

The statement writes the type of the field <f> into the variable <t>.

Apart from the predefined ABAP type [Page 97] of the field (C, D, F, I, N, P, T, X), the statement can return the following values:

- **s** for two-byte integers with leading sign
- **b** for one-byte integers without leading sign
- **r** for references
- **h** for internal tables

These values are internal type identifiers. Data objects with the internal types s and b may have been declared using references to corresponding elementary types in the ABAP Dictionary. Data objects with type h are aggregated. Data objects with internal table r are reference variables in ABAP Objects.

Determining the Attributes of Data Objects

If <f> is a structure, the statement also returns the value C. To find out more about a structure, you must use the extra addition COMPONENTS:

```
DESCRIBE FIELD <f> TYPE <t> COMPONENTS <n>.
```

Instead of C, the statement writes the following internal type identifiers into the variable <t>:

- **u** for structures without an internal table as a component
- **v** for structures with at least one internal table as a component or subcomponent

The number of direct components of the structure is written into the variable <n> as an integer.

```
TABLES spfli.
DATA: numtext(8) TYPE n,
      typ(1) TYPE c.

DESCRIBE FIELD numtext TYPE typ.
WRITE typ.

DESCRIBE FIELD spfli-fltime TYPE typ.
WRITE typ.
```

This example produces the following output:

```
N T
```

The field TYP contains first the value N, then T.

```
TYPES: surname(20)  TYPE c,
       street(30)   TYPE c,
       zip_code(10) TYPE n,
       city(30)     TYPE c,
       phone(20)    TYPE n,
       date         LIKE sy-datum.

TYPES: BEGIN OF address,
         NAME TYPE surname,
         CODE TYPE zip_code,
         TOWN TYPE city,
         STR  TYPE street,
       END OF address.

TYPES: BEGIN OF phone_list,
         adr TYPE address,
         tel TYPE phone,
       END OF phone_list.

DATA pl TYPE phone-list.

DATA: typ(1) TYPE c,
      n TYPE i.

DESCRIBE FIELD pl TYPE typ COMPONENTS n.
```

Here, the field TYP contains the value 'u' and N contains the value 2 because PL is a structure with two direct components (ADR and TEL) and without internal tables.

Determining the Output Length

To find out the field length of a data object, use the OUTPUT-LENGTH addition:

```
DESCRIBE FIELD <f> OUTPUT-LENGTH <o>.
```

The statement writes the output length of the field <f> into the variable <o>. The output length of a field is the number of characters occupied by the field contents on a list after a WRITE statement [Page 775].

```
DATA: float TYPE f,
      out TYPE i,
      len TYPE i.

DESCRIBE FIELD float LENGTH len OUTPUT-LENGTH out.
```

This example results in field LEN containing the value 8 and the field OUT containing the value 22.

Determining the Decimal Places

To find out the number of decimal places occupied by a data objet, use the DECIMALS addition:

```
DESCRIBE FIELD <f> DECIMALS <d>.
```

The statement writes the number of decimal places in field <f> into the variable <d>.

```
DATA: pack TYPE p DECIMALS 2, dec(1) TYPE c.

DESCRIBE FIELD pack DECIMALS dec.
```

This example results in field DEC containing the value 2.

Determining the Conversion Routine

To find out any conversion routine defined for the field in the ABAP Dictionary, use the EDIT MASK addition:

```
DESCRIBE FIELD <f> EDIT MASK <m>.
```

If <f> was declared with reference to an ABAP Dictionary data type and there is a conversion routine <conv> for the field, the statement writes it into the variable <m> in the form '==<conv>'. You can specify a conversion routine [Ext.] for the domain of a data element in the ABAP Dictionary. Each conversion routine is linked to two function modules that allow the ABAP Dictionary representation of a value to be converted into the program representation and vice versa. The variable <m> can be used as an edit mask in a WRITE statement.

```
DATA: fltime TYPE s_fltime,
      m(7) TYPE c.

DESCRIBE FIELD fltime EDIT MASK m.

fltime = sy-uzeit.
```

```
WRITE: / fltime,
       / sy-uzeit,
       / sy-uzeit USING EDIT MASK m.
```

The domain S_DURA of data element S_FLTIME in the ABAP Dictionary is linked to the conversion routine SDURA. The field M contains the value ==SDURA. This produces the following output list:

```
604:55
10:04:55
604:55
```

The conversion routine SDURA converts hours into minutes. Internally, field FLTIME is handled in the converted form. Field SY-UZEIT is displayed in the list in the converted form.

Determining the Help Text ID

You can find out the identifier of the help text (F1 help) defined for a field in the ABAP Dictionary using the HELP-ID addition:

```
DESCRIBE FIELD <f> HELP-ID <h>.
```

If the field <f> is defined with reference to a data type from the ABAP Dictionary, the statement writes the help text ID into the variable <h>. You can use the ID in a suitable function module to display the help text.

```
DATA: company TYPE s_carr_id,
      h(20) TYPE c,
      tlink TYPE TABLE OF tline.

DESCRIBE FIELD company HELP-ID h.

CALL FUNCTION 'HELP_OBJECT_SHOW'
     EXPORTING
          dokclass                = 'DE'
          doklangu                = sy-langu
          dokname                 = h
     TABLES
          links                   = tlink
     EXCEPTIONS
          object_not_found        = 1
          sapscript_error         = 2
          others                  = 3.

IF sy-subrc <> 0.
  ...
ENDIF.
```

In this program, the field H receives the name of the data element S_CARR_ID. The function module HELP_OBJECT_SHOW displays the documentation for the data element in a dialog box.

Determining the Distance Between Two Fields

To find out the distance between two fields in memory, use the following statement:

```
DESCRIBE FIELD DISTANCE BETWEEN <f1> AND <f2> INTO <d>.
```

The statement returns the number of bytes between data objects $<f_1>$ and $<f_2>$ into the variable <d>. The length of the first field in memory is always included. Any alignment is taken into account.

```
DATA: BEGIN OF test,
        col1(3) TYPE C,
        col2(2) TYPE C,
        col3 TYPE i,
      END OF test,
      dist TYPE i.

DESCRIBE DISTANCE BETWEEN test-col3 AND test-col1 INTO dist.
```

The field DIST has the value 8. These are the lengths 3 and 2 of the components COL1 and COL2, and an alignment gap of 3 characters between COL2 and COL3.

Examples of Data Types and Objects

This section contains examples of elementary and aggregated data types and objects as often used in ABAP programs.

This example shows how to declare elementary data objects with reference to predefined ABAP types.

```
PROGRAM demo_elementary_data_objects.

DATA text1(20) TYPE c.
DATA text2     TYPE string.
DATA number    TYPE i.

text1 = 'The number'.
number = 100.
text2 = 'is an integer.'.

WRITE: text1, number, text2.
```

This program produces the following output on the screen:

```
The number              100 is an integer.
```

In this example, the **data objects** TEXT1, TEXT2, and NUMBER are declared with the DATA statement. The technical attributes are determined by referring to the predefined ABAP types C, STRING, and I. Values from unnamed literals are assigned to the data objects. The contents of the named data objects are displayed on the list.

This example shows how to declare local elementary data types within a program.

```
REPORT demo_types_statement.

TYPES mytext(10) TYPE c.
TYPES myamount   TYPE p DECIMALS 2.

DATA text        TYPE mytext.
DATA amount      TYPE myamount.

text = ' 4 / 3 = '.
amount = 4 / 3 .

WRITE: text, amount.
```

This program produces the following output on the screen:

```
4 / 3 =                 1.33
```

The program uses the TYPES statement to create the local **data types** MYTEXT and MYAMOUNT. The technical attributes are defined with reference to predefined ABAP types. Then, the **data objects** TEXT and AMOUNT are declared with the DATA statement. Their **data types** are defined with reference to MYTEXT and MYAMOUNT. Values are assigned to the **data objects** and the contents of the data objects are displayed on the screen.

This example shows how to declare structures.

```
REPORT demo_structure.

TYPES: BEGIN OF name,
         title(5)       TYPE c,
         first_name(10) TYPE c,
         last_name(10)  TYPE c,
       END OF name.

TYPES: BEGIN OF mylist,
         client         TYPE name,
         number         TYPE i,
       END OF mylist.

DATA list TYPE mylist.

list-client-title = 'Lord'.
list-client-first_name = 'Howard'.
list-client-last_name = 'Mac Duff'.
list-number = 1.

WRITE list-client-title.
WRITE list-client-first_name.
WRITE list-client-last_name.
WRITE / 'Number'.
WRITE list-number.
```

This program produces the following output on the screen:

```
Lord  Howard      Mac Duff
Number           1
```

The local data types NAME and MYLIST are defined as structures using the TYPES statement. The structure NAME contains the three components TITLE, FIRST_NAME and LAST_NAME, which refer to the predefined ABAP type C. The structure MYLIST contains the components CLIENT and NUMBER. CLIENT is a substructure that refers to the structure NAME. NUMBER is an elementary field with the predefined type I. A **data object** LIST is defined with reference to the **data type** MYLIST. Values are assigned to the components and their contents are then displayed on the screen.

This example shows how to define an internal table.

```
PROGRAM demo_internal_table.

TYPES: BEGIN OF mytext,
         number TYPE i,
         name(10) TYPE c,
       END OF mytext.

TYPES mytab TYPE STANDARD TABLE OF mytext WITH DEFAULT KEY.

DATA text TYPE mytext.
DATA itab TYPE mytab.
```

Examples of Data Types and Objects

```
text-number = 1. text-name = 'John'.
APPEND text TO itab.

text-number = 2. text-name = 'Paul'.
APPEND text TO itab.

text-number = 3. text-name = 'Ringo'.
APPEND text TO itab.

text-number = 4. text-name = 'George'.
APPEND text TO itab.

LOOP AT itab INTO text.
  WRITE: / text-number,text-name.
ENDLOOP.
```

This program produces the following output on the screen:

```
1  John

2  Paul

3  Ringo

4  George
```

In this example, first a **data type** MYTEX is defined as a structure. Then, a data type MYTAB is defined as an internal table with the line type MYTEXT. The **data objects** TEXT and ITAB are declared with reference to the internal data types MYTEXT and MYTAB. This lines of the internal table ITAB are generated dynamically with the APPEND statement. The contents of the internal table ITAB are written to the list using the structure TEXT. See also: Internal Tables [Page 251]

Processing Data

This section describes how to work with data objects in ABAP.

It contains the following topics:

Assigning Values

This section describes the ABAP statements that you use to change the contents of variable fields. At the beginning of an ABAP program, its variables have a starting value, which you can set in the DATA statement. There are many different ways in which you can change the value of a variable; for example, using the user interface, Open SQL, or by assigning the values of data objects in your program to a variable. The most important statement for the latter method is MOVE. It has the same meaning as the equal sign (=). Most other value assignments are reduced internally to the semantics of the MOVE statement. This is particularly important in type conversions.

Another statement that you can use to pass values to variables (mainly to fill character fields) is WRITE TO.

Finally, there is the CLEAR statement, which allows you to reset a data object to the initial value appropriate to its type.

Assigning Values with MOVE [Page 145]

Assigning Values with WRITE TO [Page 148]

Resetting Values to Their Initial Value [Page 150]

Assigning Values with MOVE

To assign the value of a data object <f1> to a variable <f2>, use the following statement:

MOVE <f1> TO <f2>.

or the equivalent statement

<f2> = <f1>.

The contents of <f1> remain unchanged. <f1> does not have to be a variable - it can also be a literal, a text symbol, or a constant. You must always specify decimal points with a period (.), regardless of the user's personal settings.

Multiple value assignments in the form

<f4> = <f3> = <f2> = <f1>.

are also possible. ABAP processes them from right to left as follows:

MOVE <f1> TO <f2>.
MOVE <f2> TO <f3>.
MOVE <f3> TO <f4>.

In the MOVE statement (or when you assign one value to another with the equal sign), it is not possible to specify the field names dynamically as the contents of other fields. If you need to do this, you must use field symbols [Page 201] .

There are three possible outcomes of assigning <f1> to <f2>:

1. The data objects <f1> and <f2> are fully compatible [Page 133], that is, their data types, field length, and number of decimal places are identical. The contents of source field <f1> are transferred byte by byte into the target field <f2> without any further manipulation. The MOVE statement is most efficient when this is the case.
2. The data objects <f1> and <f2> are incompatible. This is the case, for example, if the two fields have the same type, but different lengths. The contents of the source field <f1> are converted so that they are compatible with the data type of <f2>, and are then transferred. This procedure only works if a conversion rule [Page 186] exists between the data types of <f1> and <f2>. Type conversions make the MOVE statement less efficient. How much less efficient depends on the individual conversion.
3. The data objects <f1> and <f2> are incompatible, and no conversion is possible. The assignment is not possible. If this can be recognized statically, a syntax error occurs. If it is not recognized before the program is run, a runtime error occurs.

The source and target fields can be of different data types. In contrast to other programming languages, where the assignment between different data types is often restricted to a small number of possible combinations, ABAP provides a wide range of automatic type conversions.

For example, the contents of a source field with an elementary data type can be assigned to a target field with any other data type. The single exception to this rule is that it is not possible to assign values between type D fields and type T fields. ABAP even supports assignments between a structure and an elementary field, or between two structures.

Assigning Values with MOVE

```
DATA: T(10) TYPE C,
      NUMBER TYPE P DECIMALS 2,
      COUNT  TYPE I.

T = 1111.
MOVE '5.75' TO NUMBER.
COUNT = NUMBER.
```

Following these assignments, the fields T, NUMBER, and COUNT have the values ‘1111 ’, 5.75, and 6 respectively. When you assign the number literal 1111 to T, it is converted into a character field with length 10. When you assign NUMBER to COUNT, the decimal number is rounded to an integer (as long as the program attribute *Fixed pt. arithmetic* has been set).

Assigning Values Between Components of Structures

To move values between the components of structures, use the statement

MOVE-CORRESPONDING <struct1> TO <struct2>.

This statement moves the contents of the components of structure <struct1> to the components of <struct2> that have **identical names**.

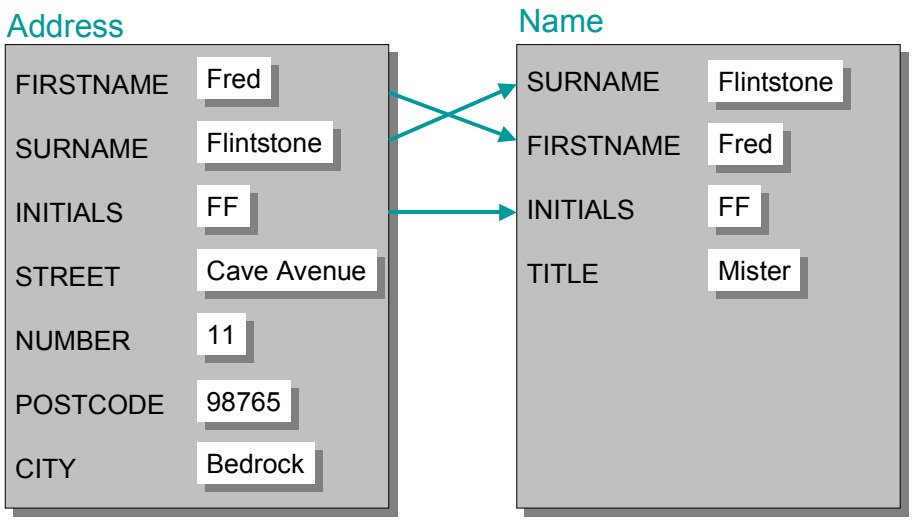
When it is executed, it is broken down into a set of MOVE statements, one for each pair of fields with identical names, as follows:

MOVE STRUCT1-<c_i> TO STRUCT2-<c_i>.

Any necessary type conversions occur at this level. This process is different to that used when you assign a whole structure using the MOVE statement, where the conversion rules for structures [Page 192] apply.

```
DATA: BEGIN OF ADDRESS,
        FIRSTNAME(20) VALUE 'Fred',
        SURNAME(20) VALUE 'Flintstone',
        INITIALS(4) VALUE 'FF',
        STREET(20) VALUE 'Cave Avenue,
        NUMBER TYPE I VALUE '11'.
       POSTCODE TYPE N VALUE '98765'.
       CITY(20) VALUE  'Bedrock',
      END OF ADDRESS.

DATA: BEGIN OF NAME,
        SURNAME(20),
        FIRSTNAME(20),
        INITIALS(4),
        TITLE(10) VALUE 'Mister',
      END OF NAME.

MOVE-CORRESPONDING ADDRESS TO NAME.
```

In this example, the values of NAME-SURNAME, NAME-FIRSTNAME and NAME-INITIALS are set to 'Flintstone’, ‘Fred’, and 'FF'. NAME-TITLE retains the value 'Mister'.


Address
FIRSTNAME
Fred
SURNAME
Flintstone
INITIALS
FF
STREET
Cave Avenue
NUMBER
11
POSTCODE
98765
CITY
Bedrock
Name
SURNAME
Flintstone
FIRSTNAME
Fred
INITIALS
FF
TITLE
Mister

Assigning Values with WRITE TO

As well as the MOVE statement, which converts the value of the source field into the data type of the target field, there is also a statement WRITE TO, which converts the contents of the source field into a field with type C.

WRITE <f1> TO <f2> [<option>].

This statement converts [Page 186] the contents of a data object <f1> to type C, and places the string in the variable <f2>. The data type of <f1> must be convertible into a character field; if it is not, a syntax or runtime error occurs. The contents of <f1> remain unchanged. The variable <f2> is always interpreted as a character string, regardless of its actual line type. The conversion does not take other data types into account. Therefore, you should not use a target field with a numeric data type (F, I , or P). If you do use a numeric data type, the syntax check displays a warning if the target field is statically defined. However, this warning may be upgraded to a real syntax or runtime error in future releases.

The WRITE TO statement always checks the settings in the user's master record. These specify, for example, whether the decimal point appears as a period (.) or a comma (,). You can also use all of the formatting options available with the WRITE [Page 775] statement, apart from UNDER and NO-GAP.

```
DATA: NUMBER TYPE F VALUE '4.3',
      TEXT(10),
      FLOAT TYPE F,
      PACK  TYPE P DECIMALS 1.

WRITE NUMBER.

WRITE NUMBER TO TEXT EXPONENT 2.
WRITE / TEXT.

WRITE NUMBER TO FLOAT.
WRITE / FLOAT.

WRITE NUMBER TO PACK.
WRITE / PACK.

MOVE NUMBER TO PACK.
WRITE / PACK.
```

In this example, the syntax check would warn you that PACK and FLOAT should be character fields (types C, D, N, T, or X). The output list is as follows:

4.30000000000000E+00

0.043E+02

1.50454551753894E-153

20342<33452;30,3

4.3

The first line contains the contents of the field NUMBER in the standard output format for type F fields. The second line contains the string resulting from the conversion of the field NUMBER to a type C field with length 10 and the formatting

option EXPONENT 2. The third and fourth output lines show that it is not feasible to use target fields which are of numeric data type. The fifth line illustrates that the MOVE statement works differently to WRITE TO. The type F field has been correctly converted [Page 186] to type P.

Specifying the Source Field Dynamically

In the WRITE TO statement, you can specify the name of the source field dynamically as the contents of another field. To do this, specify the name of the field containing the name of the source field in parentheses:

WRITE (<f>) TO <g>.

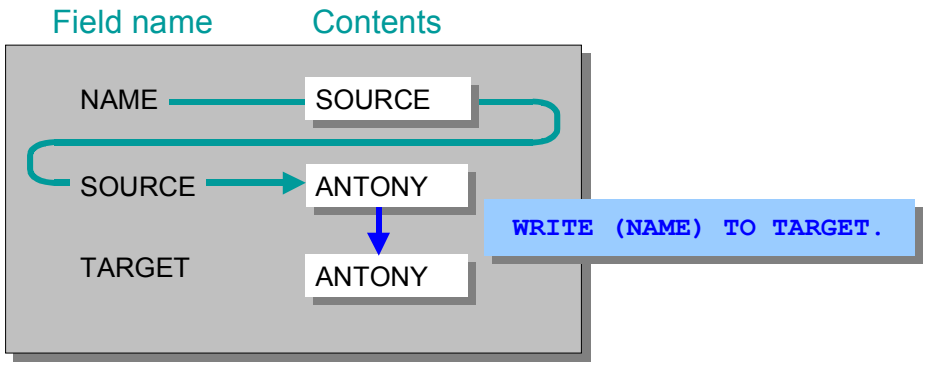
The system places the value of the data object assigned to <f> in <g>. If you want to specify a target field dynamically, you must use a field symbol [Page 201].

```
DATA: NAME(10)   VALUE 'SOURCE',
      SOURCE(10) VALUE 'Antony',
      TARGET(10).

...
WRITE (NAME) TO TARGET.
WRITE: TARGET.
```

This produces the output

Antony

The connection between field names and field contents is shown in the following diagram.



Resetting Values to Their Initial Value

To reset a variable <f> to the appropriate initial value for its type, use the statement

CLEAR <f>.

This statement has different effects for different data types:

- Elementary ABAP types

  The CLEAR statement sets the value of an elementary variable to the initial value specified for its type (see the table in Predefined ABAP Types [Page 97]), and not to the starting value that you specified in the VALUE addition to the DATA statement.

- References

  The CLEAR statement resets a reference variable to its initial value, that is, so that it does not point to an object.

- Structures

  The CLEAR statement resets the individual components of a structure to their respective initial values.

- Internal tables

  The CLEAR statement deletes the entire contents of an internal table (see also Initializing Internal Tables [Page 267]).

You cannot use the CLEAR statement to reset a constant.

DATA NUMBER TYPE I VALUE '10'.

WRITE NUMBER.

CLEAR NUMBER.

WRITE / NUMBER.

The output appears as follows:

```
10
 0
```

The CLEAR statement resets the contents of the field NUMBER from 10 to its initial value 0.

Numerical Operations

To assign the result of a mathematical calculation to a variable, use the COMPUTE statement or the assignment operator (=).

COMPUTE <n> = <expression>.

COMPUTE is optional, so you can also write the statement as follows:

<n> = <expression>.

The result of the mathematical operation specified in <expression> is assigned to the field <n>.

ABAP executes a numerical operation with a numerical precision that corresponds to one of the numerical data types I, P, or F. The numerical precision is defined by the operand with the highest hierarchical level. In this context, the result field <n> and floating point functions [Page 156] (type F) also count as operands. The hierarchical order is F before P before I. Before performing the calculation, ABAP converts [Page 186] all of the operands into the highest-occurring data type. After the operation, the result is assigned to the result field using the same logic as the MOVE statement [Page 145], that is, the system converts the result into the data type of the result field if necessary.

In ABAP, the sequence of type conversions during numerical operations is different from other programming languages. Note, in particular, that the data type of the result field influences the accuracy of the **entire** calculation.

In mathematical expressions, you can combine operations in any permutation and use any number of parentheses.

ABAP interprets mathematical expressions in the following order:

1. Expressions in parentheses
2. Functions
3. ** (powers)
4. *, / , MOD, DIV (multiplication, division)
5. +, - (addition, subtraction)

The mathematical expressions that you can use in <expression> are described in the following sections:

Performing Arithmetic Operations [Page 152]

Mathematical Functions [Page 156]

Business Calculations [Page 158]

Date and Time Calculations [Page 159]

Arithmetic Calculations

Arithmetic calculations use arithmetic operations and special ABAP statements.

Basic Arithmetic Operations

ABAP supports the four basic arithmetic operations and power calculations. You can use the following arithmetic operators in mathematical expressions:

Operator	Meaning
+	Addition
_	Subtraction
*	Multiplication
/	Division
DIV	Integer division
MOD	Remainder of integer division
**	Powers

Instead of using operators in mathematical expressions, you can perform basic arithmetic operations with the keywords ADD, SUBTRACT, MULTIPLY, and DIVIDE.

The following table shows the different ways of expressing basic arithmetic operations in ABAP:

Operation	Statement using mathematical expression	Statement using Keyword
Addition	<p> = <n> + <m>.	ADD <n> TO <m>.
Subtraction	<p> = <m> - <n>.	SUBTRACT <n> FROM <m>.
Multiplication	<p> = <m> * <n>.	MULTIPLY <m> BY <n>.
Division	<p> = <m> / <n>.	DIVIDE <m> BY <n>.
Integer division	<p> = <m> DIV <n>.	---
Remainder of division	<p> = <m> MOD <n>.	---
Powers	<p> = <m> ** <n>.	---

In the statements using keywords, the results of the operations are assigned to field <m>.

The operands <m>, <n>, <p> can be any numeric fields. If the fields are not of the same data type, the system converts all fields into the hierarchically highest data type that occurs in the statement.

When using mathematical expressions, please note that the operators +, -, *, **, and /, as well as opening and closing parentheses, are interpreted as words in ABAP and must therefore be preceded and followed by blanks.

In division operations, the divisor cannot be zero if the dividend is not zero. With integer division, you use the operators DIV or MOD instead of /. DIV calculates the integer quotient, MOD calculates the remainder.

When you combine arithmetic expressions, ABAP performs the calculation from left to right, with one exception: Powers are calculates from right to left. So <n> ** <m> ** <p> is the same as <n> ** (<m> ** <p>), not (<n> ** <m>) ** <p>.

```
DATA: COUNTER TYPE I.

COMPUTE COUNTER = COUNTER + 1.

COUNTER = COUNTER + 1.

ADD 1 TO COUNTER.
```

Here, the three operational statements perform the same arithmetic operation, i.e. adding 1 to the contents of the field COUNTER and assigning the result to COUNTER.

```
DATA: PACK TYPE P DECIMALS 4,
      N TYPE F VALUE '+5.2',
      M TYPE F VALUE '+1.1'.

PACK = N / M.
WRITE PACK.

PACK = N DIV M.
WRITE  / PACK.

PACK = N MOD M.
WRITE /PACK.
```

The output appears as follows:

```
4.7273

4.0000

0.8000
```

This example shows the different types of division.

Arithmetic Calculations Using Structures

In the same way that you can transfer values component by component between structures using the MOVE-CORRESPONDING [Page 145] statement, you can also perform arithmetic operations between the components of structures using the following statements:

- ADD-CORRESPONDING
- SUBTRACT-CORRESPONDING
- MULTIPLY-CORRESPONDING
- DIVIDE-CORRESPONDING

ABAP performs the corresponding calculation for all components that have the same name in both structures. However, it only makes sense to use the operations if all of the components involved have a numeric data type.

```
DATA: BEGIN OF RATE,
        USA TYPE F VALUE '0.6667',
```

```
        FRG TYPE F VALUE '1.0',
        AUT TYPE F VALUE '7.0',
      END OF RATE.

DATA: BEGIN OF MONEY,
        USA TYPE I VALUE 100,
        FRG TYPE I VALUE 200,
        AUT TYPE I VALUE 300,
      END OF MONEY.

MULTIPLY-CORRESPONDING MONEY BY RATE.

WRITE / MONEY-USA.
WRITE / MONEY-FRG.
WRITE / MONEY-AUT.
```

The output appears as follows:

```
     67

    200

  2,100
```

Here, MONEY-USA is multiplied by RATE-USA and so on.

Adding Sequences of Fields

There are variants of the ADD statement that allow you to add sequences of fields in memory. For example:

Adding sequences of fields and assigning the result to another field

ADD <n_1> THEN <n_2> UNTIL <n_z> GIVING <m>.

If <n_1>, <n_2>, ... , <n_z> is a sequence of equidistant fields of the same type and length in memory, they are summed and the result is assigned to <m>.

Adding sequences of fields and adding the result to the contents of another field

ADD <n_1> THEN <n_2> UNTIL <n_z> TO <m>.

This statement is identical to the preceding one, with one difference: The sum of the fields is added to the existing contents of <m>.

For further information about other similar variants, see the keyword documentation for the ADD statement.

```
DATA: BEGIN OF SERIES,
        N1 TYPE I VALUE 10,
        N2 TYPE I VALUE 20,
        N3 TYPE I VALUE 30,
        N4 TYPE I VALUE 40,
        N5 TYPE I VALUE 50,
        N6 TYPE I VALUE 60,
      END OF SERIES.

DATA SUM TYPE I.
```

```
ADD SERIES-N1 THEN SERIES-N2 UNTIL SERIES-N5 GIVING SUM.
WRITE SUM.

ADD SERIES-N2 THEN SERIES-N3 UNTIL SERIES-N6 TO SUM.
WRITE / SUM.
```

Output

```
150

350
```

Here, the contents of components N1 to N5 are summed and assigned to the field SUM. Then, the contents of components N2 to N6 are summed and added to the value of SUM.

Mathematical Functions

ABAP contains a range of built-in functions that you can use as mathematical expressions, or as part of a mathematical expression:

[COMPUTE] <n> = <func>(<m>).

The blanks between the parentheses and the argument <m> are obligatory. The result of calling the function <func> with the argument <m> is assigned to <n>.

Functions for all Numeric Data Types

The following built-in functions work with all three numeric data types (F, I, and P) as arguments.

Functions for all numeric data types

Function	Result
ABS	Absolute value of argument.
SIGN	Sign of argument: $SIGN(X) = \begin{cases} 1 & X > 0 \\ 0 & \text{if } X = 0 \\ -1 & X < 0 \end{cases}$
CEIL	Smallest integer value not smaller than the argument.
FLOOR	Largest integer value not larger than the argument.
TRUNC	Integer part of argument.
FRAC	Fraction part of argument.

The argument of these functions does not have to be a numeric data type. If you choose another type, it is converted to a numeric type. For performance reasons, however, you should use the correct type whenever possible. The functions itself do not have a data type of their own. They do not change the numerical precision of a numerical operation.

```
DATA N TYPE P DECIMALS 2.
DATA M TYPE P DECIMALS 2 VALUE '-5.55'.

N = ABS( M ).   WRITE:   'ABS:  ', N.
N = SIGN( M ).  WRITE: / 'SIGN: ', N.
N = CEIL( M ).  WRITE: / 'CEIL: ', N.
N = FLOOR( M ). WRITE: / 'FLOOR:', N.
N = TRUNC( M ). WRITE: / 'TRUNC:', N.
N = FRAC( M ).  WRITE: / 'FRAC: ', N.
```

The output appears as follows:

```
ABS:          5.55
SIGN:         1.00-
CEIL:         5.00-
FLOOR:        6.00-
TRUNC:        5.00-
FRAC:         0.55-
```

```
DATA: T1(10),
      T2(10) VALUE '-100'.

T1 = ABS( T2 ).

WRITE T1.
```

This produces the following output:

```
          100
```

Two conversions are performed. First, the contents of field T2 (type C) are converted to type P. Then the system processes the ABS function using the results of the conversion. Then, during the assignment to the type C field T1, the result of the function is converted back to type C.

Floating-Point Functions

The following built-in functions work with floating point numbers (data type F) as an argument.

Functions for floating point data types

Function	Meaning
ACOS, ASIN, ATAN; COS, SIN, TAN	Trigonometric functions.
COSH, SINH, TANH	Hyperbolic functions.
EXP	Exponential function with base e (e=2.7182818285).
LOG	Natural logarithm with base e.
LOG10	Logarithm with base 10.
SQRT	Square root.

For all functions, the normal mathematical constraints apply (for example, square root is only possible for positive numbers). If you fail to observe them, a runtime error occurs.

The argument of these functions does not have to be a floating point field. If you choose another type, it is converted [Page 186] to type F. The functions themselves have the data type F. This can change the numerical precision of a numerical operation.

```
DATA: RESULT TYPE F,
      PI(10) VALUE '3.141592654'.

RESULT = COS( PI ).

WRITE RESULT.
```

The output is `-1.00000000000000E+00`. The character field PI is automatically converted to a type F field before the calculation is performed.

Business Calculations

For calculations in business applications, use packed numbers. The program attribute [Page 75] *Fixed point arithmetic* affects calculations using packed numbers.

If the program attribute *Fixed point arithmetic* is not set, type P fields are interpreted as integers without decimal places. The decimal places that you specify in the DECIMALS addition of the TYPES or DATA statement only affect how the field is formatted in the WRITE statement.

```
DATA: PACK TYPE P DECIMALS 2.
PACK = '12345'.
WRITE PACK.
```

If the program attribute Fixed point arithmetic is not set, the output is as follows:

```
123.45
```

If the program attribute *Fixed point arithmetic* is set, the output is as follows:

```
12,345.00
```

If the Fixed point arithmetic attribute is set, the decimal places are also taken into account in arithmetic operations. Calculations with packed numbers in ABAP use the same arithmetic as a pocket calculator. Intermediate results are calculated using up to 31 digits (before and after the decimal point). You should therefore always set the *Fixed point arithmetic* attribute when you use type P fields.

```
DATA: PACK TYPE P.
PACK = 1 / 3 * 3.
WRITE PACK.
```

If you have not set the Fixed point arithmetic attribute, the result is 0, since the calculation is performed using integer accuracy, and the result is therefore rounded internally to 0.
If the program attribute *Fixed point arithmetic* is set, the result is `1` because the result of the division is stored internally as 0.33333333333333333333333333333333 with an accuracy of up to 31 digits.

Date and Time Calculations

Date and time fields have character types, not numeric ones. However, you can still use date and time fields in numeric operations. To allow you to do so, the system performs automatic type conversions [Page 186]. You may also find it useful to use offset and length [Page 196] specifications when using date and time fields in calculations.

Example of a date calculation:

```
DATA: ULTIMO TYPE D.

ULTIMO      = SY-DATUM.
ULTIMO+6(2) = '01'.         " = first day of this month
ULTIMO      = ULTIMO - 1.   " = last day of last month
```

Here, the last day of the previous month is assigned to the date field ULTIMO.

1. ULTIMO is filled with the present date.
2. Using an offset specification, the day is changed to the first day of the current month.
3. 1 is subtracted from ULTIMO. Its contents are changed to the last day of the previous month. Before performing the subtraction, the system converts ULTIMO to the number of days since 01.01.0001 and converts the result back to a date.

Example of a time calculation:

```
DATA: DIFF TYPE I,
      SECONDS TYPE I,
      HOURS TYPE I.

DATA: T1 TYPE T VALUE '200000',
      T2 TYPE T VALUE '020000'.

DIFF = T2 - T1.
SECONDS = DIFF MOD 86400.
HOURS = SECONDS / 3600.
```

The number of hours between 02:00:00 and 20:00:00 is calculated.

1. First, the difference between the time fields is calculated. This is -64800, since T1 and T2 are converted internally into 72000 and 7200 respectively.
2. Second, with the operation MOD, this negative difference is converted to the total number of seconds. A positive difference would be unaffected by this calculation.
3. Third, the number of hours is calculated by dividing the number of seconds by 3600.

   The last three lines can be replaced by the following line

   ```
   HOURS = ( ( T2 - T1 ) MOD 86400 ) / 3600.
   ```

Inverted Dates

In some cases (for example, when sorting dates in descending order), it is useful to convert a date from format D to an inverted date by using the keyword CONVERT.

Date and Time Calculations

CONVERT DATE <d1> INTO INVERTED-DATE <d2>.

Afterwards, you can convert the inverted data back into a normal date using the statement

CONVERT INVERTED-DATE <d1> INTO DATE <d2>.

These statements convert the field <d1> from the formats DATE or INVERTED-DATE to the formats INVERTED-DATE or DATE and assign it to the field <d2>.

For the conversion, ABAP forms the nine's complement.

```
DATA: ODATE TYPE D VALUE '19955011',
      IDATE LIKE ODATE.

DATA  FIELD(8).

FIELD = ODATE.   WRITE / FIELD.

CONVERT DATE ODATE INTO INVERTED-DATE IDATE.

FIELD = IDATE.   WRITE / FIELD.

CONVERT INVERTED-DATE IDATE INTO DATE ODATE.

FIELD = ODATE.   WRITE / FIELD.
```

Output:

```
80049488

19955011

19955011
```

Processing Character Strings

ABAP contains a series of statements for processing character fields (types C, D, N, and T). There is **no** type conversion in these operations. The statements treat all fields as though they were type C fields, regardless of their actual types. The internal form of type D, N, and T fields is the same as for type C fields. For this reason, you can use these statements for all character-type fields.

Shifting Field Contents [Page 162]

Replacing Field Contents [Page 165]

Converting to Upper or Lower Case or Replacing Characters [Page 167]

Converting into a Sortable Format [Page 168]

Overlaying Strings [Page 169]

Finding Strings [Page 170]

Condensing Field Contents [Page 174]

Finding out the Length of a Character String [Page 173]

Concatenating Strings [Page 175]

Splitting Strings [Page 176]

There is also a variant of the MOVE statement that only works with strings:

Assigning Parts of Character Strings [Page 177].

Shifting Field Contents

You can shift the contents of a field using the following variants of the SHIFT statement. SHIFT moves field contents character by character.

Shifting a String by a Given Number of Positions

SHIFT <c> [BY <n> PLACES] [<mode>].

This statement shifts the field <c> by <n> positions. If you omit BY <n> PLACES, <n> is interpreted as one. If <n> is 0 or negative, <c> remains unchanged. If <n> exceeds the length of <c>, <c> is filled out with blanks. <n> can be variable.

With the different <mode> options, you can shift the field <c> in the following ways:

- <mode> is LEFT:

  Shifts the field contents <n> places to the left and adds <n> blanks at the right-hand end of the field (default).

- <mode> is RIGHT:

  Shift <n> positions to the right and adds <n> blanks at the left-hand end of the field.

- <mode> is CIRCULAR:

  Shift <n> positions to the left so that <n> characters on the left appear on the right.

```
DATA: T(10) VALUE 'abcdefghij',
      STRING LIKE T.

STRING = T.
WRITE STRING.

SHIFT STRING.
WRITE / STRING.

STRING = T.
SHIFT STRING BY 3 PLACES LEFT.
WRITE / STRING.

STRING = T.
SHIFT STRING BY 3 PLACES RIGHT.
WRITE / STRING.

STRING = T.
SHIFT STRING BY 3 PLACES CIRCULAR.
WRITE / STRING.
```

Output:

```
abcdefghij

bcdefghij

defghij

   abcdefg

defghijabc
```

Shifting a Structure up to a Given String

SHIFT <c> UP TO <str> <mode>.

This statement searches the field contents of <c> until it finds the string <str> and shifts the field <c> up to the edge of the field. The <mode> options are the same as described above. <str> can be a variable.

If <str> is not found in <c>, SY-SUBRC is set to 4 and <c> is not shifted. Otherwise, SY-SUBRC is set to 0.

```
DATA: T(10) VALUE 'abcdefghij',
      STRING LIKE T,
      STR(2) VALUE 'ef'.

STRING = T.
WRITE STRING.

SHIFT STRING UP TO STR.
WRITE / STRING.

STRING = T.
SHIFT STRING UP TO STR LEFT.
WRITE / STRING.

STRING = T.
SHIFT STRING  UP TO STR RIGHT.
WRITE / STRING.

STRING = T.
SHIFT STRING  UP TO STR CIRCULAR.
WRITE / STRING.
```

Output:

abcdefghij

efghij

efghij

abcdef

efghijabcd

Shifting a Structure According to the First or Last Character

SHIFT <c> LEFT DELETING LEADING <str>.

SHIFT <c> RIGHT DELETING TRAILING <str>.

This statement shifts the field <c> to the left or to the right, provided the first character on the left or the last character on the right occur in <str>. The right or left of the field is then padded with blanks. <str> can be a variable.

```
DATA: T(14) VALUE '    abcdefghij',
      STRING LIKE T,
      STR(6) VALUE 'ghijkl'.
```

```
STRING = T.
WRITE STRING.

SHIFT STRING LEFT DELETING LEADING SPACE.
WRITE / STRING.

STRING = T.
SHIFT STRING RIGHT DELETING TRAILING STR.
WRITE / STRING.
```

Output:

abcdefghij

abcdefghij

abcdef

Replacing Field Contents

To replace a string in a field with a different string, use the REPLACE statement.

REPLACE <str1> WITH <str2> INTO <c> [LENGTH <l>].

The statement searches the field <c> for the first occurrence of the first <l> positions of the pattern <str1>. If no length is specified, it searches for the pattern <str1> in its full length.

Then, the statement replaces the first occurrence of the pattern <str1> in field <c> with the string <str2>. If a length <l> was specified, only the relevant part of the pattern is replaced.

If the return code value of the system field SY-SUBRC is set to 0, this indicates that <str1> was found in <c> and replaced by <str2>. A return code value other than 0 means that nothing was replaced. <str1>, <str2>, and <len> can be variables.

```
DATA: T(10) VALUE 'abcdefghij',
      STRING LIKE T,
      STR1(4) VALUE 'cdef',
      STR2(4) VALUE 'klmn',
      STR3(2) VALUE 'kl',
      STR4(6) VALUE 'klmnop',
      LEN TYPE I VALUE 2.

STRING = T.
WRITE STRING.

REPLACE STR1 WITH STR2 INTO STRING.
WRITE / STRING.

STRING = T.
REPLACE STR1 WITH STR2 INTO STRING LENGTH LEN.
WRITE / STRING.

STRING = T.
REPLACE STR1 WITH STR3 INTO STRING.
WRITE / STRING.

STRING = T.
REPLACE STR1 WITH STR4 INTO STRING.
WRITE / STRING.
```

The output appears as follows:

abcdefghij

abklmnghij

abklmnefgh

abklghij

abklmnopgh

Note how, in the last line, the field STRING is truncated on the right. The search pattern 'cdef' of length 4 is replaced by 'klmnop' of length 6. Then, the rest of the field STRING is filled up to the end of the field.

Converting to Upper or Lower Case or Replacing Characters

The TRANSLATE statement converts characters into upper or lower case, or uses substitution rules to convert all occurrences of one character to another character.

Converting to Upper or Lower Case

TRANSLATE <c> TO UPPER CASE.

TRANSLATE <c> TO LOWER CASE.

These statements convert all lower case letters in the field <c> to upper case or vice versa.

Substituting Characters

TRANSLATE <c> USING <r>.

This statement replaces all characters in field <c> according to the substitution rule stored in field <r>. <r> contains pairs of letters, where the first letter of each pair is replaced by the second letter. <r> can be a variable.

For more variants of the TRANSLATE statement with more complex substitution rules, see the keyword documentation in the ABAP Editor.

```
DATA: T(10) VALUE 'AbCdEfGhIj',
      STRING LIKE T,
      RULE(20) VALUE 'AxbXCydYEzfZ'.

STRING = T.
WRITE STRING.

TRANSLATE STRING TO UPPER CASE.
WRITE / STRING.

STRING = T.
TRANSLATE STRING TO LOWER CASE.
WRITE / STRING.

STRING = T.
TRANSLATE STRING USING RULE.
WRITE / STRING.
```

Output:

AbCdEfGhIj

ABCDEFGHIJ

abcdefghij

xXyYzZGhIj

Converting into a Sortable Format

The CONVERT TEXT statement converts strings into a format that can be sorted alphabetically.

CONVERT TEXT <c> INTO SORTABLE CODE <sc>.

This statement writes a string <c> to a sortable target field <sc>. The field <c> must be of type C and the field <sc> must be of type X with a minimum size of 16 times the size of <c>.

The field <sc> can serve as an alphabetic sort key for <c>. These sorts are applied to internal tables [Page 251] and extract datasets [Page 331]. If you sort unconverted character fields, the system creates an order that corresponds to the platform-specific internal coding of the individual letters. The conversion CONVERT TEXT creates target fields in such a way that, after sorting the target fields, the order of the corresponding character fields is alphabetical.

The method of conversion depends on the text environment of the current ABAP program. The text environment is fixed in the user master record, but can also be set from a program using the following ABAP statement:

SET LOCALE LANGUAGE <lg> [COUNTRY <cy>] [MODIFIER <m>].

This statement sets the text environment according to the language <lg>. With the option COUNTRY, you can specify the country additionally to the language, provided there are country-specific differences for languages. With the option MODIFIER, you can specify another identifier, provided there are differences in the language within one country. For example, in Germany, the order for sorting umlauts is different in a dictionary from the order used in a telephone book.

The fields <lg>, <cy>, and <m> must be of type C and must have the same lengths as the key fields of table TCP0C. Table TCP0C is a table, in which the text environment is maintained platform-dependent. During the statement SET LOCALE, the system sets the text environment according to the entries in TCP0C. With the exception of the platform identity, which is transferred internally, the table key is specified with the SET statement. The platform identifier is passed implicitly. If <lg> equals SPACE, the system sets the text environment according to the user's master record. If there is no entry in the table for the key specified, the system reacts with a runtime error.

The text environment influences **all** operations in ABAP that depend on the character set.

For more information about this topic, see the keyword documentation for CONVERT TEXT and SET LOCALE LANGUAGE.

For an example of alphabetical sorting, see Sorting Internal Tables [Page 271].

Overlaying Character Fields

The OVERLAY statement overlays one string with another:

OVERLAY <c1> WITH <c2> [ONLY <str>].

This statement overlays all positions in field <c1> containing letters which occur in <str> with the contents of <c2>. <c2> remains unchanged. If you omit ONLY <str>, all positions of <c1> containing spaces are overwritten.

If at least one character in <c1> was replaced, SY-SUBRC is set to 0. In all other cases, SY-SUBRC is set to 4. If <c1> is longer than <c2>, it is overlaid only in the length of <c2>.

```
DATA: T(10) VALUE 'a c e g i ',
      STRING LIKE T,
      OVER(10) VALUE 'ABCDEFGHIJ',
      STR(2) VALUE 'ai'.

STRING = T.
WRITE STRING.
WRITE / OVER.

OVERLAY STRING WITH OVER.
WRITE / STRING.

STRING = T.
OVERLAY STRING WITH OVER ONLY STR.
WRITE / STRING.
```

Output:

a c e g i

ABCDEFGHIJ

aBcDeFgHiJ

A c e g I

Finding Character Strings

To search a character field for a particular pattern, use the SEARCH statement as follows:

SEARCH <c> FOR <str> <options>.

The statement searches the field <c> for <str> starting at position <n1>. If successful, the return code value of SY-SUBRC is set to 0 and SY-FDPOS is set to the offset of the string in the field <c>. Otherwise, SY-SUBRC is set to 4.

The search string <str> can have one of the following forms.

<str>	Function
<pattern>	Searches for <pattern> (any sequence of characters). Trailing blanks are ignored.
.<pattern>.	Searches for <pattern>. Trailing blanks are not ignored.
*<pattern>	A word ending with <pattern> is sought.
<pattern>*	Searches for a word starting with <pattern>.

Words are separated by blanks, commas, periods, semicolons, colons, question marks, exclamation marks, parentheses, slashes, plus signs, and equal signs.

<option> in the SEARCH FOR statement can be any of the following:

- ABBREVIATED

  Searches the field <c> for a word containing the string in <str>. The characters can be separated by other characters. The first letter of the word and the string <str> must be the same.

- STARTING AT <n1>

  Searches the field <c> for <str> starting at position <n1>. The result SY-FDPOS refers to the offset relative to <n1> and not to the start of the field.

- ENDING AT <n2>

  Searches the field <c> for <str> up to position <n2>.

- AND MARK

  If the search string is found, all the characters in the search string (and all the characters in between when using ABBREVIATED) are converted to upper case.

```
DATA STRING(30) VALUE 'This is a little sentence.'.

WRITE: / 'Searched', 'SY-SUBRC', 'SY-FDPOS'.
ULINE /1(26).

SEARCH STRING FOR 'X'.
WRITE: / 'X', SY-SUBRC UNDER 'SY-SUBRC',
           SY-FDPOS UNDER 'SY-FDPOS'

SEARCH STRING FOR 'itt   '.
WRITE: / 'itt   ', SY-SUBRC UNDER 'SY-SUBRC',
             SY-FDPOS UNDER 'SY-FDPOS'
```

```
SEARCH STRING FOR '.e .'.
WRITE: / '.e .', SY-SUBRC UNDER 'SY-SUBRC',
           SY-FDPOS UNDER 'SY-FDPOS'.

SEARCH STRING FOR '*e'.
WRITE: / '*e ', SY-SUBRC UNDER 'SY-SUBRC',
          SY-FDPOS UNDER 'SY-FDPOS'.

SEARCH STRING FOR 's*'.
WRITE: / 's* ', SY-SUBRC UNDER 'SY-SUBRC',
          SY-FDPOS UNDER 'SY-FDPOS'.
```

Output:

SEARCHED SY-SUBRC SY-FDPOS

X 4 0

itt 0 11

.e . 0 15

***e 0 10**

s* 0 17

```
DATA: STRING(30) VALUE 'This is a fast first example.',
      POS TYPE I,
      OFF TYPE I.

WRITE / STRING.

SEARCH STRING FOR 'ft' ABBREVIATED.
WRITE: / 'SY-FDPOS:', SY-FDPOS.

POS = SY-FDPOS + 2.
SEARCH STRING FOR 'ft' ABBREVIATED STARTING AT POS AND MARK.
WRITE / STRING.
WRITE: / 'SY-FDPOS:', SY-FDPOS.
OFF = POS + SY-FDPOS -1.
WRITE: / 'Off:', OFF.
```

Output:

This is a fast first example.

SY-FDPOS: 10

This is a fast FIRST example.

SY-FDPOS: 4

Off: 15

Note that in order to find the second word containing 'ft' after finding the word 'fast', you have to add 2 to the offset SY-FDPOS and start the search at the position POS. Otherwise, the word 'fast' would be found again. To obtain the offset of 'first' in relation to the start of the field STRING, it is calculated from POS and SY-FDPOS.

Finding the Length of a Character String

The ABAP function STRLEN returns the length of a string up to the last character that is not a space.

[COMPUTE] <n> = STRLEN(<c>).

STRLEN processes any operand <c> as a character data type, regardless of its real type. There is **no** type conversion.

As with mathematical functions [Page 156], the keyword COMPUTE is optional.

```
DATA: INT TYPE I,
      WORD1(20) VALUE '12345'.
      WORD2(20).
      WORD3(20) VALUE '   4        '.

INT = STRLEN( WORD1 ). WRITE   INT.

INT = STRLEN( WORD2 ). WRITE / INT.

INT = STRLEN( WORD3 ). WRITE / INT.
```

The results are `5`, `0`, and `4` respectively.

Condensing Field Contents

The CONDENSE statement deletes redundant spaces from a string:

CONDENSE <c> [NO-GAPS].

This statement removes any leading blanks in the field <c> and replaces other sequences of blanks by exactly one blank. The result is a left-justified sequence of words, each separated by one blank. If the addition NO-GAPS is specified, all blanks are removed.

```
DATA: STRING(25) VALUE ' one  two   three    four',
      LEN TYPE I.

LEN = STRLEN( STRING ).
WRITE: STRING, '!'.
WRITE: / 'Length: ', LEN.

CONDENSE STRING.
LEN = STRLEN( STRING ).
WRITE: STRING, '!'.
WRITE: / 'Length: ', LEN.

CONDENSE STRING NO-GAPS.
LEN = STRLEN( STRING ).
WRITE: STRING, '!'.
WRITE: / 'Length: ', LEN.
```

Output:

```
one  two   three    four !

Length:                25

one two three four       !

Length:                18

onetwothreefour          !

Length:                15
```

Note that the total length of the field STRING remains unchanged, but that the deleted blanks appear again on the right.

Concatenating Character Strings

The CONCATENATE statement combines two or more separate strings into one.

CONCATENATE <c1> ... <cn> INTO <c> [SEPARATED BY <s>].

This statement concatenates the character fields <c1> to <cn> and assigns the result to <c>. The system ignores spaces at the end of the individual source strings.

The addition SEPARATED BY <s> allows you to specify a character field <s> which is placed in its defined length between the individual fields.

If the result fits into <c>, SY-SUBRC is set to 0. However, if the result has to be truncated, SY-SUBRC is set to 4.

```
DATA: C1(10)  VALUE  'Sum',
      C2(3)   VALUE  'mer',
      C3(5)   VALUE  'holi ',
      C4(10)  VALUE  'day',
      C5(30),
      SEP(3)  VALUE ' - '.

CONCATENATE C1 C2 C3 C4 INTO C5.
WRITE C5.

CONCATENATE C1 C2 C3 C4 INTO C5 SEPARATED BY SEP.

WRITE / C5.
```

Output:

Summerholiday

Sum - mer - holi - day

In C1 to C5, the trailing blanks are ignored. The separator SEP retains them.

Splitting Character Strings

To split a character string into two or more smaller strings, use the SPLIT statement as follows:

SPLIT <c> AT <del> INTO <c1> ... <cn>.

The system searches the field <c> for the separator <del>. The parts before and after the separator are placed in the target fields <c1> ... <cn>.

To place all fragments in different target fields, you must specify enough target fields. Otherwise, the last target field is filled with the rest of the field <c> and still contains delimiters.

If all target fields are long enough and no fragment has to be truncated, SY-SUBRC is set to 0. Otherwise it is set to 4.

```
DATA: STRING(60),
      P1(20) VALUE '++++++++++++++++++++',
      P2(20) VALUE '++++++++++++++++++++',
      P3(20) VALUE '++++++++++++++++++++',
      P4(20) VALUE '++++++++++++++++++++',
      DEL(3) VALUE '***'.

STRING = ' Part 1 *** Part 2 *** Part 3 *** Part 4 *** Part 5'.
WRITE STRING.

SPLIT STRING AT DEL INTO P1 P2 P3 P4.

WRITE / P1.
WRITE / P2.
WRITE / P3.
WRITE / P4.
```

The output appears as follows:

Part 1 * Part 2 *** Part 3 *** Part 4 *** Part 5**

Part 1

Part 2

Part 3

Part 4 * Part 5**

Note that the contents of the fields P1 ...P4 are totally overwritten and that they are filled out with trailing blanks.

You can also split a string into the individual lines of an internal table as follows:

SPLIT <c> AT <del> INTO TABLE <itab>.

The system adds a new line to the internal table <itab> for each part of the string.

Assigning Parts of Character Strings

The following variant of the MOVE statement works only with type C fields:

MOVE <c1> TO <c2> PERCENTAGE <p> [RIGHT].

Copies the percentage <p> percent of the character field <c1> left-justified (or right-justified if specified with the RIGHT option) to <c2>.

The value of <p> can be a number between 0 and 100. The length to be copied from <f1> is rounded up or down to the next whole number.

If one of the arguments in the statement is not type C, the parameter PERCENTAGE is ignored.

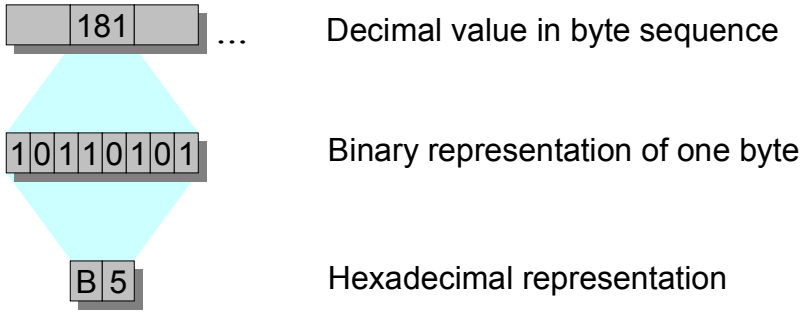
```
DATA C1(10) VALUE 'ABCDEFGHIJ',
    C2(10).

MOVE C1 TO C2 PERCENTAGE 40.

WRITE C2.

MOVE C1 TO C2 PERCENTAGE 40 RIGHT.

WRITE / C2.
```

Output:

```
'ABCD      '
          ABCD
```

Single Bit Processing in Hexadecimal Fields

In a hexadecimal field (type X), you can process the individual bits. ABAP interprets the contents of hex fields byte by byte. A hexadecimal field with length n is n bytes long, and has a display length in ABAP of 2xn. The decimal values 0 - 255 are represented in hexadecimal by the characters ‘00’ to ‘FF’.



The illustration shows how a byte in a sequence with the decimal value 181 is represented in hex. The first four bits have the decimal value 11, the second four bits have the decimal value 5. This leads to the hexadecimal value ‘B5’ - 11x16+5 = 181.

ABAP contains statements that allow you to read and set the individual bits in a type X field. There are also special logical operators that you can use to compare bit sequences [Page 233].

Bit sequence processing allows you to process complex conditions and set operations more efficiently.

Setting and Reading Bits [Page 179]

Bit Operations [Page 181]

Set Operations Using Bit Sequences [Page 183]

Setting and Reading Bits

In a hexadecimal field (type X), you can set or read individual bits.

Setting Bits

To set an individual bit, use the statement

SET BIT <n> OF <f> [TO <b>].

This statement sets the bit at position <n> of field <f> to 1 (or to the value of field <b>). The system must be able to interpret field <n> as a positive integer. The field <f> must have data type X. The field <b> must contain the value 0 or 1. If the bit is set, SY-SUBRC is set to 0. If <n> is greater than the length of <f>, SY-SUBRC is unequal to zero. If <n> or <b> contain invalid values, a runtime error occurs.

```
DATA HEX(3) TYPE X.

SET BIT: 09 OF HEX TO 1,
         10 OF HEX TO 0,
         11 OF HEX TO 1,
         12 OF HEX TO 1,
         13 OF HEX TO 0,
         14 OF HEX TO 1,
         15 OF HEX TO 0,
         16 OF HEX TO 1.

WRITE HEX.
```

The bits of the second byte in the three-character hexadecimal field HEX are set to '10110101', and the list output is as follows:

```
00B500
```

The decimal value of the second byte is 181.

Reading Bits

To read an individual bit, use the statement

GET BIT <n> OF <f> INTO <b>.

This statement reads the bit at position <n> of field <f> into field <b>. The system must be able to interpret field <n> as a positive integer. The field <f> must have data type X. If the bit is read, SY-SUBRC is set to 0. If <n> is greater than the length of <f>, SY-SUBRC is unequal to zero and <b> is set to zero. If <n> contains an invalid value, a runtime error occurs.

```
DATA: HEX(1) TYPE X VALUE 'B5',
      B(1) TYPE N.

DO 8 TIMES.
  GET BIT SY-INDEX OF HEX INTO B.
  WRITE B NO-GAP.
ENDDO.
```

Here, the eight buts of the single-character hexadecimal field HEX (value ‘B5’) are read and displayed as follows:

```
10110101
```

Bit Operations

Bit operations are performed similarly to numeric operations [Page 151] . You can either use the statement

COMPUTE <x> = <bitexp>.

or omit the keyword COMPUTE, as in the statement

<x> = <bitexp>.

<bitexp> can be one of the following bit expressions:

Bit expression	Meaning
BIT-NOT <y>	Negation
<y> BIT-AND <z>	And
<y> BIT-XOR <z>	Exclusive or
<y> BIT-OR <z>	Or

The operands <y> and <z> are linked bit by bit in the above operators, and the result is placed in the result field <x>. The result field <x>, and the operands <y> and <z> must all be of type X. If the field lengths are different, all operands are converted [Page 187] to the field length of the result field <x>.

The following rules apply to bit operations:

<y>	<z>	BIT-NOT <y>	<y> BIT-AND <z>	<y> BIT-XOR <z>	<y> BIT-XOR <z>
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	0	1

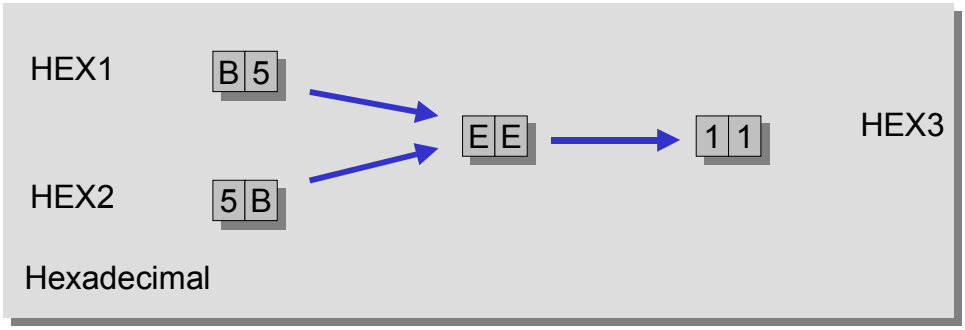
As with mathematical expressions, you can use parentheses with bit expressions.

```
DATA: HEX1(1) TYPE X VALUE 'B5',
      HEX2(1) TYPE X VALUE '5B',
      HEX3(1) TYPE X.

HEX3 = BIT-NOT ( HEX1 BIT-XOR HEX2 ).

WRITE HEX3.
```

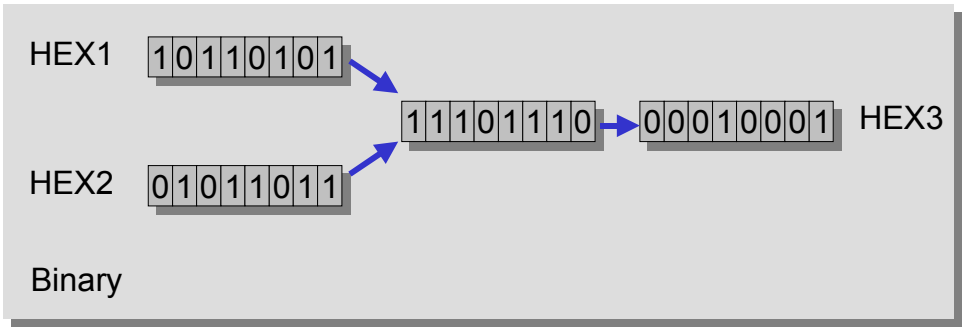
The output is:

```
11
```

The bit operation is processed as displayed below:



BIT-XOR **BIT-NOT**



Set Operations Using Bit Sequences

If you have a set of m elements, you can represent any subset as a sequence of bits. If the n^{th} element of the set is present in a subset, the n^{th} bit is set to 1, otherwise, it is set to 0. The universal set therefore contains no zeros.

In ABAP, you can represent a set of m elements using a field with type X and length of at least m/8. To include a member of the universal set in a set, use the SET BIT statement. To check which members of the universal set are included in a particular set, use the GET BIT statement. You can use bit operations for the following set operations:

Set operation	Bit operation
Intersection	BIT-AND
Union	BIT-OR
Symmetrical difference	BIT-XOR

You can also use the O operator when comparing bit sequences [Page 233] to determine whether a particular set is a subset of another.

```
DATA: FRANKFURT(4) TYPE X,
      FRISCO(4)    TYPE X,
      INTERSECT(4) TYPE X,
      UNION(4)     TYPE X,
      BIT          TYPE I.

DATA: CARRID TYPE SPFLI-CARRID,
      CARRIER LIKE SORTED TABLE OF CARRID
                          WITH UNIQUE KEY TABLE LINE.

DATA WA TYPE SPFLI.

SELECT CARRID FROM SCARR INTO TABLE CARRIER.

SELECT CARRID CITYFROM FROM SPFLI
                       INTO CORRESPONDING FIELDS OF WA.

  WRITE: / WA-CARRID, WA-CITYFROM.

  READ TABLE CARRIER FROM WA-CARRID TRANSPORTING NO FIELDS.

  CASE WA-CITYFROM.
    WHEN 'FRANKFURT'.
      SET BIT SY-TABIX OF FRANKFURT.
    WHEN 'SAN FRANCISCO'.
      SET BIT SY-TABIX OF FRISCO.
  ENDCASE.

ENDSELECT.

INTERSECT = FRANKFURT BIT-AND FRISCO.
UNION     = FRANKFURT BIT-OR  FRISCO.

SKIP.
```

Set Operations Using Bit Sequences

```
WRITE 'Airlines flying from Frankfurt and San Francisco:'.
DO 32 TIMES.
  GET BIT SY-INDEX OF INTERSECT INTO BIT.
    IF BIT = 1.
      READ TABLE CARRIER INDEX SY-INDEX INTO CARRID.
      WRITE CARRID.
    ENDIF.
ENDDO.

SKIP.

WRITE 'Airlines flying from Frankfurt or San Francisco:'.
DO 32 TIMES.
  GET BIT SY-INDEX OF UNION INTO BIT.
    IF BIT = 1.
      READ TABLE CARRIER INDEX SY-INDEX INTO CARRID.
      WRITE CARRID.
    ENDIF.
ENDDO.
```

This produces the following output list:

```
AA  NEW YORK
AA  SAN FRANCISCO
AZ  ROME
AZ  ROME
AZ  TOKYO
AZ  ROME
DL  NEW YORK
DL  SAN FRANCISCO
LH  FRANKFURT
LH  FRANKFURT
LH  FRANKFURT
LH  BERLIN
QF  SINGAPORE
QF  FRANKFURT
SQ  SINGAPORE
SQ  SINGAPORE
SQ  SINGAPORE
SQ  SINGAPORE
UA  FRANKFURT
UA  SAN FRANCISCO

Airlines flying from Frankfurt and San Francisco: UA

Airlines flying from Frankfurt or San Francisco: AA  DL  LH  QF  UA
```

The program uses four hexadecimal fields with length 4 - FRANKFURT, FRISCO, INTERSECT, and UNION. Each of these fields can represent a set of up to 32 elements. The basic set is the set of all airlines from database table SCARR. Each bit of the corresponding bit sequences representes one airline. To provide an index, the external index table CARRIER is created and filled with the airline codes from table SCARR. It is then possible to identify an airline using the internal index of table CARRIER.

In the SELECT loop for database table SPFLI, the corresponding bit for the airline is set either in the FRANKFURT field or the FRISCO field, depending on the departure city. The line number SY-TABIX is determined using a READ statement in which no fields are transported.

The intersection and union of FRANKFURT and FRISCO are constructed using the bit operations BIT-AND and BIT-OR.

The bits in INTERSECT and UNION are read one by one and evaluated in two DO loops. For each position in the fields with the value 1, a READ statement retrieves the airline code from the table CARRIER.

Type Conversions

Every time you assign a data object to a variable, the data types involved must either be compatible [Page 133] , that is, their technical attributes (data type, field length, number of decimal places) must be identical, or the data type of the source field must be convertible into the data type of the target field.

In ABAP, two non-compatible data types can be converted to each other if a corresponding conversion rule exists. If data types are compatible, no conversion rule is necessary.

If you use the MOVE statement to transfer values between non-compatible objects, the value of the source object is always converted into the data type of the target object. With all ABAP operations that perform value assignments between data objects (for example, arithmetic operations or filling internal tables), the system handles all the necessary type conversions as for the MOVE statement. If you try to assign values between two data types for which no conversion rule exists, a syntax error or runtime error occurs.

The following sections contain the conversion rules for incompatible ABAP data types:

Conversion Rules for Elementary Data Types [Page 187]

Conversion Rules for References [Page 191]

Conversion Rules for Structures [Page 192]

Conversion Rules for Internal Tables [Page 194]

With some ABAP statements that pass data between different objects, the alignment of the data objects is also important.

Alignment of Data Objects [Page 195]

Conversion Rules for Elementary Data Types

There are eight predefined ABAP data types [Page 97]. There are 64 possible type combinations between these elementary data types. ABAP supports automatic type conversion and length adjustment for all of them except type D (date) and type T (time) fields which cannot be converted into each other.

The following conversion tables define the rules for converting elementary data types for all possible combinations of source and target fields.

Source Type Character

Conversion table for source type C

Target	Conversion
C	The target field is filled from left to right. If it is too long, it is filled with blanks from the right. If it is too short, the contents are truncated at the right-hand end.
D	The character field should contain an 8-character date in the format YYYYMMDD .
F	The contents of the source field must be a valid representation of a type F field as described in Literals [Page 119] .
N	Only the digits in the source field are copied. The field is right-justified and filled with trailing zeros.
I P	The source field must contain the representation of a decimal number, that is, a sequence of digits with an optional sign and no more than one decimal point. The source field can contain blanks. If the target field is too short, an overflow may occur. This may cause the system to terminate the program.
T	The character field should contain a 6-character time in HHMMSS format.
X	Since the character field should contain a hexadecimal-character string, the only valid characters are 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. This character string is packed as a hexadecimal number, transported left-justified, and padded with zeros or truncated on the right.

Source Type Date

Conversion table for source type D

Target	Conversion
C	The date is transported left-justified without conversion.
D	Transport without conversion.
F	The date is converted into a packed number. The packed number is then converted into a floating point number (see corresponding table).
N	The date is converted into a character field. The character field is then converted into a numeric text field (see corresponding table).
I P	The date is converted to the number of days since 01.01.0001.
T	Not supported. Results in an error message during the syntax check or in a runtime error.
X	The date is converted to the number of days since 01.01.0001 in hexadecimal format.

Source Type Floating Point Number

Conversion table for source type F

Target	Conversion
C	The floating point number is converted to the <mantissa>E<exponent> format and transported to the character field. The value of the mantissa lies between 1 and 10 unless the number is zero. The exponent is always signed. If the target field is too short, the mantissa is rounded. The length of the character field should be at least 6 bytes.
D	The source field is converted into a packed number. The packed number is then converted into a date field (see corresponding table).
F	Transport without conversion.
N	The source field is converted into a packed number. The packed number is then converted into a numeric text field (see corresponding table).
I P	The floating point number is converted to an integer or fixed point value and, if necessary, rounded.
T	The source field is converted into a packed number. The packed number is then converted into a time field (see corresponding table).
X	The source field is converted into a packed number. The packed number is then converted into a hexadecimal number (see corresponding table).

Source Type Integer

Type I is always treated in the same way as type P without decimal places. Wherever type P is mentioned, the same applies to type I fields.

Source Type Numeric Text

Conversion table for source type N

Target	Conversion
C	The numeric field is treated like a character field. Leading zeros are retained.
D	The numeric field is converted into a character field. The character field is then converted into a date field (see corresponding table).
F	The numeric field is converted into a packed number. The packed number is then converted into a floating point number (see corresponding table).
N	The numeric field is transported right-justified and padded with zeros or truncated on the left.
I P	The numeric field is interpreted as a number, and transferred to the target field, where it is right-justified, and adopts a plus sign. If the target field is too short, the program may be terminated.
T	The numeric field is converted into a character field. The character field is then converted into a time field (see corresponding table).
X	The numeric field is converted into a packed number. The packed number is then converted into a hexadecimal number (see corresponding table).

Source Type Packed Number

If the program attribute *Fixed point arithmetic* is set, the system rounds type P fields according to the number of decimal places or fills them out with zeros.

Conversion table for source type P

Target	Conversion
C	The packed field is transported right-justified to the character field, if required with a decimal point. The first position is reserved for the sign. Leading zeros appear as blanks. If the target field is too short, the sign is omitted for positive numbers. If this is still not sufficient, the field is truncated on the left. ABAP indicates the truncation with an asterisk (*). If you want the leading zeros to appear in the character field, use UNPACK instead of MOVE.
D	The packed field value represents the number of days since 01.01.0001 and is converted to a date in YYYYMMDD format.
F	The packed field is accepted and transported as a floating point number.
N	The packed field is rounded if necessary, unpacked, and then transported right-justified. The sign is omitted. If required, the target field is padded with zeros on the left.
I P	The packed field is transported right-justified. If the target field is too short, an overflow occurs.
T	The packed field value represents the number of seconds since midnight and is converted to a time in HHMMSS format.
X	The packed field is rounded if necessary and then converted to a hexadecimal number. Negative numbers are represented by the two's complement. If the target field is too short, the number is truncated on the left.

Source Type Time

Conversion table for source type T

Target	Conversion
C	The source field is transported left-justified without conversion.
D	Not supported. Results in an error message during the syntax check or in a runtime error.
F	The source field is converted into a packed number. The packed number is then converted into a floating point number (see corresponding table).
N	The date is converted into a character field. The character field is then converted into a numeric text field (see corresponding table).
I P	The date is converted to the number of seconds since midnight.
T	Transport without conversion.
X	The date is converted to the number of seconds since midnight in hexadecimal format.

Source Type Hexadecimal

Conversion table for source type X

Conversion Rules for Elementary Data Types

Target	Conversion
C	The value in the hexadecimal field is converted to a hexadecimal character string, transported left-justified to the target field, and padded with zeros.
D	The source field value represents the number of days since 01.01.0001 and is converted to a date in YYYYMMDD format.
F	The source field is converted into a packed number. The packed number is then converted into a floating point number (see corresponding table).
N	The source field is converted into a packed number. The packed number is then converted into a numeric text field (see corresponding table).
I P	The value of the source field is interpreted as a hexadecimal number. It is converted to a packed decimal number and transported right-justified to the target field. If the hexadecimal field is longer than 4 bytes, only the last four bytes are converted. If it is too short, a runtime error may occur.
T	The source field value represents the number of seconds since midnight and is converted to a time in HHMMSS format.
X	The value is transported left-justified and filled with X'00' on the right, if necessary.

Conversion Rules for References

ABAP currently uses class and interface variables within ABAP Objects [Page 1291]. Both are pointers to objects. You can assign values to them in the following combinations:

- If the two class references are incompatible, the class of the target field must be the predefined empty class OBJECT.
- When you assign a class reference to an interface reference, the class of the source field must implement the interface of the target field.
- If two interface references are incompatible, the interface of the target field must contain the interface of the source field as a component.
- When you assign an interface reference to a class reference, the class of the source field must be the predefined empty class OBJECT.

Conversion Rules for Structures

ABAP has one rule for converting structures that **do not** contain internal tables as components. There are no conversion rules for structures that contain internal tables. You can only make assignments between structures that are compatible.

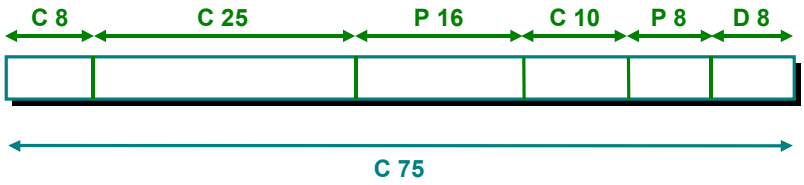
You can combine convertible structures in the following combinations:

- Converting a structure into a non-compatible structure
- Converting elementary fields into structures
- Converting structures into elementary fields

In each case, the system first converts all the structures concerned to type C fields and then performs the conversion between the two resulting elementary fields. The length of the type C fields is the sum of the lengths of the structure components. This rule applies to all operations using structures that **do not contain internal tables**.

If a structure is aligned [Page 195], the filler fields are also added to the length of the type C field.

A non-aligned structure without filler fields:



If you convert a structure into a shorter structure, the original structure is truncated. If you convert a structure into a longer one, the parts at the end are not initialized according to their type, but filled with blanks.

It can make sense to assign a structure to another, incompatible, structure if, for example, the target structure is shorter than the source, and both structures have the same construction over the length of the shorter structure. However, numeric components of structures that are filled in incompatible assignments may contain nonsensical or invalid values that may cause runtime errors.

```
DATA: BEGIN OF FS1,
        INT     TYPE I          VALUE 5,
        PACK    TYPE P DECIMALS 2 VALUE ‘2.26’,
        TEXT(10) TYPE C          VALUE ‘Fine text’,
        FLOAT   TYPE F          VALUE ‘1.234e+05’,
        DATA    TYPE D          VALUE ‘19950916’,
      END OF FS1.

DATA: BEGIN OF FS2,
        INT     TYPE I          VALUE 3,
        PACK    TYPE P DECIMALS 2 VALUE ‘72.34’,
```

```
      TEXT(5)  TYPE C          VALUE 'Hello',
    END OF FS2.

WRITE: / FS1-INT, FS1-PACK; FS1-TEXT, FS1-FLOAT, FS1-DATE.
WRITE: / FS2-INT, FS2-PACK, FS2-TEXT.

MOVE FS1 TO FS2.
WRITE: / FS2-INT, FS2-PACK, FS2-TEXT.
```

Output:

```
        5          2.26 Fine text   1.234000000000000E+05
09161995

        3         72.34 Hello

        5          2.26 Fine
```

This example defines two different structures, FS1 and FS2. In each one, the first two components have the same data type. After assigning FS1 to FS2, only the result for the first two components is as if they had been moved component-by-component. FS2-TEXT is filled with the first five characters of FS1-TEXT. All other positions of FS1 are omitted.

Conversion Rules for Internal Tables

Internal tables can only be converted into other internal tables. You cannot convert them into structures or elementary fields.

Internal tables are convertible if their line types are convertible. The convertibility of internal tables does not depend on the number of lines.

Conversion rules for internal tables:

- Internal tables which have internal tables as their line type are convertible if the internal tables which define the line types are convertible.
- Internal tables which have line types that are structures with internal tables as components are convertible according to the conversion rules for structures [Page 192] if the structures are **compatible**.

Alignment of Data Objects

Elementary fields with types I and F occupy special memory addresses that are platform-specific. For example, the address of a type I field must be divisible by 4, and the address of a type F field by 8. Consequently, type I and F fields are known as **aligned** fields. Structures containing fields with type I or F are also aligned, and may contain filler fields immediately before their aligned components.

The system normally aligns fields and structures automatically when you declare them.

You must take alignment into account in the following cases:

- When you pass elementary fields or structures to a procedure [Page 449] as actual parameters where the corresponding formal parameter is not typed accordingly.
- When you declare field symbols [Page 203]
- When you use a work area with an ABAP Open SQL statement that does not have the same type as the database table as defined in the ABAP Dictionary [Page 105].
- When you process components [Page 196] of structures.

Processing Sections of Strings

You can address a section of a string in any statement in which **non-numeric** elementary ABAP types or structures that do not contain internal tables occur using the following syntax:

<f>[+<o>][(<l>)]

By specifying an offset +<o> and a length (<l>) directly after the field name <f>, you can address the part of the field starting at position <o>+1 with length <l> as though it were an independent data object. The data type and length of the string section are as follows:

Original field	Section	
Data type	**Data type**	**Length**
C	C	<l>
D	N	<l>
N	N	<l>
T	N	<l>
X	X	<l>
Structure	C	<l>

If you do not specify the length <l>, you address the section of the field from <o> to the end of the field. If the offset and length combination that you specify leads to an invalid section of the field (for example, exceeding the length of the original field), a syntax or runtime error occurs. You cannot use offset and length to address a literal or a text symbol.

You should take particular care when addressing components of structures. To ensure the correct platform-specific alignment of type I and F components, they may contain filler fields, whose lengths you need to consider when calculating the correct offset. Furthermore, SAP plans to convert the internal representation of character types to UNICODE, in which one character will no longer occupy one byte, but instead two or four. Although offset and length specifications can work for character fields (types C, D, N, and T) or for hexadecimal fields (type X), incompatible changes may occur in structures containing a **mixture** of numeric, character, and hexadecimal fields. SAP therefore recommends that you **do not** use offset and length to address components of structures.

In nearly all cases, you must specify offset <o> and length <l> as numeric literals without a preceding sign. You may specify them dynamically in the following cases:

- When assigning values using MOVE or the assignment operator
- When assigning values with WRITE TO
- When assigning field symbols using ASSIGN.
- When passing actual parameters to subroutines in the PERFORM statement.

```
DATA TIME TYPE T VALUE '172545'.

WRITE TIME.
WRITE / TIME+2(2).
```

```
CLEAR TIME+2(4).
WRITE / TIME.
```

The output appears as follows:

172545

25

170000

First, the minutes are selected by specifying an offset in the WRITE statement. Then, the minutes and seconds are set to their initial values by specifying an offset in the clear statement.

Offset and Length Specifications in the MOVE Statement

For the MOVE statement, the syntax for specifying offset and length is as follows:

MOVE <f1>[+<o1>][(<l1>)] TO <f2>[+<o2>][(<l2>)].

Or, when you use the assignment operator:

<f2>[+<o2>][(<l2>)] = <f1>[+<o1>][(<l1>)].

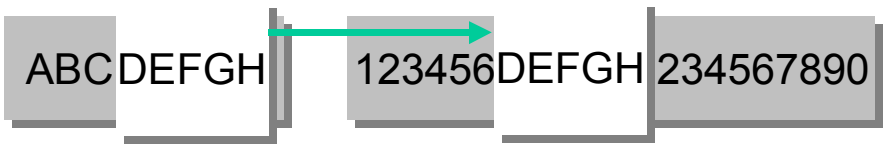
The contents of the part of the field <f1> which begins at position <o1>+1 and has a length of <l1> are assigned to field <f2>, where they overwrite the section which begins at position <o2>+1 and has a length of <l2>.

In the MOVE statement, all offset and length specifications can be variables. This also applies to statements with the assignment operator, as long as these can also be written as MOVE statements. In statements where no field name is specified after the assignment operator, (for example, in Numeric Operations), all offset and length specifications must be unsigned number literals.

SAP recommends that you assign values with offset and length specifications only between non-numeric fields. With numeric fields, the results can be meaningless.

```
DATA: F1(8)  VALUE 'ABCDEFGH',
      F2(20) VALUE '12345678901234567890'.

F2+6(5) = F1+3(5).
```

In this example, the assignment operator functions as follows:



```
DATA: F1(8) VALUE 'ABCDEFGH',
      F2(8).

DATA: O TYPE I VALUE 2,
      L TYPE I VALUE 4.

MOVE F1      TO F2.      WRITE   F2.
MOVE F1+O(L) TO F2.      WRITE / F2.
MOVE F1      TO F2+O(L). WRITE / F2.

CLEAR F2.
MOVE F1      TO F2+O(L). WRITE / F2.
MOVE F1+O(L) TO F2+O(L). WRITE / F2.
```

This produces the following output:

```
ABCDEFGH

CDEF

CDABCD

  ABCD

  CDEF
```

First, the contents of F1 are assigned to F2 without offset specifications. Then, the same happens for F1 with offset and length specification. The next three MOVE statements overwrite the contents of `F2` with offset 2. Note that `F2` is filled with spaces on the right, in accordance with the conversion rule [Page 187] for source type C.

Offset and Length Specifications in the WRITE TO Statement

For the WRITE TO statement, the syntax for specifying offset and length is as follows:

WRITE <f1>[+<o1>][(<l1>)] TO <f2>[+<o2>][(<l2>)].

The contents of the part of the field <f1> which begins at position <o1>+1 and has a length of <l1> are converted to a character field and assigned to field <f2>, where they overwrite the section which begins at position <o2>+1 and has a length of <l2>.

In the WRITE TO statement, the offset and length specifications of the **target field** can be variables. The offset and length of the target field must be numeric literals without a preceding sign.

```
DATA: STRING(20),
      NUMBER(8) TYPE C VALUE '123456',
      OFFSET TYPE I VALUE 8,
      LENGTH TYPE I VALUE 12.

WRITE NUMBER(6) TO STRING+OFFSET(LENGTH) LEFT-JUSTIFIED.
WRITE: / STRING.
CLEAR STRING.
```

```
WRITE NUMBER(6) TO STRING+OFFSET(LENGTH) CENTERED.
WRITE: / STRING.
CLEAR STRING.

WRITE NUMBER TO STRING+OFFSET(LENGTH) RIGHT-JUSTIFIED.
WRITE: / STRING.
CLEAR STRING.
```

This produces the following output:

123456

123456

123456

The first six characters of the field NUMBER are written left-justified, centered, and right-justified into the last 12 characters of the field STRING.

Field Symbols and Data References

Field symbols and data references allow you to access data objects dynamically in ABAP programs. Unlike static access to a data object, where you need to specify the name of the object, field symbols and data references allow you to access and pass data objects whose name and attributes you do not know until runtime.

You can regard a field symbol as a symbolic name for a data object. Field symbols use **value semantics**. When you address a field symbol, the system works with the contents of the data object assigned to it, and not with the contents of the field symbol itself.

Data references are pointers to data objects. Data references use **reference semantics**. Data references are the contents of data reference variables. When you access the reference variable, you are addressing the data reference. To access the contents of the data object to which a data reference is pointing, you must dereference it.

Field Symbols [Page 201]

Data References [Page 219]

Field Symbols

Datenreferenzen [Page 219]

Field symbols are placeholders or symbolic names for other fields. They do not physically reserve space for a field, but point to its contents. A field symbol cam point to any data object. The data object to which a field symbol points is assigned to it after it has been declared in the program.

Whenever you address a field symbol in a program, you are addressing the field that is assigned to the field symbol. After successful assignment, there is no difference in ABAP whether you reference the field symbol or the field itself. You must assign a field to a field symbol before you can address it in a program.

Field symbols are similar to dereferenced pointers in C (that is, pointers to which the content operator * is applied). However, the only real equivalent of pointers in ABAP, that is, variables that contain a memory address (reference) and that can be used without the contents operator, are reference variables in ABAP Objects.

All operations programmed with field symbols are applied to the field assigned to it. For example, a MOVE statement between two field symbols moves the contents of the field assigned to the first field symbol to the field assigned to the second field symbol. The field symbols themselves point to the same fields after the MOVE statement as they did before.

You can create field symbols either without or with type specifications. If you do not specify a type, the field symbol inherits all of the technical attributes of the field assigned to it. If you do specify a type, the system checks the compatibility of the field symbol and the field you are assigning to it during the ASSIGN statement.

Field symbols provide greater flexibility when you address data objects:

- If you want to process sections of fields, you can specify the offset and length of the field dynamically.
- You can assign one field symbol to another, which allows you to address parts of fields.
- Assignments to field symbols may extend beyond field boundaries. This allows you to address regular sequences of fields in memory efficiently.
- You can also force a field symbol to take different technical attributes from those of the field assigned to it.

The flexibility of field symbols provides elegant solutions to certain problems. On the other hand, it does mean that errors can easily occur. Since fields are not assigned to field symbols until runtime, the effectiveness of syntax and security checks is very limited for operations involving field symbols. This can lead to runtime errors or incorrect data assignments.

While runtime errors indicate an obvious problem, incorrect data assignments are dangerous because they can be very difficult to detect. For this reason, you should only use field symbols if you cannot achieve the same result using other ABAP statements.

For example, you may want to process part of a string where the offset and length depend on the contents of the field. You could use field symbols in this case. However, since the MOVE statement also supports variable offset and length specifications, you should use it instead. The MOVE statement (with your own auxiliary variables if required) is much safer than using field symbols, since it cannot address memory beyond the boundary of a field. However, field symbols may improve performance in some cases.

Field Symbols

Defining Field Symbols

To declare a field symbol, use the statement

FIELD-SYMBOLS <FS> [<type>|STRUCTURE <s> DEFAULT <wa>].

For field symbols, the angle brackets are part of the syntax. They identify field symbols in the program code.

If you do not specify any additions, the field symbol <FS> can have data objects of any time assigned to it. When you assign a data object, the field symbol inherits its technical attributes. The data type of the assigned data object becomes the actual data type of the field symbol.

Note that although it possible to assign reference variables and structured data objects to **untyped** field symbols, the static field symbol is only a pointer to the field in memory, and does not have the complex type attributes of a reference or structured field until runtime. You can only use the field symbol to address the whole field (for example, in a MOVE statement). Specific statements such as CREATE OBJECT <FS> or LOOP AT <FS> are not possible.

Typing Field Symbols

The <type> addition allows you to specify the type of a field symbol. When you assign a data object to a field symbol, the system checks whether the type of the data object you are trying to assign is compatible with that of the field symbol. If the types are not compatible or convertible, the system reacts with a syntax or runtime error. You can only assign variables to a field symbol if their type is compatible with that of the field symbol (see Defining the Type of a Field Symbol [Page 215]). In this case, the field symbol retains its original type, regardless of the type of the data object assigned to it.

You specify the type of a field symbol using the same semantics as for formal parameters in procedures [Page 449]. For <type> you can enter either TYPE <t> or LIKE <f>. You can specify the type either generically or in full. If you specify a generic type, the type of the field symbol is either partially specified or not specified at all. Any attributes that are not specified are inherited from the corresponding data object in the ASSIGN statement. If you specify the type fully, all of the technical attributes of the field symbol are determined when you define it. You can then only assign data objects to it that have exactly the same data type.

Generic Type Specification

The following types allow you more freedom when using actual parameters. The data object only needs to have the selection of attributes specified.

Type specification	Check for data object
No type specification TYPE ANY	All types of data object are accepted. The field symbol adopts all of the attributes of the data object.
TYPE C, N, P, or X	Only data objects with type C, N, P, or X are accepted. The field symbol adopts the field length and DECIMALS specification (type P) of the data object.
TYPE TABLE	The system checks whether the data object is a standard internal table. This is a shortened form of TYPE STANDARD TABLE (see below).
TYPE ANY TABLE	The system checks whether the data object is an internal table. The field symbol inherits all of the attributes (line type, table type, key) from the data object.

Defining Field Symbols

TYPE INDEX TABLE	The system checks whether the data object is an index table (standard or sorted table). The field symbol inherits all of the attributes (line type, table type, key) from the data object.
TYPE STANDARD TABLE	The system checks whether the data object is a standard internal table. The field symbol inherits all of the remaining attributes (line type, key) from the data object.
TYPE SORTED TABLE	The system checks whether the actual parameter is a sorted internal table. The field symbol inherits all of the remaining attributes (line type, key) from the data object.
TYPE HASHED TABLE	The system checks whether the actual parameter is a hashed internal table. The field symbol inherits all of the remaining attributes (line type, key) from the data object.

If you specify a type generically, remember that the attributes inherited by the field symbol from the program are not statically recognizable in the program. You can, at most, address them **dynamically**.

```
TYPES: BEGIN OF LINE,
         COL1,
         COL2,
       END OF LINE.

DATA: WA TYPE LINE,
      ITAB TYPE HASHED TABLE OF LINE WITH UNIQUE KEY COL1,
      KEY(4) VALUE 'COL1'.

FIELD-SYMBOLS <FS> TYPE ANY TABLE.

ASSIGN ITAB TO <FS>.

READ TABLE <FS> WITH TABLE KEY (KEY) = 'X' INTO WA.
```

The internal table ITAB is assigned to the generic field symbol <FS>, after which it is possible to address the table key of the field symbol dynamically. However, the static address

```
READ TABLE <FS> WITH TABLE KEY COL1 = 'X' INTO WA.
```

is syntactically not possible, since the formal parameter P does not adopt the key of table ITAB until runtime. In the program, the type specification ANY TABLE only indicates that <FS> is a table. If the type had been ANY (or no type had been specified at all), even the specific internal table statement READ TABLE <FS> would not have been possible.

If you adopt a structured type generically (a structure, or a table with structured line type), the individual components cannot be addressed in the program either statically or dynamically. In this case, you would have to work with further field symbols and the method of assigning structures component by component [Page 213].

Specifying the Type Fully

When you use the following types, the technical attributes of the field symbols are fully specified. The technical attributes of the data objects must correspond to those of the field symbol.

Type specification	Technical attributes of the field symbol

TYPE D, F, I, or T	The field symbol has the technical attributes of the predefined elementary type
TYPE <type>	The field symbol has the type <type>. This is a data type defined within the program using the TYPES statement, or a type from the ABAP Dictionary
TYPE REF TO <cif>	The field symbol is a reference variable (ABAP Objects) for the class or interface <cif>
TYPE LINE OF <itab>	The field symbol has the same type as a line of the internal table <itab> defined using a TYPES statement or defined in the ABAP Dictionary
LIKE <f>	The field symbol has the same type as an internal data object <f> or structure, or a database table from the ABAP Dictionary

When you use a field symbol that is fully typed, you can address its attributes statically in the program, since they are recognized in the source code. If you fully specify the type of a field symbol as a reference or structured data object, you can address it as you would the data object itself once you have assigned an object to it. So, for example, you could address the components of a structure, loop through an internal table, or create an object with reference to a field symbol.

```
DATA: BEGIN OF LINE,
         COL1,
         COL2 VALUE 'X',
       END OF LINE.

FIELD-SYMBOLS <FS> LIKE LINE.

ASSIGN LINE TO <FS>.

MOVE <FS>-COL2 TO <FS>-COL1.
```

The field symbol <FS> is fully typed as a structure, and you can address its components in the program.

Attaching a Structure to a Field Symbol

The STRUCTURE addition forces a structured **view** of the data objects that you assign to a field symbol.

FIELD-SYMBOLS <FS> STRUCTURE <s> DEFAULT <f>.

The structure <s> is either a structured local data object in the program, or a **flat** structure from the ABAP Dictionary. <f> is a data object that must be assigned to the field symbol as a starting field. However, this assignment can be changed later using the ASSIGN statement.

When you assign a data object to the field symbol, the system only checks whether it is at least as long as the structure. You can address the individual components of the field symbol. It has the same technical attributes as the structure <s>.

If <s> contains components with type I or F, you should remember the possible effects of alignment [Page 195]. When you assign a data object to a field symbol with a structure, the data object must have the same alignment, otherwise a runtime error may result. In such cases, you are recommended to assign such data objects only to structured field symbols which retain the same structure as the field symbol at least over the length of the structure.

Defining Field Symbols

```
DATA: WA(10) VALUE '0123456789'.

DATA: BEGIN OF LINE1,
         COL1(3),
         COL2(2),
         COL3(5),
      END OF LINE1.

DATA: BEGIN OF LINE2,
         COL1(2),
         COL2 LIKE SY-DATUM,
      END OF LINE2.

FIELD-SYMBOLS: <F1> STRUCTURE LINE1 DEFAULT WA,
               <F2> STRUCTURE LINE2 DEFAULT WA.

WRITE: / <F1>-COL1, <F1>-COL2, <F1>-COL3,
       / <F2>-COL1, <F2>-COL2.
```

The output is:

012 34 56789
01 2345/67/89

This example declares two field symbols to which different structures are attached. The string WA is then assigned to each of them. The output shows that the field symbols assign the strings component by component according to the type of the components.

Assigning Data Objects to Field Symbols

Before you can work with a field symbol, you must assign a data object to it. If you attach a structure to a field symbol, you assign a data object to it in the declaration. Untyped field symbols point to the predefined data object [Page 132] SPACE once the program starts. SPACE has type C and length 1. Typed field symbols do not point to any field before a data object is assigned to them.

During a program, you can assign data objects to field symbols at any time. You can also assign a series of different data objects to the same field symbol during a program.

To assign a data object to a field symbol, use the ASSIGN statement. The ASSIGN statement has several variants and parameters.

The Basic Form of the ASSIGN Statement [Page 208]

Assigning Components of Structures to a Field Symbol [Page 213]

Defining the Data Type of a Field Symbol [Page 215]

Setting the Number of Decimal Places

Data Areas for Field Symbols [Page 217]

You can assign a line of an internal table to a field symbol. For further information, refer to Processing Internal Tables [Page 263].
The statement
UNASSIGN <FS>.
allows you to specify explicitly that a field symbol <FS> **should not** have a data object assigned to it. If you try to use an unassigned field symbol, a runtime error occurs. There is a special logical expression that you can use to check whether a data object is assigned to a field symbol [Page 238].

Basic Form of the ASSIGN Statement

The ASSIGN statement has the following basic forms:

Static ASSIGN

If you already know the name of the field that you want to assign to the field symbol when you write a program, use the static ASSIGN statement:

ASSIGN <f> TO <FS>.

When you assign the data object, the system checks whether the technical attributes of the data object <f> correspond to any type specifications for the field symbol <FS>. The field symbol adopts any generic attributes of <f> that are not contained in its own type specification. Following the assignment, it points to <f> in memory.

```
FIELD-SYMBOLS: <F1>, <F2> TYPE I.

DATA: TEXT(20)  TYPE C VALUE 'Hello, how are you?',
      NUM       TYPE I VALUE 5,
      BEGIN OF LINE1,
        COL1 TYPE F VALUE '1.1e+10',
        COL2 TYPE I VALUE '1234',
      END OF LINE1,
      LINE2 LIKE LINE1.

ASSIGN TEXT TO <F1>.
ASSIGN NUM TO  <F2>.
DESCRIBE FIELD <F1> LENGTH <F2>.
WRITE: / <F1>, 'has length', NUM.

ASSIGN LINE1 TO <F1>.
ASSIGN LINE2-COL2 TO <F2>.
MOVE <F1> TO LINE2.
ASSIGN 'LINE2-COL2 =' TO <F1>.
WRITE: / <F1>, <F2>.
```

The output is:

Hello, how are you? has length 20

LINE-COL2 = 1,234

The example declares two field symbols <F1> and <F2>. <F2> has the type I, which means that only type I fields may be assigned to it. <F1> and <F2> both point to different fields during the program.

Static ASSIGN with Offset Specification

In a static assign statement, you can use positive offset and length specifications to assign a part of a field [Page 196] to a field symbol.

ASSIGN <f>[+<o>][(<l>)] TO <FS>.

When you assign parts of fields to a field symbol, the following special conditions apply:

- The offset <o> and length <l> can be variables.
- The system does not check whether the selected part lies inside the field <f>. Both offset <o> and length <l> can be larger than the length of <f>. You can address memory beyond the boundary of <f>, but not beyond the data areas for field symbols [Page 217].
- If you do not specify the length <l>, the system automatically uses the length of the field <f>. If <o> is greater than zero, <FS> always points to an area beyond the limits of <f>.
- If <o> is smaller than the length of <f>, you can enter an asterisk (*) for <l> to prevent <FS> from referring to an address beyond the limits of <f>.

```
FIELD-SYMBOLS <FS>.

DATA: BEGIN OF LINE,
        STRING1(10) VALUE '0123456789',
        STRING2(10) VALUE 'abcdefghij',
      END OF LINE.

WRITE / LINE-STRING1+5.

ASSIGN LINE-STRING1+5 TO <FS>.
WRITE / <FS>.

ASSIGN LINE-STRING1+5(*) TO <FS>.
WRITE / <FS>.
```

The output is:

56789

56789abcde

56789

In this example, you can see the difference between an offset specification in a WRITE statement and an offset specification in an ASSIGN statement. With WRITE, the output is truncated at the end of LINE-STRING1. Specifying an offset greater than 9 would lead to an error during the syntax check. In the first ASSIGN statement, the memory area of length 10 beginning with offset 5 in LINE-STRING1 is assigned to the field symbol <FS>. The output is meaningful because the memory area behind LINE-STRING1 belongs to LINE-STRING2. In the second ASSIGN statement, the length specification * prevents the system from addressing the memory area after LINE-STRING1.

```
FIELD-SYMBOLS <FS>.

DATA: BEGIN OF LINE,
        A VALUE '1', B VALUE '2', C VALUE '3', D VALUE '4',
        E VALUE '5', F VALUE '6', G VALUE '7', H VALUE '8',
      END OF LINE,
      OFF TYPE I,
      LEN TYPE I VALUE 2.

DO 2 TIMES.
  OFF = SY-INDEX * 3.
```

Basic Form of the ASSIGN Statement

```
  ASSIGN LINE-A+OFF(LEN) TO <FS>.
  <FS> = 'XX'.
ENDDO.

DO 8 TIMES.
  OFF = SY-INDEX - 1.
  ASSIGN LINE-A+OFF(1) TO <FS>.
  WRITE <FS>.
ENDDO.
```

The output is:

1 2 3 X X 6 X X

The example show how field symbols can make it easier to access and manipulate regular structures. Note, however, that this flexible method of manipulating memory contents beyond field limits also has its dangers and may lead to runtime errors.

Dynamic ASSIGN

If you do not know the name of the field that you want to assign to the field symbol when you write a program, you can use a dynamic ASSIGN statement:

ASSIGN (<f>) TO <FS>.

This statement assigns the field whose name is contained in the field <f> to the field symbol <FS>. You cannot use offset and length in a dynamic ASSIGN.

At runtime, the system searches for the corresponding data object in the following order:

1. If the ASSIGN statement is in a procedure [Page 449], the system searches first in its local data.
2. If it cannot find the object in the local data (or if the ASSIGN statement is not in a procedure), it then looks in the local data of the program.
3. If the field does not exist in the global data of the program, the system looks in the table work areas [Page 130] declared with the TABLES statement in the main program of the current program group. A program group consists of a main program and all of the programs that are loaded into the same internal session as a result of other program calls.

If the search is successful and a field can be assigned to the field symbol, SY-SUBRC is set to 0. Otherwise, it is set to 4, and the field symbol remains unchanged. For security reasons, you should always check the value of SY-SUBRC after a dynamic ASSIGN to prevent the field symbol pointing to the wrong area.

Searching for the field in this way slows down the program. You should therefore only use the dynamic ASSIGN statement when absolutely necessary. If you know when you create the program that you want to assign a table work area [Page 130] to the field symbol, you can also use the following variant of the dynamic ASSIGN statement:

ASSIGN TABLE FIELD (<f>) TO <FS>.

The system then only searches within the table work areas in the main program of the current program group for the data object that is to be assigned to the field symbol.

Suppose we have three programs. The main program:

```
PROGRAM DEMO.
TABLES SBOOK.
SBOOK-FLDATE = SY-DATUM.
PERFORM FORM1(MYFORMS1).
```

and two other programs:

```
PROGRAM MYFORMS1.
FORM FORM1.
  PERFORM FORM2(MYFORMS2).
ENDFORM.
```

and

```
PROGRAM MYFORMS2.
FORM FORM2.
  DATA NAME(20) VALUE 'SBOOK-FLDATE'.
  FIELD-SYMBOLS <FS>.
  ASSIGN (NAME) TO <FS>.
  IF SY-SUBRC EQ 0.
    WRITE / <FS>.
  ENDIF.
ENDFORM.
```

The output is:

02.06.1998

The program group in the internal session now consists of the programs DEMO, MYFORMS1 and MYFORMS2. The field symbol <FS> is defined in MYFORMS2. After the dynamic ASSIGN statement, it points to the component FLDATE of the table work area SBOOK declared in the main program DEMO.

```
TABLES SBOOK.

DATA: NAME1(20) VALUE 'SBOOK-FLDATE',
      NAME2(20) VALUE 'NAME1'.

FIELD-SYMBOLS <FS>.

ASSIGN TABLE FIELD (NAME1) TO <FS>.
WRITE: / 'SY-SUBRC:', SY-SUBRC.

ASSIGN TABLE FIELD (NAME2) TO <FS>.
WRITE: / 'SY-SUBRC:', SY-SUBRC.
```

The output is:

SY-SUBRC: 0

SY-SUBRC: 4

In the first ASSIGN statement, the system finds the component FLDATE of the table work area SBOOK and SY-SUBRC is set to 0. In the second ASSIGN statement, the system does not find the field NAME1 because it is declared by the DATA statement and not by the TABLES statement. In this case, SY-SUBRC is set to 4.

Assigning Field Symbols

Instead of using the names of data objects, you can also assign field symbols to field symbols in all variants of the ASSIGN statement.

ASSIGN <FS1>[+<o>][(<l>)] TO <FS2>.

in a static ASSIGN and:

ASSIGN [TABLE FIELD] (<f>) TO <FS2>.

in a dynamic ASSIGN, where the field <f> contains the name of a field symbol <FS1>. <FS1> and <FS2> may be identical.

```
DATA: BEGIN OF S,
        A VALUE '1', B VALUE '2', C VALUE '3', D VALUE '4',
        E VALUE '5', F VALUE '6', G VALUE '7', H VALUE '8',
      END OF S.

DATA OFF TYPE I.

FIELD-SYMBOLS <FS>.

ASSIGN S-A TO <FS>.

DO 4 TIMES.
  OFF = SY-INDEX - 1.
  ASSIGN <FS>+OFF(1) TO <FS>.
  WRITE <FS>.
ENDDO.
```

The output is:

1 2 4 7

The program declares a structure with eight components S-A to S-H, and fills them with the digits 1 to 8. These character strings are stored regularly in memory. The component S-A is assigned initially to the field symbol <FS>. The statements in the DO loop [Page 245] have the following effect:

Loop pass 1:

<FS> points to S-A, OFF is zero, and S-A is assigned to <FS>

Loop pass 2:

<FS> points to S-A, OFF is one, and S-B is assigned to <FS>

Loop pass 3:

<FS> points to S-B, OFF is two, and S-D is assigned to <FS>

Loop pass 4:

<FS> points to S-D, OFF is three, and S-G is assigned to <FS>

Assigning Components of Structures to a Field Symbol

Casting [Page 215]

For a structured data object <s>, you can use the statement

ASSIGN COMPONENT <comp> OF STRUCTURE <s> TO <FS>.

to assign one of its components <comp> to the field symbol <FS>. You can specify the component <comp> either as a literal or a variable. If <comp> is of type C or a structure which has no internal tables as components, it specifies the name of the component. If <comp> has any other elementary data type, it is converted to type I and specifies the number of the component. In the assignment is successful, SY-SUBRC is set to 0. Otherwise, it returns 4.

This statement is particularly important for addressing components of structured data objects **dynamically**. If you assign a data object to a field symbol generically, or pass it generically to the parameter interface [Page 459] of a procedure, you cannot address its components either statically or dynamically. Instead, you must use the above statement. This allows indirect access either using the component name or its index number.

```
DATA: BEGIN OF LINE,
        COL1 TYPE I VALUE '11',
        COL2 TYPE I VALUE '22',
        COL3 TYPE I VALUE '33',
      END OF LINE.

DATA COMP(5) VALUE 'COL3'.

FIELD-SYMBOLS: <F1>, <F2>, <F3>.

ASSIGN LINE TO <F1>.
ASSIGN COMP TO <F2>.

DO 3 TIMES.
  ASSIGN COMPONENT SY-INDEX OF STRUCTURE <F1> TO <F3>.
  WRITE <F3>.
ENDDO.

ASSIGN COMPONENT <F2> OF STRUCTURE <F1> TO <F3>.
WRITE / <F3>.
```

The output is:

11 22 33

33

The field symbol <F1> points to the structure LINE, <F2> points to the field COMP. In the DO loop [Page 245], the components of LINE are specified by their numbers and assigned one by one to <F3>. After the loop, the component COL3 of LINE is specified by its name and assigned to <F3>. Note that ASSIGN COMPONENT is the only possible method of addressing the components of <F1>. Expressions such as <F1>-COL1 are syntactically incorrect.

Defining the Data Type of a Field Symbol

To set the data type of a field symbol independently of that of the objects that you assign to it, use the TYPE addition in the ASSIGN statement:

ASSIGN ..... TO <FS> TYPE <t>.

You can use the TYPE addition in all variants of the ASSIGN statement. At present, you can only use the elementary ABAP types (C, D, F, I, N, P, T, and X), ‘s’ for two-byte integers (with sign) and ‘b’ for one byte integers (without sign) as literals of variables for <t>.

There are two possible cases:

- Untyped field symbols

  If you use the TYPE addition, an untyped field symbol <FS> adopts the data type specified in <t> instead of the data type and output length of the data object assigned to it. If the field symbol is used after the assignment in the program, the assigned data object is not converted to the specified type <t>. However, the contents of the data object are interpreted as though they belonged to a field of type <t>. When you specify the type of a field symbol, you force a particular **view** on the data objects assigned to it.

- Typed field symbols

  Using the TYPE option with a typed field symbol makes sense if the data type of the data object to be assigned is incompatible with the typing of the field symbol, but you want to avoid the resulting error message. In this case, the specified type <t> and the typing of the field symbol must be compatible. The field symbol then retains its data type, regardless of the assigned data object.

A runtime error occurs if

- The specified data type is unknown,
- The length of the specified data type is incompatible with the type of the assigned field,
- there is an alignment [Page 195] error.

```
DATA TXT(8) VALUE '19980606'.

DATA MYTYPE(1) VALUE 'X'.

FIELD-SYMBOLS <FS>.

ASSIGN TXT TO <FS>.
WRITE / <FS>.

ASSIGN TXT TO <FS> TYPE 'D'.
WRITE / <FS>.

ASSIGN TXT TO <FS> TYPE MYTYPE.
WRITE / <FS>.
```

The output is:

19980606

06061998

Defining the Data Type of a Field Symbol

3139393830363036

In this example, the character string TXT is assigned to <FS> three times. In the first assignment, the type is not specified. The second time, the type is specified as D, and finally as X. The format of the second output line depends on the settings of the current user in their user master record. The numbers in the last line are the hexadecimal codes of the characters in TXT. They are platform-specific (in this case, ASCII).

Data Areas for Field Symbols

You can only assign data objects from the data areas of the ABAP program to a filed symbol. When you assign a data object to a field symbol, the system checks at runtime to ensure that no data is lost due to the field symbol addressing memory outside the data area.

The data areas of an ABAP program are:

- The table memory area for internal tables. The size of this storage area depends on the number of table lines which is not fixed, but determined dynamically at runtime.
- The DATA storage area for other data objects. The size of this storage area is fixed during the data declaration.

Field symbols cannot point to addresses outside these areas. If you assign data objects to a field symbol and point to addresses outside these areas, a runtime error occurs.

Certain system information, such as the control blocks of internal tables, is also stored in the DATA storage area. Therefore, despite the runtime checks, you may unintentionally overwrite some of these fields and cause subsequent errors (for example, destruction of the table header).

```
PROGRAM DEMO.

DATA: TEXT1(10), TEXT2(10), TEXT3(5).

FIELD-SYMBOLS <FS>.

DO 100 TIMES.
  ASSIGN TEXT1+SY-INDEX(1) TO <FS>.
ENDDO.
```

After starting DEMO, a runtime error occurs. The short dump message begins as follows:

```
ABAP Runtime error  ASSIGN_OFFSET+LENGTH_TOOLARGE
      Occurred on 1998/06/02 at 23:32:09

---------------------------------------------------

Error in ASSIGN: Memory protection error

...
```

The DATA memory area is at least 25 bytes wide (it can be expanded due to alignment of the data areas in memory). In one of the loop passes, the program tries to access an address outside this area. The termination message also contains the contents of the field SY-INDEX, which may be different from case to case. Up to the 24th loop pass, no error occurs. If you replace TEXT1 with TEXT2 in the ASSIGN statement, the error occurs ten loop passes earlier.

Data References

Beispiel zu Datenreferenzen [Page 224]

Data references are pointers to data objects. They occur in ABAP as the contents of data reference variables. You can use data references to create data objects dynamically. You can also create references to existing data objects. You can only dereference a data reference using a special assignment to a field symbol [Page 201].

Reference Variables [Page 220]

Creating Data Objects Dynamically [Page 221]

Getting References to Data Objects [Page 222]

Dereferencing Data References [Page 223]

Reference Variables

Reference variables contain references. The actual contents of a reference variable, namely the value of a reference, is not visible in an ABAP program. ABAP contains data references and object references [Page 1307]. Consequently, there are two kinds of data reference variables - data reference variables and object reference variables.

You create the data type of a data reference variable using:

TYPES <t_dref> TYPE REF TO DATA.

You can create a data reference variable either by referring to the above data type or using:

DATA <dref> TYPE REF TO DATA.

Reference variables are handled in ABAP like other data objects with an elementary data type. This means that a reference variable can be defined as a component of a complex data object such as a structure or internal table as well as a single field.

Reference variables are initial when you declare them. They do not point to an object. You cannot deference [Page 223] an initial reference variable. A data reference variable can point to a data object if you

- Use it to create a data object dynamically [Page 221].
- Use it to get a reference to a data object [Page 222].
- Assign an existing data reference to it from another data reference variable.

You can assign references between data reference variables using the MOVE statement or the assignment operator (=), and also when you fill an internal table or pass interface parameters to a procedure. If you assign references in this way, the operands must be typed as data reference variables. You cannot assign to object reference variables or other variables. After the assignment, the reference in the target variable points to the same data object as the reference in the source variable.

Creating Data Objects Dynamically

All of the data objects that you define in the declaration part of a program using statements such as DATA are created statically, and already exist when you start the program. To create a data object dynamically during a program, you need a data reference variable and the following statement:

CREATE DATA <dref> TYPE <type>|LIKE <obj>.

This statement creates a data object in the internal session of the current ABAP program. After the statement, the data reference in the data reference variable <dref> points to the object. The data object that you create does not have its own name. You can only address it using a data reference variable. To access the contents of the data object, you must dereference the data reference [Page 223].

You must specify the data type of the object using the TYPE [Page 112] or LIKE [Page 116] addition. In the TYPE addition, you can specify the data type **dynamically** as the contents of a field (this is not possible with other uses of TYPE).

CREATE DATA <dref> TYPE (<name>).

Here, <name> is the name of a field that contains the name of the required data type.

Getting References to Data Objects

The following statements allow you to place a data reference to an existing data object in a reference variable:

GET REFERENCE OF <obj> INTO <dref>.

<obj> can be a statically-declared data object, or a field symbol pointing to any object (including a dynamic object). If you specify a field symbol to which an object has not yet been assigned, a runtime error occurs.

If you place a reference to a local field in a procedure [Page 449] in a global reference variable, the reference will become invalid when you leave the procedure. You cannot then deference [Page 223] the reference variable.

Dereferencing Data References

To access the contents of the data object to which a data reference is pointing, you must dereference it.

ASSIGN <dref>->* TO <FS> [CASTING ...].

This statement assigns the data object to which the data reference in the reference variable <dref> to the field symbol [Page 201] <FS>. If the assignment is successful, SY-SUBRC is set to zero.

If the field symbol is fully generic, it adopts the data type of the data object. If the field symbol is partially or fully typed, the system checks the data types for compatibility. Casting [Page 215] is also possible for the assigned data object.

If the data reference in <dref> is initial or invalid, you cannot dereference it. The field symbol remains unchanged, and SY-SUBRC is set to 4.

If you create a data object dynamically, the only way to access its contents is to use dereferencing.

Data References: Example

```
TYPES: BEGIN OF t_struct,
         col1 TYPE i,
         col2 TYPE i,
       END OF t_struct.

DATA: dref1 TYPE REF TO data,
      dref2 TYPE REF TO data.

FIELD-SYMBOLS: <fs1> TYPE t_struct,
               <fs2> TYPE i.

CREATE DATA dref1 TYPE t_struct.
ASSIGN dref1->* TO <fs1>.

<fs1>-col1 = 1.
<fs1>-col2 = 2.

dref2 = dref1.
ASSIGN dref2->* TO <fs2> CASTING.
WRITE / <fs2>.

GET REFERENCE OF <fs1>-col2 INTO dref2.
ASSIGN dref2->* TO <fs2>.
WRITE / <fs2>.
```

The output is:

```
1

2
```

This example declares two data reference variables dref1 and dref2. dref1 is used to create a structured dynamic data object. This is dereferenced with the field symbol <fs1>, and values are then assigned to it. dref1 is assigned to dref2. dref2 then points to the structured data object. When dref2 is dereferenced, it is cast to the elementary type i of field symbol <fs2>. Its first component is assigned to the field symbol. GET REFERENCE is then used to get a reference to the second component not the structured data object in dref2. It is dereferenced without casting.

Logical Expressions

When writing application programs, you often need to formulate conditions. These are used mostly to control the flow of the program [Page 240], or to decide whether to exit a processing block. You formulate a condition using logical expressions. A logical condition can be either true or false.

Comparing Data Objects

You can express a logical expression as a comparison between data objects:

.... <f1> <operator> <f2> ...

Depending on the data types of the operands <f1> and <f2>, you can use different logical operators.

Comparisons Between Different Data Types [Page 226]

Comparing Strings [Page 230]

Comparing Bit Sequences [Page 233]

Other Logical Expressions

Along with data object comparisons, you can formulate logical expressions that check whether data objects meet given conditions:

Checking Whether a Field Belongs to a Range [Page 235]

Checking for the Initial Value [Page 236]

Checking Selection Criteria [Page 237]

Checking Whether a Field Symbol is Assigned [Page 238]

Combining Several Logical Expressions

You can link several logical expressions together in a single condition:

Combining Several Logical Expressions [Page 239]

Comparisons Between Different Data Types

Use the following logical operators to compare two data objects with different data types:

<operator>	Meaning
EQ	equal to
=	equal to
NE	not equal to
<>	not equal to
><	not equal to
LT	less than
<	less than
LE	less than or equal to
<=	less than or equal to
GT	greater than
>	greater than
GE	greater than or equal to
>=	greater than or equal to

Both operands must either be compatible or convertible.

Comparing Elementary Data Types

If a data object has one of the eight predefined ABAP types [Page 97], it is compared as follows:

If the operands are **compatible**, there is no conversion. The procedure for the comparison is as follows: Numeric fields (types I, F, and P) and numeric strings (type N) are compared by their numeric values. For other data types (C, D, T, X), the comparison runs from left to right. The first character from the left that is different in each field determines which operand is greater. Text fields (type C) are compared based on the underlying character codes. In date field comparisons (type D), the more recent date is greater than the elder. In time field comparisons (type T), the later time is greater than the earlier. Hexadecimal fields (type X) are compared according to the values of their bytes.

When you compare incompatible operands with different lengths but the same data type, the comparison is as follows: Packed numbers (type P) are compared according to their numeric value without being converted. For the other types (C, N, X) with different lengths, the shorter operand is converted to the length of the longer before the comparison. It is then filled out as follows: Character strings (type C) are filled with spaces from the right, numeric strings (type N) are filled from the left with zeros, and hexadecimal fields are filled from the right with hexadecimal zero.

```
DATA: HEX1(3) TYPE X VALUE '34B7A1',
      HEX2(1) TYPE X VALUE 'F0'.

IF HEX2 > HEX1.
  ...
ENDIF.
```

The logical expression in the IF statement is true, since the first byte of HEX2 is greater than the first byte of HEX1.

When you compare incompatible data objects whose data types are different, the system performs a type conversion [Page 186] before the comparison according to the following rules:

1. If one of the operands is a floating point number (type F), the system also converts the other operands to type F.
2. If one of the operands is a packed field (type P), the system also converts the other operands to type P.
3. If one of the operands is a date field (type D) or a time field (type T), the system converts the other operands to type D or T respectively. Comparisons between date and time fields are not supported, and lead to a program termination.
4. If one of the operands is a character field (type C) and the other operand is a hexadecimal field (type X), the system converts the operand of type X to type C.
5. If one of the operands is a character field (type C) and the other a numeric field (type N), the system converts both operands to type P.

```
DATA: NUM(4)  TYPE N VALUE '1234',
      TEXT(5) TYPE C VALUE '567.8'.

IF NUM > TEXT.
  ...
ENDIF.
```

The logical expression in the IF statement is true, since the value of NUM is greater than the value of TEXT after the conversion to packed numbers. If NUM had had type C, the logical expression would have been false, since there would have been no conversion to a packed number.

Comparing References

When you compare compatible or convertible reference variables in ABAP Objects [Page 1291], it only makes sense to use the equality (EQ or =) and inequality (NE, <>, or ><) operators. The equality comparison is true if both reference variables contain references to the same object. It is false if they contain references to different objects. The converse is true if you are testing for inequality.

```
INTERFACE I1.
  ...
ENDINTERFACE.

CLASS C1 DEFINITION.
  PUBLIC SECTION.
  INTERFACES I1.
  ...
ENDCLASS.

DATA: R1 TYPE REF TO I1,
      R2 TYPE REF TO C1.

CREATE OBJECT R2.

R1 = R2.
```

Comparisons Between Different Data Types

```
IF R1 = R2.
  ...
ENDIF.
```

The logical expression in the IF statement is true, since both references point to the same object.

Comparing Structures

Compatible structures are broken down into their elementary components, which are then compared. Two structures are equal if all of their elementary components are equal. If the structures are unequal, the first pair of elements that are unequal determine which structure is larger.

```
DATA: BEGIN OF STRUCT1,
        COL1 TYPE I VALUE 1,
        COL2 TYPE P DECIMALS 2 VALUE '56.78',
      END OF STRUCT1.

DATA: BEGIN OF STRUCT2,
        COMP1 TYPE I VALUE 10,
        COMP2 TYPE P DECIMALS 2 VALUE '12.34',
      END OF STRUCT2.

IF STRUCT1 < STRUCT2.
  ...
ENDIF.
```

The logical expression in the IF expression is true, since the value of the first component of STRUCT1 is less than the value of the second component.

When you compare incompatible structures, or compare structures with elementary fields, the structures are converted [Page 192] into type C fields before the conversion, and then treated like elementary fields.

```
DATA: BEGIN OF STRUCT,
        COL1(5) TYPE C VALUE '12345',
        COL2(5) TYPE N VALUE '12345',
      END OF STRUCT.

DATA TEXT(10) TYPE C VALUE '1234512345'.

IF STRUCT = TEXT.
  ...
ENDIF.
```

The logical expression in the IF expression is true, since structure STRUCT has the same contents as the field TEXT once it has been converted into a field with type C.

Comparing Internal Tables

Refer to Comparing Internal Tables [Page 269].

Comparing Strings

Similarly to the special statements for processing strings [Page 161], there are special comparisons that you can apply to strings with types C, D, N, and T. You can use the following operators:

<operator>	Meaning
CO	Contains Only
CN	Contains Not only
CA	Contains Any
NA	contains Not Any
CS	Contains String
NS	contains No String
CP	Contains Pattern
NP	contains No Pattern

There are no conversions with these comparisons. Instead, the system compares the characters of the string. The operators have the following functions:

CO (Contains Only)

The logical expression

<f1> CO <f2>

is true if <f1> contains only characters from <f2>. The comparison is case-sensitive. Trailing blanks are included. If the comparison is true, the system field SY-FDPOS contains the length of <f1>. If it is false, SY-FDPOS contains the offset of the first character of <f1> that does not occur in <f2>.

CN (Contains Not only)

The logical expression

<f1> CN <f2>

is true if <f1> does also contains characters other than those in <f2>. The comparison is case-sensitive. Trailing blanks are included. If the comparison is true, the system field SY-FDPOS contains the offset of the first character of <f1> that does not also occur in <f2>. If it is false, SY-FDPOS contains the length of <f1>.

CA (Contains Any)

The logical expression

<f1> CA <f2>

is true if <f1> contains at least one character from <f2>. The comparison is case-sensitive. If the comparison is true, the system field SY-FDPOS contains the offset of the first character of <f1> that also occurs in <f2>. If it is false, SY-FDPOS contains the length of <f1>.

NA (contains Not Any)

The logical expression

<f1> NA <f2>

is true if <f1> does not contain any character from <f2>. The comparison is case-sensitive. If the comparison is true, the system field SY-FDPOS contains the length of <f1>. If it is false, SY-FDPOS contains the offset of the first character of <f1> that occurs in <f2>.

CS (Contains String)

The logical expression

<f1> CS <f2>

is true if <f1> contains the string <f2>. Trailing spaces are ignored and the comparison is **not** case-sensitive. If the comparison is true, the system field SY-FDPOS contains the offset of <f2> in <f1>. If it is false, SY-FDPOS contains the length of <f1>.

NS (contains No String)

The logical expression

<f1> NS <f2>

is true if <f1> does not contain the string <f2>. Trailing spaces are ignored and the comparison is **not** case-sensitive. If the comparison is true, the system field SY-FDPOS contains the length of <f1>. If it is false, SY-FDPOS contains the offset of <f2> in <f1>.

CP (Contains Pattern)

The logical expression

<f1> CP <f2>

is true if <f1> contains the pattern <f2>. If <f2> is of type C, you can use the following wildcards in <f2>:

- for any character string *
- for any single character +

Trailing spaces are ignored and the comparison is **not** case-sensitive. If the comparison is true, the system field SY-FDPOS contains the offset of <f2> in <f1>. If it is false, SY-FDPOS contains the length of <f1>.

If you want to perform a comparison on a particular character in <f2>, place the escape character # in front of it. You can use the escape character # to specify

- characters in upper and lower case
- the wildcard character "*" (enter **#***)
- the wildcard character "+" (enter **#+**)
- the escape symbol itself (enter **##**)
- blanks at the end of a string (enter **#___**)

NP (contains No Pattern)

The logical expression

<f1> NP <f2>

is true if <f1> does not contain the pattern <f2>. In <f2>, you can use the same wildcards and escape character as for the operator CP.

Comparing Strings

Trailing spaces are ignored and the comparison is **not** case-sensitive. If the comparison is true, the system field SY-FDPOS contains the length of <f1>. If it is false, SY-FDPOS contains the offset of <f2> in <f1>.

```
DATA: F1(5) TYPE C VALUE <f1>,
      F2(5) TYPE C VALUE <f2>.

IF F1 <operator> F2.
  WRITE: /  'Comparison true, SY-FDPOS=', SY-FDPOS.
ELSE.
  WRITE: /  'Comparison false, SY-FDPOS=', SY-FDPOS.
ENDIF.
```

The following table shows the results of executing this program, depending on which operators and values of F1 and F2.

<f1>	<operator>	<f2>	Result	SY-FDPOS
'BD '	CO	'ABCD '	true	5
'BD '	CO	'ABCDE'	false	2
'ABC12'	CN	'ABCD '	true	3
'ABABC'	CN	'ABCD '	false	5
'ABcde'	CA	'Bd '	true	1
'ABcde'	CA	'bD '	false	5
'ABAB '	NA	'AB '	false	0
'ababa'	NA	'AB '	true	5
'ABcde'	CS	'bC '	true	1
'ABcde'	CS	'ce '	false	5
'ABcde'	NS	'bC '	false	1
'ABcde'	NS	'ce '	true	5
'ABcde'	CP	'*b*'	true	1
'ABcde'	CP	'*#b*'	false	5
'ABcde'	NP	'*b*'	false	1
'ABcde'	NP	'*#b*'	true	5

Comparing Bit Sequences

Use the following three operators to compare the bit sequence of the first operand with that of the second:

<operator>	Meaning
O	bits are one
Z	bits are zero
M	bits are mixed

The second operand must have type X. The comparison takes place over the length of the second operand. The first operand is not converted to type X.

The function of the operators is as follows:

O (bits are one)

The logical expression

<f> O <hex>

is true if the bit positions that are 1 in <hex>, are 1 in <f>. In terms of set operations with bit sequences [Page 183], this comparison is the same as finding out whether the set represented by <hex> is a subset of that represented by <f>.

Z (bits are zero)

The logical expression

<f> Z <hex>

is true if the bit positions that are 1 in <hex>, are 0 in <f>.

M (bits are mixed)

The logical expression

<f> M <hex>

is true if from the bit positions that are 1 in <hex>, at least one is 1 and one is 0 in <f>.

```
DATA: TEXT TYPE C VALUE 'C',
      HEX TYPE X,
      I TYPE I.

HEX = 0.

DO 256 TIMES.

  I = HEX.

  IF TEXT O HEX.
    WRITE: / HEX, I.
  ENDIF.

  HEX = HEX + 1.

ENDDO.
```

The output is as follows:

00 0
01 1
02 2
03 3
40 64
41 65
42 66
43 67

Here, the bit structure of the character 'C' is compared to all hexadecimal numbers HEX between '00' and 'FF' (255 in the decimal system), using the operator O. The decimal value of HEX is determined by using the automatic type conversion during the assignment of HEX to I. If the comparison is true, the hexadecimal number and its decimal value are displayed on the screen. The following table shows the bit sequences of the numbers:

hexadecimal	decimal	Bit sequences
00	0	00000000
01	1	00000001
02	2	00000010
03	3	00000011
40	64	01000000
41	65	01000001
42	66	01000010
43	67	01000011

The bit sequence of the character 'C' is defined for the current hardware platform by its ASCII code number 67. The numbers that occur in the list display are those in which the same bit position is filled with 1 as in the bit sequence of ‘C’. The sequence 01000011 is the universal set of the bit sequences.

Checking Whether a Field Belongs to a Range

Use the following logical expression to check whether the value of a field lies within a particular range:

.... <f1> BETWEEN <f2> AND <f3> .....

The expression is true if the value of <f1> lies in the interval between <f2> and <f3>. It is a shortened form of the following expression:

IF <f1> GE <f2> AND <f1> LE <f3>.

The operands may have different data types. If necessary, they are converted as described in Comparisons Between Data Types [Page 226].

```
DATA: NUMBER TYPE I,
      FLAG.

...

NUMBER = ...

...

IF NUMBER BETWEEN 3 AND 7.
  FLAG = 'X'.
ELSE.
  FLAG = ' '.
ENDIF.
```

In this example, the value of the field FLAG is set to X if the value of NUMBER is between 3 and 7.

Checking for the Initial Value

Use the following logical expression to check whether the value of a field is initial:

.... <f> IS INITIAL .....

This expression is true if the field <f> contains the initial value for its type. To set the initial value of an elementary or aggregate field, use the statement CLEAR <f> [Page 150].

```
DATA FLAG VALUE 'X'.

IF FLAG IS INITIAL.
  WRITE / 'Flag is initial'.
ELSE.
  WRITE / 'Flag is not initial'.
ENDIF.

CLEAR FLAG.

IF FLAG IS INITIAL.
  WRITE / 'Flag is initial'.
ELSE.
  WRITE / 'Flag is not initial'.
ENDIF.
```

The output is as follows:

```
Flag is not initial

Flag is initial.
```

Here, the character string FLAG does not contain its initial value after the DATA statement because it is set to the start value 'X' using the VALUE parameter. When the CLEAR statement is executed, it is reset to its initial value.

Checking Selection Criteria

Use the following logical expression to check whether the contents of a field satisfy the criteria in a selection table:

... <f> IN <seltab> ....

For further information about selection criteria and how to use them, refer to Selection Screens [Page 681].

```
DATA WA TYPE SPFLI.

SELECT-OPTIONS S_CARRID FOR WA-CARRID.

IF 'LH' IN S_CARRID.
  WRITE 'LH was selected'.
ENDIF.
```

The logical expression in the IF statement is true if the user entered a selection on the selection screen containing the value ‘LH’.

Checking Whether a Field Symbol is Assigned

To check whether a field is assigned to a field symbol, use the following logical expression:

... <FS> IS ASSIGNED.

The expression is false if the field symbol [Page 201] <fs> is not explicitly assigned to a field, that is, in the following cases:

- Directly after you have declared a field symbol without a particular structure When you declare a field symbol with a particular structure, a data object is assigned to it straight away. The expression is false for untyped field symbols before you have assigned a field to them in an ASSIGN statement, even though they point to the predefined data object SPACE.
- After UNASSIGN <FS>.
- When a local field that was assigned to <FS> no longer exists.
- When <FS> points to a global parameter of the interface of a function module that is inactive.

```
FIELD-SYMBOLS <FS>.

DATA TEXT(10) TYPE C VALUE 'Assigned!'.

ASSIGN TEXT TO <FS>.

IF <FS> IS ASSIGNED.
  WRITE <FS>.
ENDIF.
```

The output is:

```
Assigned!
```

since the logical expression in the IF statement is true.

Combining Several Logical Expressions

You can combine several logical expressions together in one single expression by using the logical link operators AND and OR:

- To combine several logical expressions together in one single expression which is true only if all of the component expressions are true, link the expressions with AND.
- To combine several logical expressions together in one single expression which is true if at least one of the component expressions is true, link the expressions with OR.

To negate the result of a logical expression, you can precede it with the NOT operator.

NOT takes priority over AND, and AND takes priority over OR. However, you can use any combination of parentheses to specify the processing sequence. As in mathematical expressions, ABAP interprets each parenthesis as a separate word. You must therefore leave at least one space before and after each one.

ABAP processes logical expressions from left to right. If it recognizes one of the component expressions as true or false, it does not perform the remaining comparisons or checks in this component. This means that you can improve performance by organizing logical expressions in such a way that you place comparisons which are often false at the beginning of an AND chain and expensive comparisons, such as searches for character strings, at the end.

```
DATA: F    TYPE F VALUE '100.00',
      N(3) TYPE N VALUE '123',
      C(3) TYPE C VALUE '456'.

WRITE 'The following logical expression is true:'.

IF ( C LT N ) AND ( N GT F ).
  WRITE: / '(',C,'lt',N,') AND (',N,'gt',F,')'.
ELSE.
  WRITE: / '(',C,'ge',N,') OR (',N,'le',F,')'.
ENDIF.
```

The output is:

```
The following logical expression is true:

( 456 ge 123 ) OR ( 123 le  1.000000000000000E+02 )
```

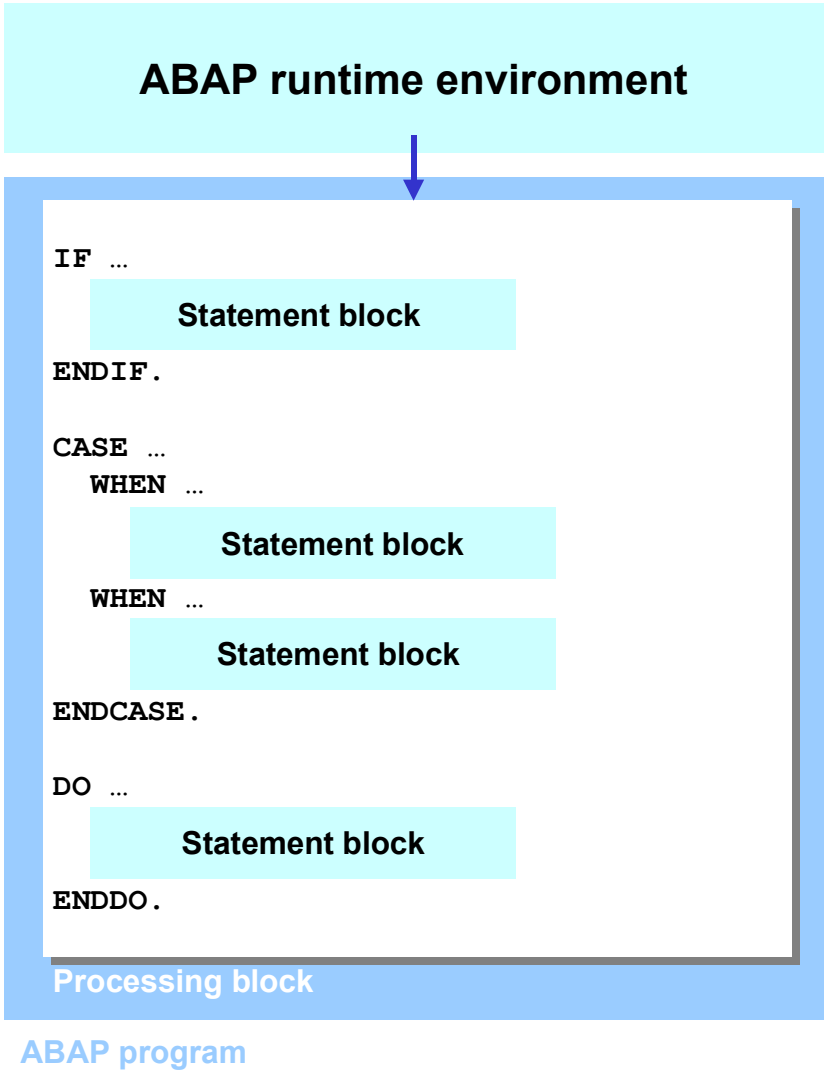
The logical expression in the IF statement is true, and the inverted expression is displayed.

Controlling the Program Flow

You can execute different parts of programs conditionally or in loops using the standard keywords IF, CASE, DO, and WHILE.

When controlling the flow of an ABAP program, remember that the structure of the program [Page 44] is made up of processing blocks [Page 49] that cannot be nested. This section describes how to control the flow of a program within a processing block. The keywords covered here do not allow you to branch outside the processing block in which you use them. You can regard this as **internal control** of an ABAP program, as opposed to the **external** control provided by events in the ABAP runtime environment.

To control the internal flow of a processing block, you can use control structures to divide it up into smaller statement blocks according to the principles of structured programming.



Unlike processing blocks, control structures can be nested.

Branching Conditionally [Page 242]

Loops [Page 245]

Branching Conditionally

When you branch conditionally, a processing block is executed or not based on the result of one or more logical conditions. ABAP contains two control structures for conditional branching.

The IF Control Structure

This control structure is introduced with the IF statement. The IF statement allows you to divert the program flow to a particular statement block, depending on a condition. The statement block concludes either with ENDIF, ELSEIF, or ELSE.

```
IF <condition1>.
  <statement block>
ELSEIF <condition2>
  <statement block>.
ELSEIF <condition3>.
  <statement block>
.....
ELSE.
  <statement block>
ENDIF.
```

To formulate conditions in IF or ELSEIF statements, you can use any logical expression [Page 225].

If the first condition is true, the system executes all the statements up to the end of the first statement block and then continues processing after the ENDIF statement. If the first condition is not true, the program jumps to the next ELSEIF statement and executes it like an IF statement. ELSE begins a statement block which is processed if none of the IF or ELSEIF conditions is true. The end of the last statement block must always be concluded by ENDIF.

You can nest IF control structures. However, the statement blocks must all end within the current processing block. So, for example, an IF - ENDIF block may not contain an event keyword.

```
DATA: TEXT1(30) VALUE 'This is the first text',
      TEXT2(30) VALUE 'This is the second text',
      TEXT3(30) VALUE 'This is the third text',
      STRING(5) VALUE 'eco'.

IF TEXT1 CS STRING.
  WRITE / 'Condition 1 is fulfilled'.
ELSEIF TEXT2 CS STRING.
  WRITE / 'Condition 2 is fulfilled'.
ELSEIF TEXT3 CS STRING.
  WRITE / 'Condition 3 is fulfilled'.
ELSE.
  WRITE / 'No condition is fulfilled'.
ENDIF.
```

The output is:
Condition 2 is fulfilled.
Here, the second logical expression TEXT2 CS STRING is true because the string "eco" occurs in TEXT2.

The CASE Control Structure

This control structure is introduced with the CASE statement. The CASE control structure allows you to control which statement blocks are processed based on the contents of a data object.

CASE <f>.
  WHEN <f_{11}> [OR <f_{12}> OR ...].
    <Statement block>
  WHEN <f_{21}>.[OR <f_{22}> OR ...]
    <Statement block>
  WHEN <f_{31}> [OR <f_{32}> OR ...].
    <statement block>
  WHEN ...
  ......
  WHEN OTHERS.
    <statement block>
ENDCASE.

The statement block following a WHEN statement is executed if the contents of <f> are the same as those of one of the fields <f_{ij}>. Afterwards, the program carries on processing after the ENDCASE statement. The statement block after the optional WHEN OTHERS statement is executed if the contents of <f> does not equal any of the <f_{ij}> contents. The last statement block must be concluded with ENDCASE.

The CASE control structure is a shortened form of the following IF structure:

IF <f> = <f_{11}> OR <f> = <f_{12}> OR <f> = ...
  <Statement block>
ELSEIF <f> = <f_{21}> OR <f> = <f_{22}> OR <f> =...
  <Statement block>
ELSEIF <f> = <f_{21}> OR <f> = <f_{22}> OR <f> =...
  <statement block>
ELSEIF <f> = ...
  ...
ELSE.
  <statement block>
ENDIF.

You can nest CASE control structures and also combine them with IF structures. However, they must always end with an ENDCASE statement within the current processing block.

```
DATA: TEXT1   VALUE 'X',
      TEXT2   VALUE 'Y',
      TEXT3   VALUE 'Z',
      STRING  VALUE 'A'.

CASE STRING.
  WHEN TEXT1 OR TEXT2.
    WRITE: / 'String is', TEXT1, 'OR', TEXT2.
  WHEN TEXT3.
    WRITE: / 'String is', TEXT3.
  WHEN OTHERS.
    WRITE: / 'String is not', TEXT1, TEXT2, TEXT3.
ENDCASE.
```

The output is:

```
String is not X Y Z
```

Here, the last statement block after WHEN OTHERS is processed because the contents of STRING, ‘A’, does not equal ‘X’, ‘Y’, or ‘Z’.

Loops

In a loop, a statement block is executed several times in succession. There are four kinds of loops in ABAP:

- Unconditional loops using the DO statement.
- Conditional loops using the WHILE statement.
- Loops through internal tables and extract datasets using the LOOP statement.
- Loops through datasets from database tables using the SELECT statement.

This section deals with DO and WHILE loops. SELECT is an Open SQL statement, and is described in the Open SQL [Page 1041] section. The LOOP statement is described in the sections on internal tables [Page 251] and extract datasets [Page 331].

Unconditional Loops

To process a statement block several times unconditionally, use the following control structure:

```
DO [<n> TIMES] [VARYING <f> FROM <f1> NEXT <f2>].
  <Statement block>
ENDDO.
```

If you do not specify any additions, the statement block is repeated until it reaches a termination statement such as EXIT or STOP (see below). The system field SY-INDEX contains the number of loop passes, including the current loop pass.

Use the TIMES addition to restrict the number of loop passes to <n>. <n> can be literal or a variable. If <n> is 0 or negative, the system does not process the loop. If you do not use the TIMES option, you must ensure that the loop contains at least one EXIT or STOP statement to avoid endless loops.

You can assign new values to a variable <f> in each loop pass by using the VARYING option. You can use the VARYING addition more than once in a DO statement. $<f_1>$ and $<f_2>$ are the first two fields of a sequence of fields the same distance apart in memory and with the same type and length. In the first loop pass, <f> takes the value $<f_1>$, in the second loop pass, $<f_2>$, and so on. If you change the value of the field <f> within the DO loop, the value of the current field $<f_i>$ is also changed. You must ensure that there are not more loop passes than fields in the sequence, otherwise a runtime error occurs.

You can nest DO loops and combine them with other loop forms.

Simple example of a DO loop:

```
DO.
WRITE SY-INDEX.
  IF SY-INDEX = 3.
    EXIT.
  ENDIF.
ENDDO.
```

The output is:

```
1           2           3
```

The loop is processed three times. Here, the processing passes through the loop three times and then leaves it after the EXIT statement.

Example of two nested loops with the TIMES addition:

```
DO 2 TIMES.
  WRITE SY-INDEX.
  SKIP.
  DO 3 TIMES.
    WRITE SY-INDEX.
  ENDDO.
  SKIP.
ENDDO.
```

The output is:

```
1
1           2           3
2
1           2           3
```

The outer loop is processed twice. Each time the outer loop is processed, the inner loop is processed three times. Note that the system field SY-INDEX contains the number of loop passes for each loop individually.

Example of the VARYING addition in a DO loop:

```
DATA: BEGIN OF TEXT,
        WORD1(4) VALUE 'This',
        WORD2(4) VALUE 'is',
        WORD3(4) VALUE 'a',
        WORD4(4) VALUE 'loop',
      END OF TEXT.

DATA: STRING1(4), STRING2(4).

DO 4 TIMES VARYING STRING1 FROM TEXT-WORD1 NEXT TEXT-WORD2.
  WRITE STRING1.
  IF STRING1 = 'is'.
    STRING1 = 'was'.
  ENDIF.
ENDDO.

SKIP.

DO 2 TIMES VARYING STRING1 FROM TEXT-WORD1 NEXT TEXT-WORD3
           VARYING STRING2 FROM TEXT-WORD2 NEXT TEXT-WORD4.
  WRITE: STRING1, STRING2.
ENDDO.
```

The output is:

```
This is   a    loop
```

This was a loop

The structure TEXT represents a series of four equidistant fields in memory. Each time the first DO loop is processed, its components are assigned one by one to STRING1. Whenever STRING1 contains ‘is’, its contents are changed to ‘was’. This also changes TEXT-WORD2 to ‘was’. Each time the second DO loop is processed, the components of TEXT are passed to STRING1 and to STRING2.

Conditional Loops

To repeat a statement block for as long as a certain condition is true, use the following control structure:

WHILE <condition> [VARY <f> FROM <f_1> NEXT <f_2>].
  <statement block>
ENDWHILE.

<condition> can be any logical expression [Page 225]. The statement block between WHILE and ENDWHILE is repeated as long as the condition is true or until a termination statement such as EXIT or STOP occurs. The system field SY-INDEX contains the number of loop passes, including the current loop pass. The VARY option of the WHILE statement works in the same way as the VARYING option of the DO statement (see above).

To avoid endless loops, you must ensure that the condition of a WHILE statement can be false, or that the statement block contains a termination statement such as EXIT or STOP.

You can nest WHILE loops to any depth, and combine them with other loop forms.

```
DATA: LENGTH     TYPE I VALUE 0,
      STRL       TYPE I VALUE 0,
      STRING(30) TYPE C VALUE 'Test String'.

STRL = STRLEN( STRING ).

WHILE STRING NE SPACE.
  WRITE STRING(1).
  LENGTH = SY-INDEX.
  SHIFT STRING.
ENDWHILE.

WRITE: / 'STRLEN:         ', STRL.
WRITE: / 'Length of string:', LENGTH.
```

The output appears as follows:

```
T e s t   S t r i n g
STRLEN:                    11
Length of String:          11
```

Here, a WHILE loop is used to determine the length of a character string. This is done by shifting the string one position to the left each time the loop is processed until it contains only blanks. This example has been chosen to demonstrate the WHILE statement. Of course, you can determine the length of the string far more easily and efficiently using the STRLEN function.

Terminating Loops

ABAP contains termination statements that allow you to terminate a loop prematurely. There are two categories of termination statement - those that only apply to the loop, and those that apply to the entire processing block in which the loop occurs. The STOP and REJECT statements belong to the latter group, and are described in more detail under Leaving Event Blocks [Page 966].

The termination statements that apply only to the loop in which they occur are CONTINUE, CHECK, and EXIT. You can only use the CONTINUE statement in a loop. CHECK and EXIT, on the other hand, are context-sensitive. Within a loop, they only apply to the execution of the loop itself. Outside of a loop, they terminate the entire processing block in which they occur (subroutine, dialog module, event block, and so on).

CONTINUE, CHECK, and EXIT can be used in all four loop types in ABAP (DO, WHILE, LOOP, and SELECT).

Terminating a Loop Pass Unconditionally

To terminate a single loop pass immediately and unconditionally, use the CONTINUE statement in the statement block of the loop.

After the statement, the system ignores any remaining statements in the current statement block, and starts the next loop pass.

```
DO 4 TIMES.
  IF SY-INDEX = 2.
    CONTINUE.
  ENDIF.
  WRITE SY-INDEX.
ENDDO.
```

The output is:

```
1             3             4
```

The second loop pass is terminated without the WRITE statement being processed.

Terminating a Loop Pass Conditionally

To terminate a single loop pass conditionally, use the CHECK <condition> statement in the statement block of the loop.

If the condition is not true, any remaining statements in the current statement block after the CHECK statement are ignored, and the next loop pass starts. <condition> can be any logical expression [Page 225].

```
DO 4 TIMES.
  CHECK SY-INDEX BETWEEN 2 and 3.
  WRITE SY-INDEX.
ENDDO.
```

The output is:

```
2             3
```

The first and fourth loop passes are terminated without the WRITE statement being processed, because SY-INDEX is not between 2 and 3.

Exiting a Loop

To terminate an entire loop immediately and unconditionally, use the EXIT statement in the statement block of the loop.

After this statement, the loop is terminated, and processing resumes after the closing statement of the loop structure (ENDDO, ENDWHILE, ENDLOOP, ENDSELECT). In nested loops, only the current loop is terminated.

```
DO 4 TIMES.
  IF SY-INDEX = 3.
    EXIT.
  ENDIF.
  WRITE SY-INDEX.
ENDDO.
```

The output is:

```
1             2
```

In the third loop pass, the loop is terminated before the WRITE statement is processed.

Processing Large Volumes of Data

There are two ways of processing large quantities of data in ABAP - either using internal tables or extract datasets.

An internal table is a dynamic sequential dataset in which all records have the same structure and a key. They are part of the ABAP type concept. You can access individual records in an internal table using either the index or the key.

Extracts are dynamic sequential datasets in which different lines can have different structures. Each ABAP program may currently only have a single extract dataset. You cannot access the individual records in an extract using key or index. Instead, you always process them using a loop.

Internal Tables [Page 251]

Extracts [Page 331]

The following section contains examples of how to process large datasets for display in a list:

Formatting Data [Page 350]

Internal tables

Internal tables provide a means of taking data from a fixed structure and storing it in working memory in ABAP. The data is stored line by line in memory, and each line has the same structure. In ABAP, internal tables fulfill the function of arrays. Since they are dynamic data objects, they save the programmer the task of dynamic memory management in his or her programs. You should use internal tables whenever you want to process a dataset with a fixed structure within a program. A particularly important use for internal tables is for storing and formatting data from a database table within a program. They are also a good way of including very complicated data structures in an ABAP program.

Like all elements in the ABAP type concept, internal tables can exist both as data types [Page 92] and as data objects [Page 118] A data type is the abstract description of an internal table, either in a program or centrally in the ABAP Dictionary, that you use to create a concrete data object. The data type is also an attribute of an existing data object.

Internal Tables as Data Types

Internal tables and structures are the two structured data types in ABAP. The data type of an internal table is fully specified by its line type, key, and table type.

Line type

The line type of an internal table can be any data type. The data type of an internal table is normally a structure. Each component of the structure is a column in the internal table. However, the line type may also be elementary or another internal table.

Key

The key identifies table rows. There are two kinds of key for internal tables - the standard key and a user-defined key. You can specify whether the key should be UNIQUE or NON-UNIQUE. Internal tables with a unique key cannot contain duplicate entries. The uniqueness depends on the table access method.

If a table has a structured line type, its default key consists of all of its non-numerical columns that are not references or themselves internal tables. If a table has an elementary line type, the default key is the entire line. The default key of an internal table whose line type is an internal table, the default key is empty.

The user-defined key can contain any columns of the internal table that are not references or themselves internal tables. Internal tables with a user-defined key are called key tables. When you define the key, the sequence of the key fields is significant. You should remember this, for example, if you intend to sort the table according to the key.

Table type

The table type determines how ABAP will access individual table entries. Internal tables can be divided into three types:

Standard tables have an internal linear index. From a particular size upwards, the indexes of internal tables are administered as trees. In this case, the index administration overhead increases in logarithmic and not linear relation to the number of lines. The system can access records either by using the table index or the key. The response time for key access is proportional to the number of entries in the table. The key of a standard table is always non-

unique. You cannot specify a unique key. This means that standard tables can always be filled very quickly, since the system does not have to check whether there are already existing entries.

Sorted tables are always saved sorted by the key. They also have an internal index. The system can access records either by using the table index or the key. The response time for key access is logarithmically proportional to the number of table entries, since the system uses a binary search. The key of a sorted table can be either unique or non-unique. When you define the table, you must specify whether the key is to be unique or not. Standard tables and sorted tables are known generically as index tables.

Hashed tables have no linear index. You can only access a hashed table using its key. The response time is independent of the number of table entries, and is constant, since the system access the table entries using a hash algorithm. The key of a hashed table must be unique. When you define the table, you must specify the key as UNIQUE.

Generic Internal Tables

Unlike other local data types in programs, you do not have to specify the data type of an internal table fully. Instead, you can specify a generic construction, that is, the key or key and line type of an internal table data type may remain unspecified. You can use generic internal tables to specify the types of field symbols [Page 201] and the interface parameters of procedures [Page 449] . You cannot use them to declare data objects.

Internal Tables as Dynamic Data Objects

Data objects that are defined either with the data type of an internal table, or directly as an internal table, are always fully defined in respect of their line type, key and access method. However, the number of lines is not fixed. Thus internal tables are dynamic data objects, since they can contain any number of lines of a particular type. The only restriction on the number of lines an internal table may contain are the limits of your system installation. The maximum memory that can be occupied by an internal table (including its internal administration) is 2 gigabytes. A more realistic figure is up to 500 megabytes. An additional restriction for hashed tables is that they may not contain more than 2 million entries. The line types of internal tables can be any ABAP data types - elementary, structured, or internal tables. The individual lines of an internal table are called table lines or table entries. Each component of a structured line is called a column in the internal table.

Choosing a Table Type

The table type (and particularly the access method) that you will use depends on how the typical internal table operations will be most frequently executed.

Standard tables

This is the most appropriate type if you are going to address the individual table entries using the index. Index access is the quickest possible access. You should fill a standard table by appending lines (ABAP APPEND statement), and read, modify and delete entries by specifying the index (INDEX option with the relevant ABAP command). The access time for a standard table increases in a linear relationship with the number of table entries. If you need key access, standard tables are particularly useful if you can fill and process the table in separate steps. For example, you could fill the table by appending entries, and then sort it. If you use the binary search option with key access, the response time is logarithmically proportional to the number of table entries.

Sorted tables

This is the most appropriate type if you need a table which is sorted as you fill it. You fill sorted tables using the INSERT statement. Entries are inserted according to the sort sequence defined through the table key. Any illegal entries are recognized as soon as you try to add them to the table. The response time for key access is logarithmically proportional to the number of table entries, since the system always uses a binary search. Sorted tables are particularly useful for partially sequential processing in a LOOP if you specify the beginning of the table key in the WHERE condition.

Hashed tables

This is the most appropriate type for any table where the main operation is key access. You cannot access a hashed table using its index. The response time for key access remains constant, regardless of the number of table entries. Like database tables, hashed tables always have a unique key. Hashed tables are useful if you want to construct and use an internal table which resembles a database table or for processing large amounts of data.

Creating Internal Tables [Page 254]

Processing Internal Tables [Page 263]

Creating Internal Tables

Like other elements in the ABAP type concept [Page 91], you can declare internal tables as abstract data types in programs or in the ABAP Dictionary, and then use them to define data objects. Alternatively, you can define them directly as data objects. When you create an internal table as a data object, you should ensure that only the administration entry which belongs to an internal table is declared statically. The minimum size of an internal table is 256 bytes. This is important if an internal table occurs as a component of an aggregated data object, since even empty internal tables within tables can lead to high memory usage. (In the next functional release, the size of the table header for an initial table will be reduced to 8 bytes). Unlike all other ABAP data objects, you do not have to specify the memory required for an internal table. Table rows are added to and deleted from the table dynamically at runtime by the various statements for adding and deleting records.

Internal Table Types [Page 255]

Internal Table Objects [Page 259]

Internal table types

This section describes how to define internal tables locally in a program. You can also define internal tables globally as data types in the ABAP Dictionary [Page 105].

Like all local data types in programs [Page 100] , you define internal tables using the TYPES statement. If you do not refer to an existing table type using the TYPE [Page 112] or LIKE addition, you can use the TYPES statement to construct a new local internal table in your program.

```
TYPES <t> TYPE|LIKE <tabkind> OF <linetype> [WITH <key>]
          [INITIAL SIZE <n>].
```

After TYPE or LIKE, there is no reference to an existing data type. Instead, the type constructor occurs:

```
<tabkind> OF <linetype> [WITH <key>]
```

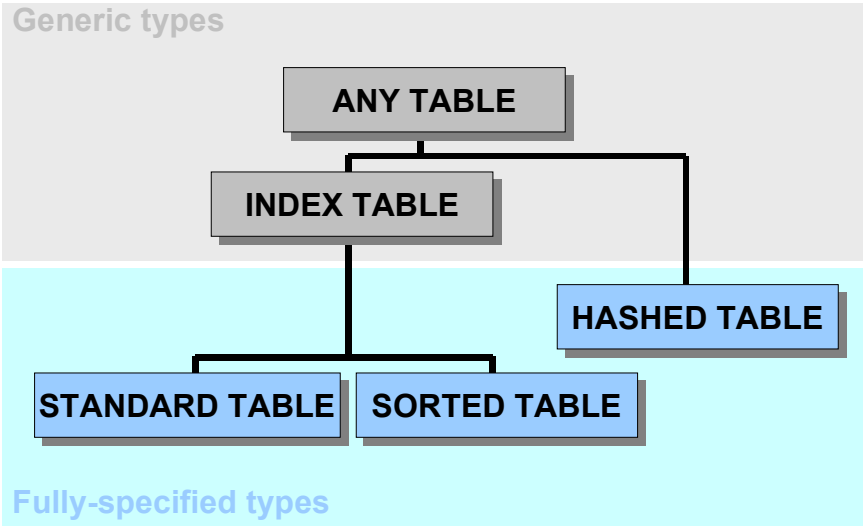
The type constructor defines the table type <tabkind>, the line type <linetype>, and the key <key> of the internal table <t>.

You can, if you wish, allocate an initial amount of memory to the internal table using the INITIAL SIZE addition.

Table type

You can specify the table type <tabkind> as follows:

Hierarchy of Table Types



Generic table types

`INDEX TABLE`

For creating a generic table type with index access.

`ANY TABLE`

Internal table types

For creating a fully-generic table type.

Data types defined using generic types can currently only be used for field symbols [Page 201] and for interface parameters in procedures [Page 449] . The generic type INDEX TABLE includes standard tables and sorted tables. These are the two table types for which index access is allowed. You cannot pass hashed tables to field symbols or interface parameters defined in this way. The generic type ANY TABLE can represent any table. You can pass tables of all three types to field symbols and interface parameters defined in this way. However, these field symbols and parameters will then only allow operations that are possible for all tables, that is, index operations are not allowed.

Fully-Specified Table Types

`STANDARD TABLE or TABLE`

For creating standard tables.

`SORTED TABLE`

For creating sorted tables.

`HASHED TABLE`

For creating hashed tables.

Fully-specified table types determine how the system will access the entries in the table in key operations. It uses a linear search for standard tables, a binary search for sorted tables, and a search using a hash algorithm for hashed tables.

Line type

For the line type <linetype>, you can specify:

- Any data type [Page 92] if you are using the TYPE addition. This can be a predefined ABAP type, a local type in the program, or a data type from the ABAP Dictionary. If you specify any of the generic elementary types C, N, P, or X, any attributes that you fail to specify (field length, number of decimal places) are automatically filled with the default values. You cannot specify any other generic types.
- Any data object [Page 118] recognized within the program at that point if you are using the LIKE addition. The line type adopts the fully-specified data type of the data object to which you refer. Except for within classes, you can still use the LIKE addition to refer to database tables and structures in the ABAP Dictionary (for compatibility reasons).

All of the lines in the internal table have the fully-specified technical attributes of the specified data type.

Key

You can specify the key <key> of an internal table as follows:

`[UNIQUE|NON-UNIQUE] KEY <col`$_1$`> ... <col`$_n$`>`

In tables with a **structured line** type, all of the components <col_i> belong to the key as long as they are not internal tables or references, and do not contain internal tables or references. Key fields can be nested structures. The substructures are expanded component by component when you access the table using the key. The system follows the sequence of the key fields.

`[UNIQUE|NON-UNIQUE] KEY TABLE LINE`

If a table has an **elementary** line type (C, D, F, I, N, P, T, X), you can define the entire line as the key. If you try this for a table whose line type is itself a table, a syntax error occurs. If a table has a structured line type, it is possible to specify the entire line as the key. However, you should remember that this is often **not** suitable.

`[UNIQUE|NON-UNIQUE] DEFAULT KEY`

This declares the fields of the default key as the key fields. If the table has a structured line type, the default key contains all non-numeric columns of the internal table that are not and do not contain references or internal tables. If the table has an elementary line type, the default key is the entire line. The default key of an internal table whose line type is an internal table, the default key is empty.

Specifying a key is optional. If you do not specify a key, the system defines a table type with an arbitrary key. You can only use this to define the types of field symbols [Page 201] and the interface parameters of procedures [Page 449] . For exceptions, refer to Special Features of Standard Tables [Page 261].

The optional additions UNIQUE or NON-UNIQUE determine whether the key is to be unique or non-unique, that is, whether the table can accept duplicate entries. If you do not specify UNIQUE or NON-UNIQUE for the key, the table type is generic in this respect. As such, it can only be used for specifying types. When you specify the table type simultaneously, you must note the following restrictions:

- You cannot use the UNIQUE addition for standard tables. The system always generates the NON-UNIQUE addition automatically.
- You must always specify the UNIQUE option when you create a hashed table.

Initial Memory Requirement

You can specify the initial amount of main memory assigned to an internal table object when you define the data type using the following addition:

`INITIAL SIZE <n>`

This size does not belong to the data type of the internal table, and does not affect the type check. You can use the above addition to reserve memory space for <n> table lines when you declare the table object.

When this initial area is full, the system makes twice as much extra space available up to a limit of 8KB. Further memory areas of 12KB each are then allocated.

You can usually leave it to the system to work out the initial memory requirement. The first time you fill the table, little memory is used. The space occupied, depending on the line width, is 16 <= <n> <= 100.

It only makes sense to specify a concrete value of <n> if you can specify a precise number of table entries when you create the table and need to allocate exactly that amount of memory (exception: Appending table lines [Page 307] to ranked lists). This can be particularly important for deep-structured internal tables where the inner table only has a few entries (less than 5, for example).

To avoid excessive requests for memory, large values of <n> are treated as follows: The largest possible value of <n> is 8KB divided by the length of the line. If you specify a larger value of <n>, the system calculates a new value so that n times the line width is around 12KB.

Examples

```
TYPES: BEGIN OF LINE,
         COLUMN1 TYPE I,
         COLUMN2 TYPE I,
         COLUMN3 TYPE I,
       END OF LINE.

TYPES ITAB TYPE SORTED TABLE OF LINE WITH UNIQUE KEY COLUMN1.
```

The program defines a table type ITAB. It is a sorted table, with line type of the structure LINE and a unique key of the component COLUMN1.

```
TYPES VECTOR TYPE HASHED TABLE OF I WITH UNIQUE KEY TABLE
LINE.

TYPES: BEGIN OF LINE,
         COLUMN1 TYPE I,
         COLUMN2 TYPE I,
         COLUMN3 TYPE I,
       END OF LINE.

TYPES ITAB TYPE SORTED TABLE OF LINE WITH UNIQUE KEY COLUMN1.

TYPES: BEGIN OF DEEPLINE,
         FIELD  TYPE C,
         TABLE1 TYPE VECTOR,
         TABLE2 TYPE ITAB,
       END OF DEEPLINE.

TYPES DEEPTABLE TYPE STANDARD TABLE OF DEEPLINE
                WITH DEFAULT KEY.
```

The program defines a table type VECTOR with type hashed table, the elementary line type I and a unique key of the entire table line. The second table type is the same as in the previous example. The structure DEEPLINE contains the internal table as a component. The table type DEEPTABLE has the line type DEEPLINE. Therefore, the elements of this internal table are themselves internal tables. The key is the default key - in this case the column FIELD. The key is non-unique, since the table is a standard table.

Internal table objects

Internal tables are dynamic variable [Page 123] data objects. Like all variables, you declare them using the DATA statement. You can also declare static internal tables in procedures [Page 449] using the STATICS statement, and static internal tables in classes using the CLASS-DATA statement. This description is restricted to the DATA statement. However, it applies equally to the STATICS and CLASS-DATA statements.

Reference to Declared Internal Table Types

Like all other data objects, you can declare internal table objects using the LIKE or TYPE addition [Page 112] of the DATA statement.

```
DATA <itab> TYPE <type>|LIKE <obj> [WITH HEADER LINE].
```

Here, the LIKE addition refers to an existing table object in the same program. The TYPE addition can refer to an internal type in the program [Page 255] declared using the TYPES statement, or a table type in the ABAP Dictionary.

You must ensure that you only refer to tables that are fully typed. Referring to generic table types (ANY TABLE, INDEX TABLE) or not specifying the key fully is not allowed (for exceptions, refer to Special Features of Standard Tables [Page 261]).

The optional addition WITH HEADER line declares an **extra data object** with the **same name** and **line type** as the internal table. This data object is known as the header line of the internal table. You use it as a work area when working with the internal table (see Using the Header Line as a Work Area [Page 328]). When you use internal tables with header lines, you must remember that the header line and the body of the table have the same name. If you have an internal table with header line and you want to address the body of the table, you must indicate this by placing brackets after the table name (<itab>[]). Otherwise, ABAP interprets the name as the name of the header line and not of the body of the table. You can avoid this potential confusion by using internal tables without header lines. In particular, internal tables nested in structures or other internal tables **must not** have a header line, since this can lead to ambiguous expressions.

```
TYPES VECTOR TYPE SORTED TABLE OF I WITH UNIQUE KEY TABLE
LINE.

DATA: ITAB TYPE VECTOR,
      JTAB LIKE ITAB WITH HEADER LINE.

* MOVE ITAB TO JTAB.   <-  Syntax error!

MOVE ITAB TO JTAB[].
```

The table object ITAB is created with reference to the table type VECTOR. The table object JTAB has the same data type as ITAB. JTAB also has a header line. In the first MOVE statement, JTAB addresses the header line. Since this has the data type I, and the table type of ITAB cannot be converted into an elementary type, the MOVE statement causes a syntax error. The second MOVE statement is correct, since both operands are table objects.

Declaring New Internal Tables

You can use the DATA statement to construct new internal tables as well as using the LIKE or TYPE addition to refer to existing types or objects. The table type that you construct does not exist in its own right; instead, it is only an attribute of the table object. You can refer to it using the LIKE addition, but not using TYPE. The syntax for constructing a table object in the DATA statement is similar to that for defining a table type in the TYPES statement.

```
DATA <itab> TYPE|LIKE <tabkind> OF <linetype> WITH <key>
            [INITIAL SIZE <n>]
            [WITH HEADER LINE].
```

As when you define a table type [Page 255] , the type constructor

```
<tabkind> OF <linetype> WITH <key>
```

defines the table type <tabkind>, the line type <linekind>, and the key <key> of the internal table <itab>. Since the technical attributes of data objects are always fully specified, the table must be fully specified in the DATA statement. You cannot create generic table types (ANY TABLE, INDEX TABLE), only fully-typed tables (STANDARD TABLE, SORTED TABLE, HASHED TABLE). You must also specify the key and whether it is to be unique (for exceptions, refer to Special Features of Standard Tables [Page 261]).

As in the TYPES statement, you can, if you wish, allocate an initial amount of memory to the internal table using the INITIAL SIZE addition. You can create an internal table with a header line using the WITH HEADER LINE addition. The header line is created under the same conditions as apply when you refer to an existing table type.

```
DATA ITAB TYPE HASHED TABLE OF SPFLI
          WITH UNIQUE KEY CARRID CONNID.
```

The table object ITAB has the type hashed table, a line type corresponding to the flat structure SPFLI from the ABAP Dictionary, and a unique key with the key fields CARRID and CONNID. The internal table ITAB can be regarded as an internal template for the database table SPFLI. It is therefore particularly suitable for working with data from this database table as long as you only access it using the key.

Special Features of Standard Tables

Unlike sorted tables, hashed tables, and key access to internal tables, which were only introduced in Release 4.0, standard tables already existed several releases previously. Defining a line type, table type, and tables without a header line have only been possible since Release 3.0. For this reason, there are certain features of standard tables that still exist for compatibility reasons.

Standard Tables Before Release 3.0

Before Release 3.0, internal tables all had header lines and a flat-structured line type. There were no independent table types. You could only create a table object using the OCCURS addition in the DATA statement, followed by a declaration of a flat structure:

```
DATA: BEGIN OF <itab> OCCURS <n>,
        ...
        <f_i> ...,
        ...
      END OF <itab>.
```

This statement declared an internal table <itab> with the line type defined following the OCCURS addition. Furthermore, all internal tables had header lines.

The number <n> in the OCCURS addition had the same meaning as in the INITIAL SIZE addition from Release 4.0. Entering '0' had the same effect as omitting the INITIAL SIZE addition. In this case, the initial size of the table is determined by the system.

The above statement is still possible in Release 4.0, and has roughly the same function as the following statements:

```
TYPES: BEGIN OF <itab>,
         ...
         <f_i> ...,
         ...
       END OF <itab>.

DATA <itab> TYPE STANDARD TABLE OF <itab>
                 WITH NON-UNIQUE DEFAULT KEY
                 INITIAL SIZE <n>
                 WITH HEADER LINE.
```

In the original statement, no independent data type <itab> is created. Instead, the line type only exists as an attribute of the data object <itab>.

Standard Tables From Release 3.0

Since Release 3.0, it has been possible to create table types using

```
TYPES <t> TYPE|LIKE <linetype> OCCURS <n>.
```

and table objects using

```
DATA <itab> TYPE|LIKE <linetype> OCCURS <n> [WITH HEADER LINE].
```

The effect of the OCCURS addition is to construct a standard table with the data type <linetype>. The line type can be any data type. The number <n> in the OCCURS addition has the same meaning as before Release 3.0. Before Release 4.0, the key of an internal table was always the default key, that is, all non-numeric fields that were not themselves internal tables.

Special Features of Standard Tables

The above statements are still possible in Release 4.0, and have the same function as the following statements:

```
TYPES|DATA <itab> TYPE|LIKE STANDARD TABLE OF <linetype>
                           WITH NON-UNIQUE DEFAULT KEY
                           INITIAL SIZE <n>
                           [WITH HEADER LINE].
```

They can also be replaced by the following statements:

Standard Tables From Release 4.0

When you create a standard table, you can use the following forms of the TYPES and DATA statements. The addition INITIAL SIZE is also possible in all of the statements. The addition WITH HEADER LINE is possible in the DATA statement.

Standard Table Types

Generic Standard Table Type:

```
TYPES <itab> TYPE|LIKE [STANDARD] TABLE OF <linetype>.
```

The table key is not defined.

Fully-Specified Standard Table Type:

```
TYPES <itab> TYPE|LIKE [STANDARD] TABLE OF <linetype>
                       WITH [NON-UNIQUE] <key>.
```

The key of a fully-specified standard table is always non-unique.

Standard Table Objects

Short Forms of the DATA Statement

```
DATA <itab> TYPE|LIKE [STANDARD] TABLE OF <linetype>.
```

```
DATA <itab> TYPE|LIKE [STANDARD] TABLE OF <linetype>
                      WITH DEFAULT KEY.
```

Both of these DATA statements are automatically completed by the system as follows:

```
DATA <itab> TYPE|LIKE STANDARD TABLE OF <linetype>
                      WITH NON-UNIQUE DEFAULT KEY.
```

The purpose of the shortened forms of the DATA statement is to keep the declaration of standard tables, which are compatible with internal tables from previous releases, as simple as possible. When you declare a standard table with reference to the above type, the system automatically adopts the default key as the table key.

Fully-Specified Standard Tables:

```
DATA <itab> TYPE|LIKE [STANDARD] TABLE OF <linetype>
                       WITH [NON-UNIQUE] <key>.
```

The key of a standard table is always non-unique.

Processing Internal Tables

When you process an internal table object, you must distinguish between the following two cases:

Operations on Entire Internal Tables [Page 264]

Operations on Individual Lines [Page 278]

Operations on Entire Internal Tables

When you access the entire internal table, you address the body of the table as a single data object. The following operations on the body of an internal table are relevant:

Assigning Internal Tables [Page 265]

Initialize Internal Tables [Page 267]

Comparing Internal Tables [Page 269]

Sorting Internal Tables [Page 271]

Internal Tables as Interface Parameters [Page 276]

Determining the Attributes of Internal Tables [Page 277]

You can also address internal tables in Open SQL statements - either to fill an internal table from a database table or the other way round. For further information, refer to Reading and Processing Database Tables [Page 1041].

Assigning Internal Tables

Like other data objects, you can make internal tables operands in the MOVE statement [Page 145]

```
MOVE <itab1> TO <itab2>.
```

or the equivalent statement

```
<itab2> = <itab1>.
```

Both operands must either be compatible or convertible [Page 194]. These statements assign the entire contents of table <itab1> to table <itab2>, including the data in any nested internal tables. The original contents of the target table are overwritten.

If you are using internal tables with header lines, remember that the header line and the body of the table have the same name. If you want to address the body of the table in an assignment, you must place two brackets ([ ]) after the table name.

```
DATA: BEGIN OF LINE,
        COL1,
        COL2,
      END OF LINE.

DATA: ETAB LIKE TABLE OF LINE WITH HEADER LINE,
      FTAB LIKE TABLE OF LINE.

LINE-COL1 = 'A'. LINE-COL2 = 'B'.

APPEND LINE TO ETAB.

MOVE ETAB[] TO FTAB.

LOOP AT FTAB INTO LINE.
  WRITE: / LINE-COL1, LINE-COL2.
ENDLOOP.
```

The output is:

```
A B
```

The example creates two standard tables ETAB and FTAB with the line type of the structure LINE. ETAB has a header line. After filling ETAB line by line using the APPEND statement, its entire contents are assigned to FTAB. Note the brackets in the statement.

```
DATA: FTAB TYPE SORTED TABLE OF F
           WITH NON-UNIQUE KEY TABLE LINE,
      ITAB TYPE HASHED TABLE OF I
           WITH UNIQUE KEY TABLE LINE,
      FL   TYPE F.

DO 3 TIMES.
  INSERT SY-INDEX INTO TABLE ITAB.
ENDDO.
```

```
FTAB = ITAB.

LOOP AT FTAB INTO FL.
  WRITE: / FL.
ENDLOOP.
```

The output is:

```
 1.000000000000000E+00

 2.000000000000000E+00

 3.000000000000000E+00
```

FTAB is a sorted table with line type F and a non-unique key. ITAB is a hashed table with line type I and a unique key. The line types, and therefore the entire tables, are convertible. It is therefore possible to assign the contents of ITAB to FTAB. When you assign the unsorted table ITAB to the sorted table FTAB, the contents are automatically sorted by the key of FTAB.

```
DATA: BEGIN OF ILINE,
        NUM TYPE I,
      END OF ILINE,

      BEGIN OF FLINE,
        NUM TYPE F,
      END OF FLINE,

      ITAB LIKE TABLE OF ILINE,
      FTAB LIKE TABLE OF FLINE.

DO 3 TIMES.
  ILINE-NUM = SY-INDEX.
  APPEND ILINE-NUM TO ITAB.
ENDDO.

FTAB = ITAB.

LOOP AT FTAB INTO FLINE.
  WRITE: / FLINE-NUM.
ENDLOOP.
```

The output might look like this:

```
6.03823403895813E-154

6.03969074613219E-154

6.04114745330626E-154
```

Here, the line types of the internal tables ITAB and FTAB are structures each with one component of type I or F. The line types are convertible, but not compatible. Therefore, when assigning ITAB to FTAB, the contents of Table ITAB are converted [Page 192] to type C fields and then written to FTAB. The system interprets the transferred data as type F fields, and obtains meaningless results.

Initializing Internal Tables

initiale Speicheranforderung [Page 255]

Like all data objects, you can initialize internal tables with the

```
CLEAR <itab>.
```

statement. This statement restores an internal table to the state it was in immediately after you declared it. This means that the table contains no lines. However, the memory already occupied by the memory up until you cleared it remains allocated to the table.

If you are using internal tables with header lines, remember that the header line and the body of the table have the same name. If you want to address the body of the table in a comparison, you must place two brackets ([ ]) after the table name.

```
CLEAR <itab>[].
```

To ensure that the table itself has been initialized, you can use the

```
REFRESH <itab>.
```

statement. This always applies to the body of the table. As with the CLEAR statement, the memory used by the table before you initialized it remains allocated. To release the memory space, use the statement

```
FREE <itab>.
```

You can use FREE to initialize an internal table and release its memory space without first using the REFRESH or CLEAR statement. Like REFRESH, FREE works on the table body, not on the table work area. After a FREE statement, you can address the internal table again. It still occupies the amount of memory required for its header (currently 256 bytes). When you refill the table, the system has to allocate new memory space to the lines.

```
DATA: BEGIN OF LINE,
         COL1,
         COL2,
      END OF LINE.

DATA ITAB LIKE TABLE OF LINE.

LINE-COL1 = 'A'. LINE-COL2 = 'B'.

APPEND LINE TO ITAB.

REFRESH ITAB.

IF ITAB IS INITIAL.
  WRITE 'ITAB is empty'.
  FREE ITAB.
ENDIF.
```

The output is:

```
ITAB is empty.
```

In this program, an internal table ITAB is filled and then initialized with REFRESH. The IF statement uses the expression ITAB IS INITIAL to find out whether ITAB is empty. If so, the memory is released.

Comparing Internal Tables

Like other data objects, you can use internal tables as operands in logical expressions [Page 225].

```
.... <itab1> <operator> <itab2> ...
```

For <operator>, all operators listed in the table in Comparisons Between Data Types [Page 226] can be used (EQ, =, NE, <>, ><, GE, >=, LE, <=, GT, >, LT, <).

If you are using internal tables with header lines, remember that the header line and the body of the table have the same name. If you want to address the body of the table in a comparison, you must place two brackets ([ ]) after the table name.

The first criterion for comparing internal tables is the number of lines they contain. The more lines an internal table contains, the larger it is. If two internal tables contain the same number of lines, they are compared line by line, component by component. If components of the table lines are themselves internal tables, they are compared recursively. If you are testing internal tables for anything other than equality, the comparison stops when it reaches the first pair of components that are unequal, and returns the corresponding result.

```
DATA: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF LINE.

DATA: ITAB LIKE TABLE OF LINE,
      JTAB LIKE TABLE OF LINE.

DO 3 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

MOVE ITAB TO JTAB.

LINE-COL1 = 10. LINE-COL2 = 20.
APPEND LINE TO ITAB.

IF ITAB GT JTAB.
  WRITE / 'ITAB GT JTAB'.
ENDIF.

APPEND LINE TO JTAB.

IF ITAB EQ JTAB.
  WRITE / 'ITAB EQ JTAB'.
ENDIF.

LINE-COL1 = 30. LINE-COL2 = 80.
APPEND LINE TO ITAB.

IF JTAB LE ITAB.
  WRITE / 'JTAB LE ITAB'.
ENDIF.
```

```
LINE-COL1 = 50. LINE-COL2 = 60.
APPEND LINE TO JTAB.

IF ITAB NE JTAB.
  WRITE / 'ITAB NE JTAB'.
ENDIF.

IF ITAB LT JTAB.
  WRITE / 'ITAB LT JTAB'.
ENDIF.
```

The output is:

```
ITAB GT JTAB

ITAB EQ JTAB

JTAB LE ITAB

ITAB NE JTAB

ITAB LT JTAB
```

This example creates two standard tables, ITAB and JTAB. ITAB is filled with 3 lines and copied to JTAB. Then, another line is appended to ITAB and the first logical expression tests whether ITAB is greater than JTAB. After appending the same line to JTAB, the second logical expression tests whether both tables are equal. Then, another line is appended to ITAB and the third logical expressions tests whether JTAB is less than or equal to ITAB. Next, another line is appended to JTAB. Its contents are unequal to the contents of the last line of ITAB. The next logical expressions test whether ITAB is not equal to JTAB. The first table field whose contents are different in ITAB and JTAB is COL1 in the last line of the table: 30 in ITAB and 50 in JTAB. Therefore, in the last logical expression, ITAB is less than JTAB.

Sorting Internal Tables

You can sort a standard or hashed table in a program. To sort a table by its key, use the statement

```
SORT <itab> [ASCENDING|DESCENDING] [AS TEXT] [STABLE].
```

The statement sorts the internal table <itab> in ascending order by its key. The statement always applies to the table itself, not to the header line. The sort order depends on the sequence of the standard key fields in the internal table. The default key is made up of the non-numeric fields of the table line in the order in which they occur.

You can specify the direction of the sort using the additions ASCENDING and DESCENDING. The default is ascending.

The larger the sort key, the more time the system needs to sort the table. If the sort key contains an internal table, the sorting process may be slowed down considerably.

You cannot sort a sorted table using the SORT statement. The system always maintains these tables automatically by their sort order. If an internal table is statically recognizable as a sorted table, the SORT statement causes a syntax error. If the table is a generic sorted table, the SORT statement causes a runtime error if the sort key is not the same as an extract of the beginning of the table key, you sort in descending order, or use the AS TEXT addition. In other words, the SORT statement is only allowed for generic internal tables, if it does not violate the internal sort order.

Sorting by Another Sort Key

If you have an internal table with a structured line type that you want sort by a different key, you can specify the key in the SORT statement:

```
SORT <itab> [ASCENDING|DESCENDING] [AS TEXT] [STABLE]
            BY <f1> [ASCENDING|DESCENDING] [AS TEXT]
               ...
               <fn> [ASCENDING|DESCENDING] [AS TEXT].
```

The table is now sorted by the specified components <f_1> ... <f_n> instead of by the table key. The number of sort fields is limited to 250. The sort order depends on the sequence of the fields <f_i>. The sort sequence specified before BY applies to all fields. The sort sequence after a field applies only to that column of the table.

You can specify a sort field dynamically by specifying (<f>) instead of <f_i>. The contents of the field <f> determines the name of the sort field. If <f> is empty when the statement is executed, the field is ignored in the sort. If it contains an invalid component name, a runtime error occurs.

Sorting alphabetically

As well as the ASCENDING or DESCENDING addition, you can also specify that the entire sort or each sort field should be alphabetical.

```
SORT <itab> ... AS TEXT ... .
```

This addition affects the sort method for strings. Without the addition, strings are sorted according to the sequence specified by the hardware platform. With the option AS TEXT, the system sorts character fields alphabetically according to the current text environment. By default,

the text environment is set in the user master record. However, you can also set it specifically using the statement

```
SET LOCALE LANGUAGE
```

The AS TEXT addition saves you having to convert strings into a sortable format [Page 168]. Such a conversion is only necessary if you want to

- Sort an internal table alphabetically and then use a binary search
- resort an internal table with character fields as its key several times, since only one conversion is then required
- Construct an alphabetical index for database tables in your program

If the AS TEXT addition is applied to the entire sort, it only affects sort fields with type C. If you apply the AS TEXT addition to a single sort field, it must have type C.

Stable sort

The option

```
SORT <itab> ... STABLE.
```

allows you to perform a stable sort, that is, the relative sequence of lines that are unchanged by the sort is not changed. If you do not use the STABLE option, the sort sequence is not preserved. If you sort a table several times by the same key, the sequence of the table entries will change in each sort. However, a stable sort takes longer than an unstable sort.

Examples

```
DATA: BEGIN OF LINE,
        LAND(3)   TYPE C,
        NAME(10)  TYPE C,
        AGE       TYPE I,
        WEIGHT    TYPE P DECIMALS 2,
      END OF LINE.

DATA ITAB LIKE STANDARD TABLE OF LINE WITH NON-UNIQUE KEY
LAND.

LINE-LAND = 'G'.   LINE-NAME   = 'Hans'.
LINE-AGE  = 20.    LINE-WEIGHT = '80.00'.
APPEND LINE TO ITAB.

LINE-LAND = 'USA'. LINE-NAME   = 'Nancy'.
LINE-AGE  = 35.    LINE-WEIGHT = '45.00'.
APPEND LINE TO ITAB.

LINE-LAND = 'USA'. LINE-NAME   = 'Howard'.
LINE-AGE  = 40.    LINE-WEIGHT = '95.00'.
APPEND LINE TO ITAB.

LINE-LAND = 'GB'.  LINE-NAME   = 'Jenny'.
LINE-AGE  = 18.    LINE-WEIGHT = '50.00'.
APPEND LINE TO ITAB.
```

```
LINE-LAND = 'F'.   LINE-NAME   = 'Michele'.
LINE-AGE  = 30.    LINE-WEIGHT = '60.00'.
APPEND LINE TO ITAB.

LINE-LAND = 'G'.   LINE-NAME   = 'Karl'.
LINE-AGE  = 60.    LINE-WEIGHT = '75.00'.
APPEND LINE TO ITAB.

PERFORM LOOP_AT_ITAB.

SORT ITAB.
PERFORM LOOP_AT_ITAB.

SORT ITAB.
PERFORM LOOP_AT_ITAB.

SORT ITAB STABLE.
PERFORM LOOP_AT_ITAB.

SORT ITAB DESCENDING BY LAND WEIGHT ASCENDING.
PERFORM LOOP_AT_ITAB.

FORM LOOP_AT_ITAB.
  LOOP AT ITAB INTO LINE.
    WRITE: / LINE-LAND, LINE-NAME, LINE-AGE, LINE-WEIGHT.
  ENDLOOP.
  SKIP.
ENDFORM.
```

The output is:

```
G   Hans              20         80.00
USA Nancy             35         45.00
USA Howard            40         95.00
GB  Jenny             18         50.00
F   Michele           30         60.00
G   Karl              60         75.00

F   Michele           30         60.00
G   Hans              20         80.00
G   Karl              60         75.00
GB  Jenny             18         50.00
USA Howard            40         95.00
USA Nancy             35         45.00

F   Michele           30         60.00
G   Karl              60         75.00
G   Hans              20         80.00
GB  Jenny             18         50.00
USA Howard            40         95.00
USA Nancy             35         45.00

F   Michele           30         60.00
G   Karl              60         75.00
G   Hans              20         80.00
GB  Jenny             18         50.00
USA Howard            40         95.00
USA Nancy             35         45.00
```

```
USA Nancy              35         45.00
USA Howard             40         95.00
GB  Jenny              18         50.00
G   Karl               60         75.00
G   Hans               20         80.00
F   Michele            30         60.00
```

The program sorts a standard table with one key field four times. First, the table is sorted twice by the key field (LAND) without the STABLE addition. The sort is unstable. The sequence of the second and third lines changes. The same sort is then performed using the STABLE addition. The sort is stable. The lines remain in the same sequence. Then, it is sorted by a sort key defined as LAND and WEIGHT. The general sort order is defined as descending, but for WEIGHT it is defined as ascending.

```
DATA: BEGIN OF LINE,
        TEXT(6),
        XTEXT(160) TYPE X,
      END OF LINE.

DATA ITAB LIKE HASHED TABLE OF LINE WITH UNIQUE KEY TEXT.

LINE-TEXT = 'Muller'.
CONVERT TEXT LINE-TEXT INTO SORTABLE CODE LINE-XTEXT.
INSERT LINE INTO TABLE ITAB.

LINE-TEXT = 'Möller'.
CONVERT TEXT LINE-TEXT INTO SORTABLE CODE LINE-XTEXT.
INSERT LINE INTO TABLE ITAB.

LINE-TEXT = 'Moller'.
CONVERT TEXT LINE-TEXT INTO SORTABLE CODE LINE-XTEXT.
INSERT LINE INTO TABLE ITAB.

LINE-TEXT = 'Miller'.
CONVERT TEXT LINE-TEXT INTO SORTABLE CODE LINE-XTEXT.
INSERT LINE INTO TABLE ITAB.

SORT ITAB.
PERFORM LOOP_AT_ITAB.

SORT ITAB BY XTEXT.
PERFORM LOOP_AT_ITAB.

SORT ITAB AS TEXT.
PERFORM LOOP_AT_ITAB.

FORM LOOP_AT_ITAB.
  LOOP AT ITAB INTO LINE.
    WRITE / LINE-TEXT.
  ENDLOOP.
  SKIP.
ENDFORM.
```

This example demonstrates alphabetical sorting of character fields. The internal table ITAB contains a column with character fields and a column with corresponding binary codes that are alphabetically sortable. The binary codes are created with the

CONVERT statement (see Converting to a Sortable Format [Page 168]). The table is sorted three times. First, it is sorted binarily by the TEXT field. Second, it is sorted binarily by the XTEXT field. Third, it is sorted alphabetically by the TEXT field. Since there is no directly corresponding case in English, we have taken the results from a German text environment:

```
Miller
Moller
Muller
Möller

Miller
Moller
Möller
Muller

Miller
Moller
Möller
Muller
```

After the first sorting, 'Möller' follows behind 'Muller' since the internal code for the letter 'ö' comes after the code for 'u'. The other two sorts are alphabetical. The binary sort by XTEXT has the same result as the alphabetical sorting by the field TEXT.

Internal Tables as Interface Parameters

Like other data objects, you can pass internal tables by value or reference to parameter interfaces [Page 459] of procedures [Page 449]. If an internal table has a header line, you must indicate that you want to address the body of the table by placing two brackets ([ ]) after the table name.

You can define the formal parameters of the parameter interfaces of procedures as internal tables. When you do this, you can use both the predefined generic types in the TYPE addition [Page 112] and the generic internal table types [Page 255].

To ensure compatibility with previous releases, you can also specify formal parameters in subroutines and function modules as TABLES parameters. This defines a formal parameter as a standard table with default key and header line. Whenever you pass a table without a header line as an actual parameter to a formal parameter with a header line (TABLES), the system automatically creates the corresponding header line in the routine.

Determining the Attributes of Internal Tables

To find out the attributes of an internal table at runtime that were not available statically, use the statement:

```
DESCRIBE TABLE <itab> [LINES <l>] [OCCURS <n>] [KIND <k>].
```

If you use the LINES parameter, the number of filled lines is written to the variable <lin>. If you use the OCCURS parameter, the value of the INITIAL SIZE of the table is returned to the variable <n>. If you use the KIND parameter, the table type is returned to the variable <k>: ‘T’ for standard table, ‘S’ for sorted table, and ‘H’ for hashed table.

```
DATA: BEGIN OF LINE,
         COL1 TYPE I,
         COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE HASHED TABLE OF LINE WITH UNIQUE KEY COL1
                                    INITIAL SIZE 10.

DATA: LIN TYPE I,
      INI TYPE I,
      KND TYPE C.

DESCRIBE TABLE ITAB LINES LIN OCCURS INI KIND KND.
WRITE: / LIN, INI, KND.

DO 1000 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  INSERT LINE INTO TABLE ITAB.
ENDDO.

DESCRIBE TABLE ITAB LINES LIN OCCURS INI KIND KND.
WRITE: / LIN, INI, KND.
```

The output is:

```
          0         10  H
      1,000         10  H
```

Here, a hashed table ITAB is created and filled. The DESCRIBE TABLE statement is processed before and after the table is filled. The current number of lines changes, but the number of initial lines cannot change.

Operations on Individual Lines

The following are typical operations involving single lines of a table:

- Filling a table line by line
- Reading a table line by line
- Modifying individual lines
- Deleting individual lines

Influence of the Table Type

When working with single table lines, we must distinguish between the operators that are possible for all table types, and those that are only possible with index tables. Index tables (standard and sorted tables) have an internal index, making linear access possible. Hashed tables have no linear index. Consequently, only key access is possible. The operations that are permitted for all table types do not use indexes. They can also be used in procedures [Page 449] or with field symbols [Page 201] , where the internal table type is not fully typed. These operations are known as generic operations.

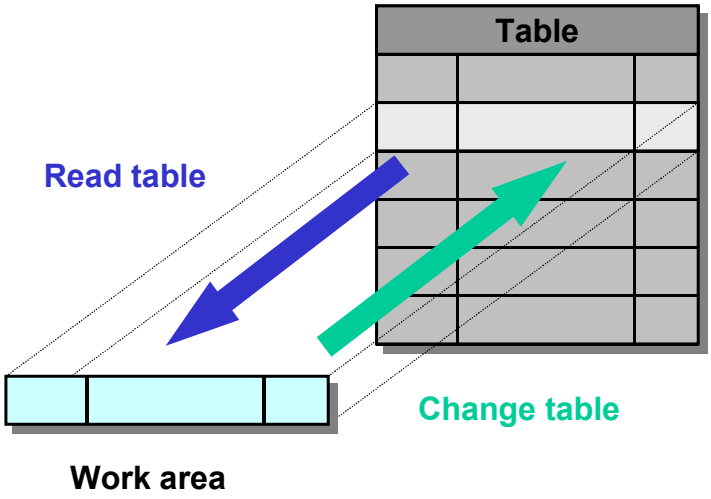
The statements used to access lines of any type of table differ from those used to access index tables mainly through the TABLE addition following the corresponding keyword. For example, you would use MODIFY to change lines in index tables, but MODIFY TABLE to change lines in any type of table.

Access methods

There are two ways to access a single table entry:

Access Using a Work Area

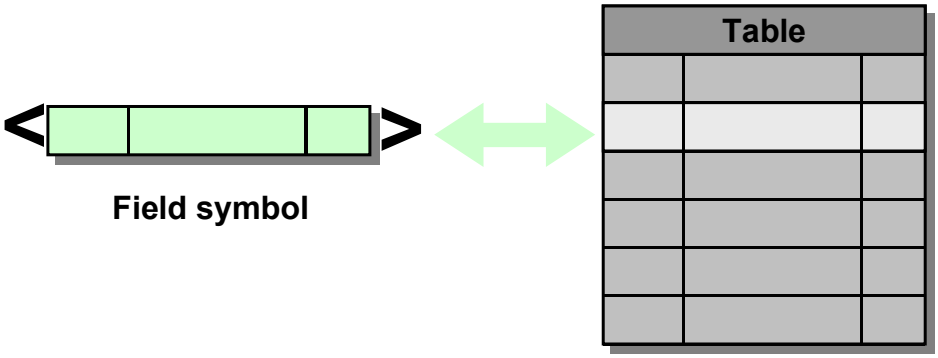
When you access individual table entries using a work area, you are not working directly with the data in the table. Instead, you work with another data object as a work area. The work area is an interface to the entries in the internal table, and must be convertible into the line type of the internal table. The most efficient working method is to use a work area compatible with the line type of the internal table. When you read data from a table record, the data you are reading overwrites the current contents of the work area. You can then use this data in the program. When you write data to the internal table, this must first be placed in the work area. The system then transfers it from the work area to the appropriate table entry. Data transfer follows the rules of assigning data using MOVE [Page 145].



If the internal table has a header line, you can use it as the work area. The ABAP statements that you use to access individual table entries can use the header line implicitly as a work area.

Access Using Field Symbols

If you access an internal table using a field symbol [Page 201] , you do not need to copy the data into a work area. You can assign a line of an internal table to a field symbol. Ideally, the field symbol will have the same type as the line type of the internal table. Once you have assigned the entry to the field symbol, working with the field symbol has exactly the same effect as accessing the corresponding line directly.



The following sections discuss operations for the different table types. After that comes a short section about accessing internal tables using field symbols and header lines.

Operations for all table types [Page 281]

Operations on Index Tables [Page 306]

Access Using Field Symbols [Page 326]

Operations on Individual Lines

Using Header Lines as Work Areas [Page 328]

Operations for all Table Types

The operations listed in this section can be applied to any table type. They are listed for each table type individually. You should use these operations if they are the only possibility for the table type, or when the table type is not known when you write the program (for example, generic formal parameters in procedures).

If you know the table type, you should, for performance reasons, use the corresponding specific operation. For example, you should use the APPEND ... TO statement to fill index tables, but INSERT ... INTO TABLE to fill hashed or generic tables.

Inserting Lines into Tables

You can insert lines into internal tables either singly or in groups:

Inserting a Single Line

To add a line to an internal table, use the statement:

```
INSERT <line> INTO TABLE <itab>.
```

<line> is either a work area that is compatible with the line type, or the expression INITIAL LINE. The work are must be compatible [Page 133] because the fields in the table key must be filled from fields of the correct type. INITIAL LINE inserts a blank line containing the correct initial value for each field of the structure.

If the table has a unique key and you attempt to insert lines whose key already exists in the table, the system does not add the line to the table, and sets SY-SUBRC to 4. When the system successfully adds a line to the table, SY-SUBRC is set to 0.

Lines are added to internal tables as follows:

- Standard tables

  The line is appended [Page 307] to the end of the internal table. This has the same effect as the explicit APPEND statement.

- Sorted tables

  The line is inserted into the table according to the table key. If the key is non-unique, duplicates are inserted above the existing entry with the same key. The runtime for the operation increases logarithmically with the number of existing table entries.

- Hashed tables

  The table is inserted into the internal hash administration according to the table key.

Inserting Several Lines

To add several lines to an internal table, use the statement:

```
INSERT LINES OF <itab1> [FROM <n1>] [TO <n2>] INTO TABLE <itab2>.
```

<itab1> and <itab2> are tables with a **compatible** line type. The system inserts the lines of table <itab1> one by one into <itab2> using the same rules as for single lines.

If <itab1> is an index table, you can specify the first and last lines of the table that you want to append in $<n_1>$ and $<n_2>$.

Depending on the size of the tables and where they are inserted, this method of inserting lines of one table into another can be up to 20 times faster than inserting them line by line in a loop.

Examples

```
DATA: BEGIN OF LINE,
        LAND(3)   TYPE C,
        NAME(10)  TYPE C,
        AGE       TYPE I,
```

```
        WEIGHT   TYPE P DECIMALS 2,
      END OF LINE.

DATA ITAB LIKE SORTED TABLE OF LINE
          WITH NON-UNIQUE KEY LAND NAME AGE WEIGHT.

LINE-LAND = 'G'.   LINE-NAME   = 'Hans'.
LINE-AGE  = 20.    LINE-WEIGHT = '80.00'.
INSERT LINE INTO TABLE ITAB.

LINE-LAND = 'USA'. LINE-NAME   = 'Nancy'.
LINE-AGE  = 35.    LINE-WEIGHT = '45.00'.
INSERT LINE INTO TABLE ITAB.

LINE-LAND = 'USA'. LINE-NAME   = 'Howard'.
LINE-AGE  = 40.    LINE-WEIGHT = '95.00'.
INSERT LINE INTO TABLE ITAB.

LINE-LAND = 'GB'.  LINE-NAME   = 'Jenny'.
LINE-AGE  = 18.    LINE-WEIGHT = '50.00'.
INSERT LINE INTO TABLE ITAB.

LINE-LAND = 'F'.   LINE-NAME   = 'Michele'.
LINE-AGE  = 30.    LINE-WEIGHT = '60.00'.
INSERT LINE INTO TABLE ITAB.

LINE-LAND = 'G'.   LINE-NAME   = 'Karl'.
LINE-AGE  = 60.    LINE-WEIGHT = '75.00'.
INSERT LINE INTO TABLE ITAB.

LOOP AT ITAB INTO LINE.
  WRITE: / LINE-LAND, LINE-NAME, LINE-AGE, LINE-WEIGHT.
ENDLOOP.
```

The output is:

```
F   Michele            30         60.00
G   Hans               20         80.00
G   Karl               60         75.00
GB  Jenny              18         50.00
USA Howard             40         95.00
USA Nancy              35         45.00
```

The example fills a sorted internal table with six entries.

```
DATA: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF LINE.

DATA: ITAB LIKE STANDARD TABLE OF LINE,
      JTAB LIKE SORTED TABLE OF LINE
           WITH NON-UNIQUE KEY COL1 COL2.

DO 3 TIMES.
  LINE-COL1 = SY-INDEX. LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
  LINE-COL1 = SY-INDEX. LINE-COL2 = SY-INDEX ** 3.
```

Inserting Lines into Tables

```
  APPEND LINE TO JTAB.
ENDDO.

INSERT LINES OF ITAB INTO TABLE JTAB.

LOOP AT JTAB INTO LINE.
  WRITE: / SY-TABIX, LINE-COL1, LINE-COL2.
ENDLOOP.
```

The output is: :

```
1          1          1
2          1          1
3          2          4
4          2          8
5          3          9
6          3         27
```

The example creates two internal tables with the same line type but different table types. Each is filled with three lines. Then, ITAB is sorted into the sorted table JTAB.

Appending Summarized Lines

The following statement allows you to summate entries in an internal table:

```
COLLECT <wa> INTO <itab>.
```

<itab> must have a flat line type, and all of the fields that are not part of the table key must have a **numeric type** (F, I, or P). You specify the line that you want to add in a work area that is compatible with the line type.

When the line is inserted, the system checks whether there is already a table entry that matches the key. If there is no corresponding entry already in the table, the COLLECT statement has the same effect as inserting the new line [Page 282]. If an entry with the same key already exists, the COLLECT statement does not append a new line, but adds the contents of the numeric fields in the work area to the contents of the numeric fields in the existing entry.

You should only use the COLLECT statement if you want to create summarized tables. If you use other statements to insert table entries, you may end up with duplicate entries.

Lines are added to internal tables as follows:

- Standard tables

  If the COLLECT statement is the first statement to fill the standard table, the system creates a temporary hash administration that identifies existing entries in the table. The hash administration is retained until another statement changes the contents of key fields or changes the sequence of the lines in the internal table. After this, the system finds existing entries using a linear search. The runtime for this operation increases in linear relation to the number of existing table entries. The system field SY-TABIX contains the index of the line inserted or modified in the COLLECT statement.

- Sorted tables

  The system uses a binary search to locate existing lines. The runtime for the operation increases logarithmically with the number of existing lines. The system field SY-TABIX contains the index of the line inserted or modified in the COLLECT statement.

- Hashed tables

  The system finds existing lines using the hash algorithm of the internal table. After the COLLECT statement, the system field SY-TABIX has the value 0, since hashed tables have no linear index.

Example

```
DATA: BEGIN OF LINE,
        COL1(3) TYPE C,
        COL2(2) TYPE N,
        COL3    TYPE I,
      END OF LINE.

DATA ITAB LIKE SORTED TABLE OF LINE
          WITH NON-UNIQUE KEY COL1 COL2.
```

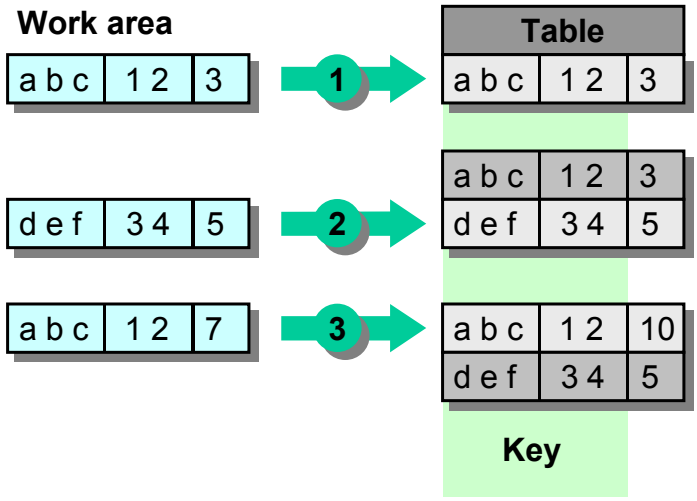
Appending Summarized Lines

```
LINE-COL1 = 'abc'. LINE-COL2 = '12'. LINE-COL3 = 3.
COLLECT LINE INTO ITAB.
WRITE / SY-TABIX.

LINE-COL1 = 'def'. LINE-COL2 = '34'. LINE-COL3 = 5.
COLLECT LINE INTO ITAB.
WRITE / SY-TABIX.

LINE-COL1 = 'abc'. LINE-COL2 = '12'. LINE-COL3 = 7.
COLLECT LINE INTO ITAB.
WRITE / SY-TABIX.

LOOP AT ITAB INTO LINE.
  WRITE: / LINE-COL1, LINE-COL2, LINE-COL3.
ENDLOOP.
```

The output is:

```
         1

         2

         1

abc 12          10

def 34           5
```

The example fills a sorted table. The first two COLLECT statements work like normal insertion statements. In the third COLLECT statement, the first line of ITAB is modified. The following diagram shows the three steps:



Reading Lines of Tables

To read a single line of any table, use the statement:

```
READ TABLE <itab> <key> <result>.
```

For the statement to be valid for any kind of table, you must specify the entry using the key and not the index. You specify the key in the <key> part of the statement. The <result> part can specify a further processing option for the line that is retrieved.

If the system finds an entry, it sets SY-SUBRC to zero, if not, it takes the value 4, as long as it is not influenced by one of the possible additions. If the internal table is an index table, SY-TABIX is set to the index of the line retrieved. If the table has a non-unique key and there are duplicate entries, the first entry is read.

Specifying the Search Key

The search key may be either the table key or another key.

Using the Table Key

To use the table key of <itab> as a search key, enter <key> as follows:

```
READ TABLE <itab> FROM <wa> <result>.
```

or as follows

```
READ TABLE <itab> WITH TABLE KEY <k1> = <f1> ... <kn> = <fn> <result>.
```

In the first case, <wa> must be a work area **compatible** with the line type of <itab>. The values of the key fields are taken from the corresponding components of the work area.

In the second case, you have to supply the values of each key field explicitly. If you do not know the name of one of the key fields until runtime, you can specify it as the content of a field $<n_i>$ using the form $(<n_i>) = <f_i>$. If the data types of $<f_i>$ are not compatible with the key fields, the system converts them.

The system searches for the relevant lines as follows:

- Standard tables

  Linear search, where the runtime is in linear relation to the number of table entries.

- Sorted tables

  Binary search, where the runtime is in logarithmic relation to the number of table entries.

- Hashed tables

  The entry is found using the hash algorithm of the internal table. The runtime is independent of the number of table entries.

Using a Different Search Key

To use a key other than the table key as a search key, enter <key> as follows:

```
READ TABLE <itab> WITH KEY = <f> <result>.
```

or as follows

```
READ TABLE <itab> WITH KEY <k1> = <f1> ... <kn> = <fn> <result>.
```

In the first case, the whole line of the internal table is used as the search key. The contents of the entire table line are compared with the contents of field <f>. If <f> is not compatible with the line type of the table, the value is converted into the line type. The search key allows you to find entries in internal tables that do not have a structured line type, that is, where the line is a single field or an internal table type.

In the second case, the search key can consist of any of the table fields <k_1>...<k_n>. If you do not know the name of one of the components until runtime, you can specify it as the content of a field <n_i> using the form (<n_i>) = <f_i>. If <n_i> is empty when the statement is executed, the search field is ignored. If the data types of <f_i> are not compatible with the components in the internal table, the system converts them. You can restrict the search to partial fields [Page 196] by specifying offset and length.

The search is linear for **all table types**. The runtime is in linear relation to the number of table lines.

Specifying the Extra Processing Option

You can specify an option that specifies what the system does with the table entry that it finds.

Using a Work Area

You can write the table entry read from the table into a work area by specifying <result> as follows:

```
READ TABLE <itab> <key> INTO <wa> [COMPARING <f1> <f2> ...
                                           |ALL FIELDS]
                                  [TRANSPORTING <f1> <f2> ...
                                                |ALL FIELDS
                                                |NO FIELDS].
```

If you do not use the additions COMPARING or TRANSPORTING, the contents of the table line must be convertible into the data type of the work area <wa>. If you specify COMPARING or TRANSPORTING, the line type and work area must be compatible. You should always use a work area that is compatible with the line type of the relevant internal table.

If you use the COMPARING addition, the specified table fields <f_i> of the structured line type are compared with the corresponding fields of the work area **before** being transported. If you use the ALL FIELDS option, the system compares all components. If the system finds an entry with the specified key <key> and if the contents of the compared fields are the same, SY-SUBRC is set to 0. If the contents of the compared fields are not the same, it returns the value 2. If the system cannot find an entry, SY-SUBRC is set to 4. If the system finds an entry, it copies it into the target work area regardless of the result of the comparison.

If you use the TRANSPORTING addition, you can specify the table fields of the structured line type that you want to transport into the work area. If you specify ALL FIELDS without TRANSPORTING, the contents of all of the fields are transported. If you specify NO FIELDS, no fields are transported. In the latter case, the READ statement only fills the system fields SY-SUBRC and SY-TABIX. Specifying the work area <wa> with TRANSPORTING NO FIELDS is unnecessary, and should be omitted.

In both additions, you can specify a field <f_i> dynamically as the contents of a field <n_i> in the form (<n_i>). If <n_i> is empty when the statement is executed, it is ignored. You can restrict the search to partial fields [Page 196] by specifying offset and length.

Using a Field Symbol

You can assign the table entry read from the table to a field symbol by specifying <result> as follows:

```
READ TABLE <itab> <key> ASSIGNING <FS>.
```

After the READ statement, the field symbol points to the table line. If the line type is structured, you should specify the same type for the field symbol when you declare [Page 203] it. This allows you to address the components of the field symbol. If you cannot specify the type statically, you must use further field symbols and the technique of assigning components of structures [Page 213] to address the components of the structure.

For further information about assigning table lines to field symbols, refer to Access Using Field Symbols [Page 326].

Examples

```
DATA: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE HASHED TABLE OF LINE WITH UNIQUE KEY COL1.

DO 4 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  INSERT LINE INTO TABLE ITAB.
ENDDO.

LINE-COL1 = 2. LINE-COL2 = 3.

READ TABLE ITAB FROM LINE INTO LINE COMPARING COL2.

WRITE: 'SY-SUBRC =', SY-SUBRC.
SKIP.
WRITE: / LINE-COL1, LINE-COL2.
```

The output is:

```
SY-SUBRC =   2

           2          4
```

The program fills a hashed table with a list of square numbers. The work area LINE, which is compatible with the line type, is filled with the numbers 2 and 3. The READ statement reads the line of the table in which the key field COL1 has the same value as in the work area and copies it into the work area. SY-SUBRC is set to 2, because the contents of field COL2 were different.

```
DATA: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF LINE.
```

```
DATA ITAB LIKE SORTED TABLE OF LINE WITH UNIQUE KEY COL1.

DO 4 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  INSERT LINE INTO TABLE ITAB.
ENDDO.

CLEAR LINE.

READ TABLE ITAB WITH TABLE KEY COL1 = 3
                INTO LINE TRANSPORTING COL2.

WRITE:   'SY-SUBRC =', SY-SUBRC,
       / 'SY-TABIX =', SY-TABIX.
SKIP.
WRITE: / LINE-COL1, LINE-COL2.
```

The output is:

```
SY-SUBRC =     0
SY-TABIX =         3

          0         9
```

The program fills a sorted table with a list of square numbers. The READ statement reads the line of the table in which the key field COL1 has the same value as in the work area and copies it into the work area. Only the contents of COL2 are copied into the work area LINE. SY-SUBRC is zero, and SY-TABIX is 3, because ITAB is an index table.

```
DATA: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE SORTED TABLE OF LINE WITH UNIQUE KEY COL1.

DO 4 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  INSERT LINE INTO TABLE ITAB.
ENDDO.

READ TABLE ITAB WITH KEY COL2 = 16  TRANSPORTING NO FIELDS.

WRITE:   'SY-SUBRC =', SY-SUBRC,
       / 'SY-TABIX =', SY-TABIX.
```

The output is:

```
SY-SUBRC =     0
SY-TABIX =         4
```

The program fills a sorted table with a list of square numbers. The READ statement reads the line of the table in which the key field COL1 has the value 16. It does not use the table key. No fields are copied to a work area or assigned to a field symbol. Instead, only the system fields are set. SY-SUBRC is zero, since a line was found, and SY-TABIX is four.

```
DATA: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE HASHED TABLE OF LINE WITH UNIQUE KEY COL1.

FIELD-SYMBOLS <FS> LIKE LINE OF ITAB.

DO 4 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  INSERT LINE INTO TABLE ITAB.
ENDDO.

READ TABLE ITAB WITH TABLE KEY COL1 = 2 ASSIGNING <FS>.

<FS>-COL2 = 100.

LOOP AT ITAB INTO LINE.
  WRITE: / LINE-COL1, LINE-COL2.
ENDLOOP.
```

The output is:

```
1         1
2       100
3         9
4        16
```

The program fills a hashed table with a list of square numbers. The READ statement reads the line of the table in which the key field COL1 has the value 2 and assigns it to the field symbol <FS>. The program then assigns the value 100 to component COL2 of <FS>. This also changes the corresponding table field.

Changing Lines

To change a single line of any internal table, use the MODIFY statement. You can either use the table key to find and change a single line using its key, or find and change a set of lines that meet a certain condition. If the table has a non-unique key and there are duplicate entries, the first entry is changed.

Changing a Line Using the Table Key

To change a single line, use the following statement:

```
MODIFY TABLE <itab> FROM <wa> [TRANSPORTING <f1> <f2> ...].
```

The work area <wa>, which must be compatible with the line type of the internal table, plays a double role in this statement. Not only it is used to find the line that you want to change, but it also contains the new contents. The system searches the internal table for the line whose table key corresponds to the key fields in <wa>.

The system searches for the relevant lines as follows:

- Standard tables

  Linear search, where the runtime is in linear relation to the number of table entries. The first entry found is changed.

- Sorted tables

  Binary search, where the runtime is in logarithmic relation to the number of table entries. The first entry found is changed.

- Hashed tables

  The entry is found using the hash algorithm of the internal table. The runtime is independent of the number of table entries.

If a line is found, the contents of the non-key fields of the work area are copied into the corresponding fields of the line, and SY-SUBRC is set to 0. Otherwise, SY-SUBRC is set to 4. If the table has a non-unique key and the system finds duplicate entries, it changes the first entry.

You can specify the non-key fields that you want to assign to the table line in the TRANSPORTING addition. You can also specify a field <f_i> dynamically as the contents of a field <n_i> in the form (<n_i>). If <n_i> is empty when the statement is executed, it is ignored. You can restrict the search to partial fields [Page 196] by specifying offset and length.

For tables with a complex line structure, the usage of the transporting option results in better performance, if the system must not transport unnecessary table-like components.

Changing Several Lines Using a Condition

To change one or more lines using a condition, use the following statement:

```
MODIFY <itab> FROM <wa> TRANSPORTING <f1> <f2> ... WHERE <cond>.
```

This processes all of the lines that meet the logical condition [Page 225] <cond>. The logical condition can consist of more than one comparison. In each comparison, the first operand must be a component of the line structure. If the table lines are not structured, the first operand can also be the expression TABLE LINE. The comparison then applies to the entire line.

The work area <wa>, which must be compatible with the line type of the internal table, contains the new contents, which will be assigned to the relevant table line using the TRANSPORTING addition. Unlike the above MODIFY statement, the TRANSPORTING addition is not optional here. Furthermore, you can only modify the key fields of the internal table if it is a standard table. If at least one line is changed, the system sets SY-SUBRC to 0, otherwise to 4.

Examples

```
DATA: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE HASHED TABLE OF LINE WITH UNIQUE KEY COL1.

DO 4 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  INSERT LINE INTO TABLE ITAB.
ENDDO.

LINE-COL1 = 2. LINE-COL2 = 100.

MODIFY TABLE ITAB FROM LINE.

LOOP AT ITAB INTO LINE.
  WRITE: / LINE-COL1, LINE-COL2.
ENDLOOP.
```

The output is:

```
1         1
2       100
3         9
4        16
```

The program fills a hashed table with a list of square numbers. The MODIFY statement changes the line of the table in which the key field COL1 has the value 2.

```
DATA: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE HASHED TABLE OF LINE WITH UNIQUE KEY COL1.

DO 4 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  INSERT LINE INTO TABLE ITAB.
ENDDO.

LINE-COL2 = 100.
```

Changing Lines

```
MODIFY ITAB FROM LINE TRANSPORTING COL2
            WHERE ( COL2 > 1 ) AND ( COL1 < 4 ).

LOOP AT ITAB INTO LINE.
  WRITE: / LINE-COL1, LINE-COL2.
ENDLOOP.
```

The output is:

```
1         1
2       100
3       100
4        16
```

The program fills a hashed table with a list of square numbers. The MODIFY statement changes the lines of the table where the content of field COL2 is greater than 1 and the content of field COL1 is less than 4.

Deleting Lines

To delete a single line of any internal table, use the DELETE statement. You can either use the table key to find and delete a single line using its key, delete a set of lines that meet a condition, or find and delete neighboring duplicate entries. If the table has a non-unique key and there are duplicate entries, the first entry is deleted.

Deleting a Line Using the Table Key

To use the table key of table <itab> as a search key, use one of the following statements:

DELETE TABLE <itab> FROM <wa>.

or

DELETE TABLE <itab> WITH TABLE KEY $<k_1>$ = $<f_1>$ ... $<k_n>$ = $<f_n>$.

In the first case, <wa> must be a work area **compatible** with the line type of <itab>. The values of the key fields are taken from the corresponding components of the work area.

In the second case, you have to supply the values of each key field explicitly. If you do not know the name of one of the key fields until runtime, you can specify it as the content of a field $<n_i>$ using the form ($<n_i>$) = $<f_i>$. If the data types of $<f_i>$ are not compatible with the key fields, the system converts them.

The system searches for the relevant lines as follows:

- Standard tables

  Linear search, where the runtime is in linear relation to the number of table entries.

- Sorted tables

  Binary search, where the runtime is in logarithmic relation to the number of table entries.

- Hashed tables

  The entry is found using the hash algorithm of the internal table. The runtime is independent of the number of table entries.

If the system finds a line, it deletes it from the table and sets SY-SUBRC to zero. Otherwise, SY-SUBRC is set to 4. If the table has a non-unique key and the system finds duplicate entries, it deletes the first entry.

Deleting Several Lines Using a Condition

To delete more than one line using a condition, use the following statement:

DELETE <itab> WHERE <cond>.

This processes all of the lines that meet the logical condition [Page 225] <cond>. The logical condition can consist of more than one comparison. In each comparison, the first operand must be a component of the line structure. If the table lines are not structured, the first operand can also be the expression TABLE LINE. The comparison then applies to the entire line. If at least one line is deleted, the system sets SY-SUBRC to 0, otherwise to 4.

Deleting Adjacent Duplicate Entries

To delete adjacent duplicate entries use the following statement:

Deleting Lines

DELETE ADJACENT DUPLICATE ENTRIES FROM <itab>
[COMPARING <f_1> <f_2> ...
|ALL FIELDS].

The system deletes all adjacent duplicate entries from the internal table <itab>. Entries are duplicate if they fulfill one of the following compare criteria:

- Without the COMPARING addition, the contents of the key fields of the table must be identical in both lines.
- If you use the addition COMPARING <f_1> <f_2> ... the contents of the specified fields <f_1> <f_2> ... must be identical in both lines. You can also specify a field <f_i> dynamically as the contents of a field <n_i> in the form (<n_i>). If <n_i> is empty when the statement is executed, it is ignored. You can restrict the search to partial fields [Page 196] by specifying offset and length.
- If you use the addition COMPARING ALL FIELDS the contents of all fields of both lines must be identical.

You can use this statement to delete **all** duplicate entries from an internal table if the table is sorted by the specified compare criterion.

If at least one line is deleted, the system sets SY-SUBRC to 0, otherwise to 4.

Examples

```
DATA: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE HASHED TABLE OF LINE WITH UNIQUE KEY COL1.

DO 4 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  INSERT LINE INTO TABLE ITAB.
ENDDO.

LINE-COL1 = 1.

DELETE TABLE ITAB: FROM LINE,
                   WITH TABLE KEY COL1 = 3.

LOOP AT ITAB INTO LINE.
  WRITE: / LINE-COL1, LINE-COL2.
ENDLOOP.
```

The output is:

```
2         4
4        16
```

The program fills a hashed table with a list of square numbers. The DELETE statement delete the lines from the table where the key field COL1 has the contents 1 or 3.

```
DATA: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE HASHED TABLE OF LINE WITH UNIQUE KEY COL1.

DO 4 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  INSERT LINE INTO TABLE ITAB.
ENDDO.

DELETE ITAB WHERE ( COL2 > 1 ) AND ( COL1 < 4 ).

LOOP AT ITAB INTO LINE.
  WRITE: / LINE-COL1, LINE-COL2.
ENDLOOP.
```

The output is:

```
         1         1
         4        16
```

The program fills a hashed table with a list of square numbers. The DELETE statement deletes the lines of the table where the content of field COL2 is greater than 1 and the content of field COL1 is less than 4.

```
DATA OFF TYPE I.

DATA: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE C,
      END OF LINE.

DATA ITAB LIKE STANDARD TABLE OF LINE
          WITH NON-UNIQUE KEY COL2.

LINE-COL1 = 1. LINE-COL2 = 'A'. APPEND LINE TO ITAB.
LINE-COL1 = 1. LINE-COL2 = 'A'. APPEND LINE TO ITAB.
LINE-COL1 = 1. LINE-COL2 = 'B'. APPEND LINE TO ITAB.
LINE-COL1 = 2. LINE-COL2 = 'B'. APPEND LINE TO ITAB.
LINE-COL1 = 3. LINE-COL2 = 'B'. APPEND LINE TO ITAB.
LINE-COL1 = 4. LINE-COL2 = 'B'. APPEND LINE TO ITAB.
LINE-COL1 = 5. LINE-COL2 = 'A'. APPEND LINE TO ITAB.

OFF = 0. PERFORM LIST.

DELETE ADJACENT DUPLICATES FROM ITAB COMPARING ALL FIELDS.

OFF = 14. PERFORM LIST.

DELETE ADJACENT DUPLICATES FROM ITAB COMPARING COL1.

OFF = 28. PERFORM LIST.

DELETE ADJACENT DUPLICATES FROM ITAB.
```

Deleting Lines

```
OFF = 42. PERFORM LIST.

FORM LIST.
  SKIP TO LINE 3.
  LOOP AT ITAB INTO LINE.
     WRITE: AT /OFF  LINE-COL1, LINE-COL2.
  ENDLOOP.
ENDFORM.
```

The output is:

```
1 A          1 A          1 A          1 A
1 A          1 B          2 B          2 B
1 B          2 B          3 B          5 A
2 B          3 B          4 B
3 B          4 B          5 A
4 B          5 A
5 A
```

The example creates and fills a standard table. Here, the first DELETE statement deletes the second line from ITAB because the second line has the same contents as the first line. The second DELETE statement deletes the second line from the remaining table because the contents of the field COL1 is the same as in the first line. The third DELETE statement deletes the third and fourth line from the remaining table because the contents of the default key field COL2 are the same as on the second line. Although the contents of the default key are the same for the first and the fifth line, the fifth line is not deleted because it is not adjacent to the first line.

Processing Table Entries in Loops

You can use the LOOP statement to process special loops [Page 245] for any internal table.

```
LOOP AT <itab> <result> <condition>.
  <statement block>
ENDLOOP.
```

This reads the lines of the table one by one as specified in the <result> part of the LOOP statement. You can then process them in the statements within the LOOP... ENDLOOP control structure. You can either run the loop for all entries in the internal table, or restrict the number of lines read by specifying a <condition>. Control level processing is allowed within the loop.

The sequence in which the lines are processed depends on the table type:

- Standard tables and sorted tables

  The lines are processed according to the linear index. Within the processing block, the system field SY-TABIX contains the index of the current line.

- Hashed tables

  As long as the table has not been sorted, the lines are processed in the order in which you added them to the table. Within the processing block, the system field SY-TABIX is always 0.

You can nest LOOP blocks. When you leave the loop, SY-TABIX has the same value as when you entered it. After the ENDLOOP statement, SY-SUBRC is zero if at least one table entry was processed. Otherwise, it is 4.

The loop may not contain any operations on entire internal tables [Page 264] that change the table. However, you should remember that even saving a global internal table with the LOCAL statement [Page 453] in a procedure is a change operation on the entire table, since it exchanges the table contents. When you call procedures within loops, you should therefore check that it does not change the entire internal table. If you change the table, the loop can no longer work properly.

If you insert or delete a table entry within a loop pass, it is taken into account in subsequent loop passes as follows:

- If you insert a line after the current line, it will be processed in a subsequent loop pass.
- If you delete a line after the current line, it will not be processed in a subsequent loop pass.
- If you insert a line before or at the current line, the internal loop counter will be increased accordingly.
- If you delete a line before or at the current line, the internal loop counter will be decreased accordingly.

Specifying the Extra Processing Option

The processing option specifies how a table line is available in the statement block of the list.

Using a Work Area

To place the current loop line into a work area, specify <result> as follows:

Processing Table Entries in Loops

```
LOOP AT <itab> INTO <wa> <condition>.
```

The contents of the table lines must be convertible into the data type of the work area <wa>. In each loop pass, one line of the table is copied into the work area. The end of the loop does not affect the work area, that is, the contents of <wa> are the same after the ENDLOOP statement as they were in the final loop pass. If no table entries are processed in the loop, because the table is entry, or no line meets the condition <condition>, the work area is not changed.

Using a Field Symbol

To assign the contents of the current loop line to a field symbol, specify <result> as follows:

```
LOOP AT <itab> ASSIGNING <FS> <conditions>.
```

In each loop pass, the field symbol <FS> points to the table entry read in that pass. If the line type is structured, you should specify the same type for the field symbol when you declare [Page 203] it. This allows you to address the components of the field symbol. If you cannot specify the type statically, you must use further field symbols and the technique of assigning components of structures [Page 213] to address the components of the structure.

The end of the loop does not affect the field symbol, that is, after ENDLOOP it is still assigned to the same line as in the final loop pass. If no table entries are processed in the loop, because the table is entry, or no line meets the condition <condition>, the field symbol is not changed.

For further information about assigning table lines to field symbols, refer to Access Using Field Symbols [Page 326].

Suppressing the Assignment of Lines

If you do not need to transfer the contents of the current table line to a work area or assign them to a field symbol, you can use the following statement:

```
LOOP AT <itab> TRANSPORTING NO FIELDS <condition>.
```

This form of the LOOP statement is useful if you want to find the index of a particular internal table, or the number of lines in a table that meet a particular condition.

Specifying Conditions

To avoid reading all of the lines in an internal table, you can specify a condition for the line selection as follows:

```
LOOP AT <itab> <result> WHERE <cond>.
```

This processes all of the lines that meet the logical condition [Page 225] <cond>. The logical condition can consist of more than one comparison. In each comparison, the first operand must be a component of the line structure. If the table lines are not structured, the first operand can also be the expression TABLE LINE. The comparison then applies to the entire line.

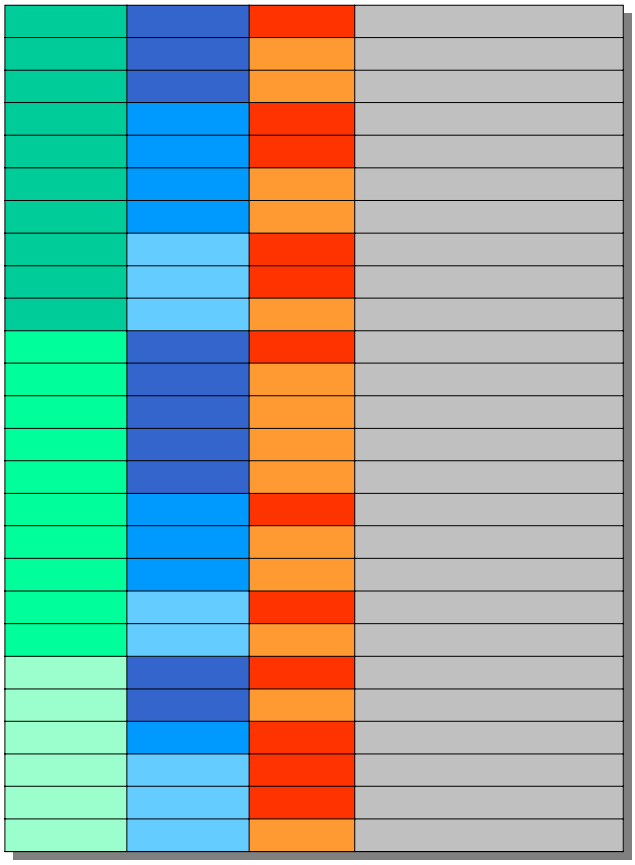
Control level processing

Control level processing is allowed within a LOOP over an internal table. This means that you can divide sequences of entries into groups based on the contents of certain fields.

Internal tables are divided into groups according to the sequence of the fields in the line structure. The first column defines the highest control level and so on. The control level hierarchy must be known when you create the internal table.

The control levels are formed by sorting the internal table in the sequence of its structure, that is, by the first field first, then by the second field, and so on. Tables in which the table key occurs at the start of the table are particularly suitable for control level processing.

The following diagram illustrates control level processing in a sorted table, where different field contents in the first three fields are indicated by different colors:



Control levels

Each change of color in a column indicates a control level change in the corresponding hierarchy level. Within the processing block of a loop, you can use the control level statement AT to react to a control level change. This enables you to restrict statements to a certain set of lines. You can thus use the SUM statement to calculate totals from subsets of all lines.

The AT statement introduces a statement block that you end with the ENDAT statement.

```
AT <level>.
  <statement block>
ENDAT.
```

You can react to the following control level changes:

<level>	Meaning
FIRST	First line of the internal table
LAST	Last line of the internal table

Processing Table Entries in Loops

NEW <f>	Beginning of a group of lines with the same contents in the field <f> and in the fields left of <f>
END Of <f>	End of a group of lines with the same contents in the field <f> and in the fields left of <f>

You can use control level statements to react to control breaks in internal tables instead of programming them yourself with logical expressions. Within the loop, you must order the AT-ENDAT statement blocks according to the hierarchy of the control levels. If the internal table has the columns <f_1>, <f_2>, ..., and if it is sorted by these columns, you must program the loop as follows:

```
LOOP AT <itab>.
  AT FIRST. ... ENDAT.
    AT NEW <f1>. ..... ENDAT.
      AT NEW <f2>. ..... ENDAT.
        .......
          <single line processing>
        .......
      AT END OF <f2>. ... ENDAT.
    AT END OF <f1>. ... ENDAT.
  AT LAST. .... ENDAT.
ENDLOOP.
```

The innermost hierarchy level <single line processing> processes the table lines that do not correspond to a control level change. You do not have to use all control level statements. But you must place the used ones in the above sequence. You should not use control level statements in loops where the line selection is restricted by WHERE or FROM and TO. Neither should the table be modified during the loop.

If a control level field <f_i> is not known until runtime, you can specify it dynamically as (<n_i>) where <n_i> contains the field of <f_i>. If <n_i> is empty at runtime, the criterion for changing the control level is ignored. You can restrict the search to partial fields [Page 196] by specifying offset and length.

If you are working with a work area <wa>, it **does not** contain the current line in the AT... ENDAT statement block. All character fields to the right of the current group key are filled with asterisks (*). All other fields to the right of the current group key contain their initial value.

Within an AT...ENDAT block, you can calculate the contents of the numeric fields of the corresponding control level using the SUM statement.

```
SUM.
```

You can only use this statement within a LOOP. If you use SUM in an AT - ENDAT block, the system calculates totals for the numeric fields of **all** lines in the current line group and writes them to the corresponding fields in the work area (see example in). If you use the SUM statement outside an AT - ENDAT block (single entry processing), the system calculates totals for the numeric fields of **all** lines of the internal table in **each** loop pass and writes them to the corresponding fields of the work area. It therefore only makes sense to use the SUM statement in AT...ENDAT blocks.

If the table contains a nested table, you cannot use the SUM statement. Neither can you use it if you are using a field symbol instead of a work area in the LOOP statement.

Examples

```
DATA: BEGIN OF LINE,
         COL1 TYPE C,
         COL2 TYPE I,
         COL3 TYPE I,
       END OF LINE.

DATA ITAB LIKE HASHED TABLE OF LINE
          WITH UNIQUE KEY COL1 COL2.

LINE-COL1 = 'A'.
DO 3 TIMES.
  LINE-COL2 = SY-INDEX.
  LINE-COL3 = SY-INDEX ** 2.
  INSERT LINE INTO TABLE ITAB.
ENDDO.

LINE-COL1 = 'B'.
DO 3 TIMES.
  LINE-COL2 = 2 * SY-INDEX.
  LINE-COL3 = ( 2 * SY-INDEX ) ** 2.
  INSERT LINE INTO TABLE ITAB.
ENDDO.

SORT ITAB.

LOOP AT ITAB INTO LINE.
  WRITE: / LINE-COL1, LINE-COL2, LINE-COL3.
  AT END OF COL1.
    SUM.
    ULINE.
    WRITE: / LINE-COL1, LINE-COL2, LINE-COL3.
    SKIP.
  ENDAT.
  AT LAST.
    SUM.
    ULINE.
    WRITE: / LINE-COL1, LINE-COL2, LINE-COL3.
  ENDAT.
ENDLOOP.
```

The output is:

```
A           1           1
A           2           4
A           3           9
_____________________________________
A           6          14

B           2           4
B           4          16
```

```
B          6         36
_____________________________________
B         12         56

_____________________________________
*         18         70
```

The program creates a hashed table ITAB, fills it with six lines, and sorts it. In the LOOP - ENDLOOP block, the work area LINE is output for each loop pass. The first field of the table key, COL1, is used for control level processing. The total for all numeric fields is always calculated when the contents of COL1 change and when the system is in the last loop pass.

```
DATA: BEGIN OF LINE,
        CARRID   TYPE SBOOK-CARRID,
        CONNID   TYPE SBOOK-CONNID,
        FLDATE   TYPE SBOOK-FLDATE,
        CUSTTYPE TYPE SBOOK-CUSTTYPE,
        CLASS    TYPE SBOOK-CLASS,
        BOOKID   TYPE SBOOK-BOOKID,
      END OF LINE.

DATA ITAB LIKE SORTED TABLE OF LINE WITH UNIQUE KEY TABLE
LINE.

SELECT CARRID CONNID FLDATE CUSTTYPE CLASS BOOKID
       FROM SBOOK INTO CORRESPONDING FIELDS OF TABLE ITAB.

LOOP AT ITAB INTO LINE.

  AT FIRST.
    WRITE / 'List of Bookings'.
    ULINE.
  ENDAT.

    AT NEW CARRID.
      WRITE: / 'Carrid:', LINE-CARRID.
    ENDAT.

      AT NEW CONNID.
        WRITE: / 'Connid:', LINE-CONNID.
      ENDAT.

        AT NEW FLDATE.
          WRITE: / 'Fldate:', LINE-FLDATE.
        ENDAT.

          AT NEW CUSTTYPE.
            WRITE: / 'Custtype:', LINE-CUSTTYPE.
          ENDAT.

               WRITE: / LINE-BOOKID, LINE-CLASS.

            AT END OF CLASS.
              ULINE.
            ENDAT.
```

```
ENDLOOP.
```

In this example, the sorted internal table ITAB is filled with data from the database table SBOOK using the Open SQL statement SELECT. The sequence of the columns in the internal table defines the control level hierarchy. Since the table key is the entire line, the sort sequence and the control level hierarchy are the same. The sequence of the AT-ENDAT blocks within the LOOP and ENDLOOP statements is important.

The output is as follows:

```
List of Bookings
Carrid: AA
Connid: 0017
Fldate: 1998/11/22
Custtype: B
00063509 C
00063517 C
...
_____________________________________________________
00063532 F
00063535 F
...
_____________________________________________________
Custtype: P
00063653 C
00063654 C
...
_____________________________________________________
00063668 F
00063670 F
...
_____________________________________________________
Fldate: 1998/29/11
Custtype: B
00064120 C
00064121 C
...
```

and so on.

Operations for Index Tables

The operations listed below are only permitted for index tables (sorted and standard tables). Some of them are restricted to standard tables. Since it is quicker to access a table by index than by key, you should always use specific index operations when you know that a particular internal table is an index table.

In particular, the quickest way to fill a table line by line is to append lines to a standard table, since a standard table cannot have a unique key and therefore appends the lines without having to check the existing lines in the table. If you can either accommodate duplicate entries in a table, or exclude them in a different way, it can be quicker to fill a standard table and then sort it or assign it to a sorted table if the data does not have to be inserted into the table in the correct sort sequence.

Furthermore, the performance of operations that change the internal linear index has been improved in Release 4.5A. Previously, index manipulation costs for inserting and deleting liens in standard tables and sorted tables increased in linear relation to the number of lines. From Release 4.5A, the index manipulation costs only increase logarithmically with the number of lines, since the table indexes are now maintained as a tree structure. This makes insertion and deletion operations efficient, even in very large standard and sorted tables.

Appending Lines [Page 307]

Inserting Lines [Page 311]

Inserting Lines Using the Index [Page 311]

Reading Lines Using the Index [Page 314]

Binary Search in Standard Tables [Page 315]

Finding Character Strings in Internal Tables [Page 316]

Changing Lines Using the Index [Page 318]

Deleting Lines Using the Index [Page 321]

Specifying the Index in Loops [Page 324]

Appending Table Lines

There are several ways of adding lines to index tables. The following statements have no equivalent that applies to all internal tables.

Appending a Single Line

To add a line to an index table, use the statement:

```
APPEND <line> TO <itab>.
```

<line> is either a work area that is convertible to the line type, or the expression INITIAL LINE. If you use <wa>, the system adds a new line to the internal table <itab> and fills it with the contents of the work area. INITIAL LINE appends a blank line containing the correct initial value for each field of the structure. After each APPEND statement, the system field SY-TABIX contains the index of the appended line.

Appending lines to standard tables and sorted tables with a non-unique key works regardless of whether lines with the same key already exist in the table. Duplicate entries may occur. A runtime error occurs if you attempt to add a duplicate entry to a sorted table with a unique key. Equally, a runtime error occurs if you violate the sort order of a sorted table by appending to it.

Appending Several Lines

You can also append internal tables to index tables using the following statement:

```
APPEND LINES OF <itab1> TO <itab2>.
```

This statement appends the whole of ITAB1 to ITAB2. ITAB1 can be any type of table, but its line type must be convertible into the line type of ITAB2.

When you append an index table to another index table, you can specify the lines to be appended as follows:

```
APPEND LINES OF <itab1> [FROM <n1>] [TO <n2>] TO <itab2>.
```

$<n_1>$ and $<n_2>$ specify the indexes of the first and last lines of ITAB1 that you want to append to ITAB2.

This method of appending lines of one table to another is about 3 to 4 times faster than appending them line by line in a loop. After the APPEND statement, the system field SY-TABIX contains the index of the last line appended. When you append several lines to a sorted table, you must respect the unique key (if defined), and not violate the sort order. Otherwise, a runtime error will occur.

Ranked lists

You can use the APPEND statement to create ranked lists in standard tables. To do this, create an empty table, and then use the statement:

```
APPEND <wa> TO <itab> SORTED BY <f>.
```

The new line is **not** added to the end of the internal table <itab>. Instead, the table is sorted by field <f> in **descending** order. The work area <wa> must be compatible with the line type of the internal table. You cannot use the SORTED BY addition with sorted tables.

When you use this technique, the internal table may only contain as many entries as you specified in the INITIAL SIZE parameter of the table declaration. This is an exception to the

general rule, where internal tables can be extended dynamically. If you add more lines than specified, the last line is discarded. This is useful for creating ranked lists of limited length (for example "Top Ten"). You can use the APPEND statement to generate ranked lists containing up to 100 entries. When dealing with larger lists, it is advisable to sort [Page 271] tables normally for performance reasons.

Examples

```
DATA: BEGIN OF WA,
        COL1 TYPE C,
        COL2 TYPE I,
      END OF WA.

DATA ITAB LIKE TABLE OF WA.

DO 3 TIMES.
  APPEND INITIAL LINE TO ITAB.
  WA-COL1 = SY-INDEX. WA-COL2 = SY-INDEX ** 2.
  APPEND WA TO ITAB.
ENDDO.

LOOP AT ITAB INTO WA.
  WRITE: / WA-COL1, WA-COL2.
ENDLOOP.
```

The output is:

```
            0
1           1
            0
2           4
            0
3           9
```

This example creates an internal table ITAB with two columns that is filled in the DO loop. Each time the processing passes through the loop, an initialized line is appended and then the table work area is filled with the loop index and the square root of the loop index and appended.

```
DATA: BEGIN OF LINE1,
        COL1(3) TYPE C,
        COL2(2) TYPE N,
        COL3    TYPE I,
      END OF LINE1,
      TAB1 LIKE TABLE OF LINE1.

DATA: BEGIN OF LINE2,
        FIELD1(1)  TYPE C,
        FIELD2     LIKE TAB1,
      END OF LINE2,
      TAB2 LIKE TABLE OF LINE2.

LINE1-COL1 = 'abc'. LINE1-COL2 = '12'. LINE1-COL3 = 3.
APPEND LINE1 TO TAB1.
```

```
LINE1-COL1 = 'def'. LINE1-COL2 = '34'. LINE1-COL3 = 5.
APPEND LINE1 TO TAB1.

LINE2-FIELD1 = 'A'. LINE2-FIELD2 = TAB1.
APPEND LINE2 TO TAB2.

REFRESH TAB1.

LINE1-COL1 = 'ghi'. LINE1-COL2 = '56'. LINE1-COL3 = 7.
APPEND LINE1 TO TAB1.

LINE1-COL1 = 'jkl'. LINE1-COL2 = '78'. LINE1-COL3 = 9.
APPEND LINE1 TO TAB1.

LINE2-FIELD1 = 'B'. LINE2-FIELD2 = TAB1.
APPEND LINE2 TO TAB2.

LOOP AT TAB2 INTO LINE2.
  WRITE: / LINE2-FIELD1.
  LOOP AT LINE2-FIELD2 INTO LINE1.
    WRITE: / LINE1-COL1, LINE1-COL2, LINE1-COL3.
  ENDLOOP.
ENDLOOP.
```

The output is:

```
A
abc 12          3
def 34          5
B
ghi 56          7
jkl 78          9
```

The example creates two internal tables TAB1 and TAB2. TAB2 has a deep structure because the second component of LINE2 has the data type of internal table TAB1. LINE1 is filled and appended to TAB1. Then, LINE2 is filled and appended to TAB2. After clearing TAB1 with the REFRESH statement, the same procedure is repeated.

```
DATA: BEGIN OF LINE,
        COL1 TYPE C,
        COL2 TYPE I,
      END OF LINE.

DATA: ITAB LIKE TABLE OF LINE,
      JTAB LIKE ITAB.

DO 3 TIMES.
  LINE-COL1 = SY-INDEX. LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
  LINE-COL1 = SY-INDEX. LINE-COL2 = SY-INDEX ** 3.
  APPEND LINE TO JTAB.
ENDDO.

APPEND LINES OF JTAB FROM 2 TO 3 TO ITAB.
```

Appending Table Lines

```
LOOP AT ITAB INTO LINE.
  WRITE: / LINE-COL1, LINE-COL2.
ENDLOOP.
```

The output is:

```
1          1
2          4
3          9
2          8
3         27
```

This example creates two internal tables of the same type, ITAB and JTAB. In the DO loop, ITAB is filled with a list of square numbers, and JTAB with a list of cube numbers. Then, the last two lines of JTAB are appended to ITAB.

```
DATA: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
        COL3 TYPE I,
      END OF LINE.

DATA ITAB LIKE TABLE OF LINE INITIAL SIZE 2.

LINE-COL1 = 1. LINE-COL2 = 2. LINE-COL3 = 3.
APPEND LINE TO ITAB SORTED BY COL2.

LINE-COL1 = 4. LINE-COL2 = 5. LINE-COL3 = 6.
APPEND LINE TO ITAB SORTED BY COL2.

LINE-COL1 = 7. LINE-COL2 = 8. LINE-COL3 = 9.
APPEND LINE TO ITAB SORTED BY COL2.

LOOP AT ITAB INTO LINE.
  WRITE: / LINE-COL2.
ENDLOOP.
```

The output is:

```
          8
          5
```

The program inserts three lines into the internal table ITAB using the APPEND statement and the SORTED BY addition. The line with the smallest value for the field COL2 is deleted from the table, since the number of lines that can be appended is fixed through the INITIAL SIZE 2 addition in the DATA statement.

Inserting Lines Using the Index

The INSERT statement allows you not only to insert lines in any type of internal table, but also allows you to change them using a line index. You can insert either a single line or a group of lines into index tables using the index.

Inserting a Single Line

To insert a line into an index table, use the statement:

```
INSERT <line> INTO <itab> [INDEX <idx>].
```

<line> is either a work area that is convertible to the line type, or the expression INITIAL LINE. If you use <wa>, the system adds a new line to the internal table <itab> and fills it with the contents of the work area. INITIAL LINE inserts a blank line containing the correct initial value for each field of the structure.

If you use the INDEX option, the new line is inserted before the line which has the index <idx>. After the insertion, the new entry has the index <idx> and the index of the following lines is incremented by 1. If the table contains <idx> -1 lines, the new line is added at the end of the table. If the table has less than <idx> - 1 lines, the new line cannot be inserted, and SY-SUBRC is set to 4. When the system successfully adds a line to the table, SY-SUBRC is set to 0.

Without the INDEX addition, you can only use the above statement within a LOOP. Then, the new line is inserted before the current line (<idx> is implicitly set to SY-TABIX).

Appending lines to standard tables and sorted tables with a non-unique key works regardless of whether lines with the same key already exist in the table. Duplicate entries may occur. A runtime error occurs if you attempt to add a duplicate entry to a sorted table with a unique key. Equally, a runtime error occurs if you violate the sort order of a sorted table by appending to it.

Inserting Several Lines

To add several lines to an internal table, use the statement:

```
INSERT LINES OF <itab1> INTO <itab2> [INDEX <idx>].
```

The system inserts the lines of table <itab1> one by one into <itab2> using the same rules as for single lines. ITAB1 can be any type of table. The line type of ITAB1 must be convertible into the line type of ITAB2.

When you append an index table to another index table, you can specify the lines to be appended as follows:

```
INSERT LINES OF <itab1> [FROM <n1>] [TO <n2>] INTO <itab2>
                        [INDEX <idx>].
```

$<n_1>$ and $<n_2>$ specify the indexes of the first and last lines of ITAB1 that you want to insert into ITAB2.

Depending on the size of the tables and where they are inserted, this method of inserting lines of one table into another can be up to 20 times faster than inserting them line by line in a loop.

Examples

```
DATA: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE TABLE OF LINE.

DO 2 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

LINE-COL1 = 11. LINE-COL2 = 22.
INSERT LINE INTO ITAB INDEX 2.

INSERT INITIAL LINE INTO ITAB INDEX 1.

LOOP AT ITAB INTO LINE.
  WRITE: / SY-TABIX, LINE-COL1, LINE-COL2.
ENDLOOP.
```

The output is:

```
1           0           0
2           1           1
3          11          22
4           2           4
```

The example creates an internal table ITAB and fills it with two lines. A new line containing values is inserted before the second line. Then, an initialized line is inserted before the first line.

```
DATA: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE TABLE OF LINE.

DO 2 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

LOOP AT ITAB INTO LINE.
  LINE-COL1 = 3 * SY-TABIX. LINE-COL2 = 5 * SY-TABIX.
  INSERT LINE INTO ITAB.
ENDLOOP.
```

```
LOOP AT ITAB INTO LINE.
  WRITE: / SY-TABIX, LINE-COL1, LINE-COL2.
ENDLOOP.
```

The output is:

```
          1          3           5
          2          1           1
          3          9          15
          4          2           4
```

The example creates an internal table ITAB and fills it with two lines. Using a LOOP construction, the program inserts a new line before each existing line.

```
DATA: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF LINE.

DATA: ITAB LIKE TABLE OF LINE,
      JTAB LIKE ITAB.

DO 3 TIMES.
  LINE-COL1 = SY-INDEX. LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
  LINE-COL1 = SY-INDEX. LINE-COL2 = SY-INDEX ** 3.
  APPEND LINE TO JTAB.
ENDDO.

INSERT LINES OF ITAB INTO JTAB INDEX 1.

LOOP AT JTAB INTO LINE.
  WRITE: / SY-TABIX, LINE-COL1, LINE-COL2.
ENDLOOP.
```

The output is: :

```
          1          1           1
          2          2           4
          3          3           9
          4          1           1
          5          2           8
          6          3          27
```

The example creates two internal tables of the same type. Each is filled with three lines. Then, the entire table ITAB is inserted before the first line of JTAB.

Reading Lines Using the Index

You can use the READ statement to read lines in tables using their index. To read a single line of an index table, use the statement:

```
READ TABLE <itab> INDEX <idx> <result>.
```

The system reads the line with the index <idx> from the table <itab>. This is quicker than searching using the key [Page 287]. The <result> part can specify a further processing option for the line that is retrieved.

If an entry with the specified index was found, the system field SY-SUBRC is set to 0 and SY-TABIX contains the index of that line. Otherwise, SY-SUBRC is set to a value other than 0.

If <idx> is less than or equal to 0, a runtime error occurs. If <idx> is greater than the number of lines in the table, SY-SUBRC is set to 4.

Example

```
DATA: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE SORTED TABLE OF LINE WITH UNIQUE KEY COL1.

FIELD-SYMBOLS <FS> LIKE LINE OF ITAB.

DO 20 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = 2 * SY-INDEX.
  APPEND LINE TO ITAB.
ENDDO.

READ TABLE ITAB ASSIGNING <FS> INDEX 7.

WRITE:   SY-SUBRC, SY-TABIX.
WRITE: / <FS>-COL1, <FS>-COL2.
```

The output is:

```
    0          7
          7          14
```

The example creates a sorted table ITAB and fills it with 20 lines. The line with index 7 is read and assigned to the field symbol <FS>.

Binary Search in Standard Tables

If you read entries [Page 287] from standard tables using a key other than the default key, you can use a binary search instead of the normal linear search. To do this, include the addition BINARY SEARCH in the corresponding READ statements.

```
READ TABLE <itab> WITH KEY = <f> <result> BINARY SEARCH.
```

and

```
READ TABLE <itab> WITH KEY <k1> = <f1> ... <kn> = <fn> <result>
                                                    BINARY SEARCH.
```

The standard table must be sorted in ascending order by the specified search key. The BINARY SEARCH addition means that you can access an entry in a standard table by its key as quickly as you would be able to in a sorted table.

Example

```
DATA: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE STANDARD TABLE OF LINE.

DO 4 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

SORT ITAB BY COL2.

READ TABLE ITAB WITH KEY COL2 = 16 INTO LINE BINARY SEARCH.

WRITE: 'SY-SUBRC =', SY-SUBRC.
```

The output is:

```
SY-SUBRC =     0
```

The program fills a standard table with a list of square numbers and sorts them into ascending order by field COL2. The READ statement uses a binary search to look for and find the line in the table where COL2 has the value 16.

Finding Character Strings in Internal Tables

To find a string in a line of an index table, use the following statement:

```
SEARCH <itab> FOR <str> <options>.
```

The statement searches the internal table <itab> for the character string <str>. If the search is successful, SY-SUBRC is set to 0, and SY-TABIX is set to the index of the table line in which the string was found. SY-FDPOS contains the offset position of the string in the table line. Otherwise, SY-SUBRC is set to 4.

The statement treats all table lines as type C fields, regardless of their actual line type. There is no conversion. The search string <str> can have the same form as for a normal string search [Page 170] in a field.

The different options (<options>) for the search in an internal table <itab> are:

- **ABBREVIATED**

  Field <c> is searched for a word containing the string in <str>. The characters can be separated by other characters. The first letter of the word and the string <str> must be the same.

- **STARTING AT <lin$_1$>**

  Searches table <itab> for <str>, starting at line <lin$_1$>.<lin$_1$> can be a variable.

- **ENDING AT <lin$_2$>**

  Searches table <itab> for <str> up to line <lin$_2$>.<lin$_2$> can be a variable.

- **AND MARK**

  If the search string is found, all the characters in the search string (and all the characters in between when using ABBREVIATED) are converted to upper case.

This statement only works with index tables. There is no corresponding statement for hashed tables.

Example

```
DATA: BEGIN OF LINE,
        INDEX   TYPE I,
        TEXT(8) TYPE C,
      END OF LINE.

DATA ITAB LIKE SORTED TABLE OF LINE WITH UNIQUE KEY INDEX.

DATA NUM(2) TYPE N.

DO 10 TIMES.
  LINE-INDEX = SY-INDEX.
  NUM = SY-INDEX.
  CONCATENATE 'string' NUM INTO LINE-TEXT.
  APPEND LINE TO ITAB.
ENDDO.
```

```
SEARCH ITAB FOR 'string05' AND MARK.

WRITE: / '''string05'' found at line', (1) SY-TABIX,
         'with offset', (1) SY-FDPOS.

SKIP.

READ TABLE ITAB INTO LINE INDEX SY-TABIX.
WRITE: / LINE-INDEX, LINE-TEXT.
```

The output is:

```
'string05' found at line 5 with offset 4

         5 STRING05
```

The offset of the string found in the table is determined by the width of the first table column, which has type I and length 4. The option AND MARK changes the table contents in the corresponding line.

Changing Table Lines Using the Index

You can use the MODIFY statement to change lines in tables using their index. There is also a special variant of the WRITE TO statement that you can use to modify standard tables.

Changing Single Lines with MODIFY

To change a line using its index, use the following statement:

```
MODIFY <itab> FROM <wa> [INDEX <idx>] [TRANSPORTING <f1> <f2> ... ].
```

The work area <wa> specified in the FROM addition replaces the existing line in <itab>. The work area must be convertible into the line type of the internal table.

If you use the INDEX option, the contents of the work area overwrites the contents of the line with index <idx>. If the operation is successful, SY-SUBRC is set to 0. If the internal table contains fewer lines than <idx>, no line is changed and SY-SUBRC is set to 4.

Without the INDEX addition, you can only use the above statement within a LOOP. In this case, you change the current loop line <idx> is implicitly set to SY-TABIX.

When you change lines in sorted tables, remember that you must not change the contents of key fields, and that a runtime error occurs if you try to replace the contents of a key field with another value. However, you can assign the same value.

The TRANSPORTING addition allows you to specify the fields that you want to change explicitly in a list. See also Changing Table Entries [Page 292]. If you change a sorted table, you may only specify non-key fields.

Changing Lines Using WRITE TO

You can change lines of standard tables using the following statement:

```
WRITE <f> TO <itab> INDEX <idx>.
```

This variant of the WRITE TO [Page 148] statement converts the contents of field <f> to type C and then transfers the resulting character string into the line with index <idx>. If the operation is successful, SY-SUBRC is set to 0. If the internal table contains fewer lines than <idx>, no line is changed and SY-SUBRC is set to 4.

The data type of <f> must be convertible into a character field; if it is not, a syntax or runtime error occurs. The line is always interpreted as a character string, regardless of its actual line type. You can process components [Page 196] in the same way as in the normal WRITE TO statement. You should only use this statement for structured line types if you want to change a single character whose exact position you already know. Another possibility is to use internal tables whose structure is made up of a single character field. Tables of this kind are often used in dynamic programming [Page 513].

Examples

```
DATA: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF LINE.
```

```
DATA ITAB LIKE TABLE OF LINE.

DO 3 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

LOOP AT ITAB INTO LINE.
  IF SY-TABIX = 2.
     LINE-COL1 = SY-TABIX * 10.
     LINE-COL2 = ( SY-TABIX * 10 ) ** 2.
     MODIFY ITAB FROM LINE.
  ENDIF.
ENDLOOP.

LOOP AT ITAB INTO LINE.
  WRITE: / SY-TABIX, LINE-COL1, LINE-COL2.
ENDLOOP.
```

This produces the following output:

```
1          1           1
2         20         400
3          3           9
```

Here, a sorted table ITAB is created and filled with three lines. The second line is replaced by the contents of the work area LINE.

```
DATA NAME(4) VALUE 'COL2'.

DATA: BEGIN OF LINE,
         COL1 TYPE I,
         COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE SORTED TABLE OF LINE WITH UNIQUE KEY COL1.

DO 4 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

LINE-COL2 = 222.
MODIFY ITAB FROM LINE INDEX 2 TRANSPORTING (NAME).

LINE-COL1 = 3.
LINE-COL2 = 333.
MODIFY ITAB FROM LINE INDEX 3.

LOOP AT ITAB INTO LINE.
  WRITE: / SY-TABIX, LINE-COL1, LINE-COL2.
ENDLOOP.
```

The output is:

```
1          1            1
2          2          222
3          3          333
4          4           16
```

The example fills a sorted table with four lines. In the second and third lines, the component COL2 is modified. If the third line were to be changed so that the value of LINE-COL1 was no longer 3, a runtime error would occur, since the key fields of sorted tables may not be changed.

```
DATA TEXT(72).

DATA CODE LIKE TABLE OF TEXT.

TEXT = 'This is the first line.'.

APPEND TEXT TO CODE.

TEXT = 'This is the second line. It is ugly.'.

APPEND TEXT TO CODE.

TEXT = 'This is the third and final line.'.

APPEND TEXT TO CODE.

WRITE 'nice.' TO CODE+31 INDEX 2.

LOOP AT CODE INTO TEXT.

   WRITE / TEXT.

ENDLOOP.
```

This produces the following output:

```
This is the first line.
This is the second line. It is nice.
This is the third and final line.
```

Here, an internal table CODE is defined with an elementary type C field which is 72 characters long. After filling the table with three lines, the second line is changed by using the WRITE TO statement. The word "ugly" is replaced by the word "nice".

Deleting Lines Using the Index

You can use the DELETE statement to delete one or more lines from tables using their index.

Deleting a Single Line

To delete a line using its index, use the following statement:

```
DELETE <itab> [INDEX <idx>].
```

If you use the INDEX addition, the system deletes the line with the index <idx> from table <itab>, reduces the index of the subsequent lines by 1, and sets SY-SUBRC to zero. Otherwise, if no line with index <idx> exists, SY-SUBRC is set to 4.

Without the INDEX addition, you can only use the above statement within a LOOP. In this case, you delete the current loop line (<idx> is implicitly set to SY-TABIX).

Deleting Several Lines

To delete more than one line using the index, use the following statement:

```
DELETE <itab> [FROM <n1>] [TO <n2>] [WHERE <condition>].
```

Here, you must specify at least one of the additions. The WHERE addition has the same effect as when you delete entries [Page 295] from any table. As well as the WHERE clause, you can specify the lines that you want to delete by their index using FROM and TO. The system deletes all of the lines of <itab> whose index lies between $<n_1>$ and $<n_2>$. If you do not specify a FROM addition, the system deletes lines from the first line onwards. If you do not specify a TO addition, the system deletes lines up to the last line.

If at least one line is deleted, the system sets SY-SUBRC to 0, otherwise to 4.

Examples

```
DATA: BEGIN OF LINE,
         COL1 TYPE I,
         COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE SORTED TABLE OF LINE WITH UNIQUE KEY COL1.

DO 5 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

DELETE ITAB INDEX: 2, 3, 4.

WRITE: 'SY-SUBRC =', SY-SUBRC.

SKIP.

LOOP AT ITAB INTO LINE.
  WRITE: / SY-TABIX, LINE-COL1, LINE-COL2.
ENDLOOP.
```

The output is:

```
SY-SUBRC      4

          1           1           1
          2           3           9
          3           5          25
```

The example fills a sorted table ITAB with five lines. Then it deletes the three lines with the indexes 2, 3, and 4. After deleting the line with index 2, the index of the following lines is decremented by one. Therefore, the next deletion removes the line with an index which was initially 4. The third delete operation fails, since the table now only has three lines.

```
DATA: BEGIN OF LINE,
         COL1 TYPE I,
         COL2 TYPE I,
       END OF LINE.

DATA ITAB LIKE TABLE OF LINE.

DO 30 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

LOOP AT ITAB INTO LINE.
  IF LINE-COL1 < 28.
    DELETE ITAB.
  ENDIF.
ENDLOOP.

LOOP AT ITAB INTO LINE.
  WRITE: / SY-TABIX, LINE-COL1, LINE-COL2.
ENDLOOP.
```

The output is:

```
          1         28         784

          2         29         841

          3         30         900
```

The example fills a sorted table ITAB with 30 lines. Using a LOOP construction, the program deletes all of the lines in the table with a value of less than 28 in field COL1.

```
DATA: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE TABLE OF LINE.

DO 40 TIMES.
  LINE-COL1 = SY-INDEX.
```

```
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

DELETE ITAB FROM 3 TO 38 WHERE COL2 > 20.

LOOP AT ITAB INTO LINE.
  WRITE: / LINE-COL1, LINE-COL2.
ENDLOOP.
```

The output is:

```
          1           1
          2           4
          3           9
          4          16
         39       1.521
         40       1.600
```

The program deletes all entries from the standard table ITAB with an index between 3 and 39 where the value in COL2 is greater than 20.

Specifying the Index in Loops

When you process an internal table in a loop, you can specify the index of an index table to restrict the number of entries that are read:

```
LOOP AT <itab> <result> [FROM <n1>] [TO <n2>] <condition>.
  <statement block>
ENDLOOP.
```

The loop is processed as described in Processing Table Lines in Loops [Page 299]. Within the processing block, the system field SY-TABIX contains the index of the current line.

You can use the additions FROM and TO to specify the indexes <n_1> and <n_2> of the first and last entries that you want to read. The FROM and TO options restrict the number of lines which the system has to read. The WHERE addition in the condition only prevents the <result> from being processed. However, all of the table lines are read. To improve performance, you should use the FROM and TO options as much as possible. It can be also beneficial under certain conditions to leave the loop with the CONTINUE or EXIT statement.

Example

```
DATA: BEGIN OF LINE,
         COL1 TYPE I,
         COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE SORTED TABLE OF LINE WITH UNIQUE KEY TABLE
LINE.

DO 30 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

LOOP AT ITAB INTO LINE FROM 10 TO 25 WHERE COL2 > 400.
  WRITE: / SY-TABIX, LINE-COL2.
ENDLOOP.
```

The output is:

```
        21       441
        22       484
        23       529
        24       576
        25       625
```

The example fills a sorted table ITAB with 30 lines. In the loop, only the lines 10 to 25 are read. There is also a condition that the contents of COL2 must be more than 400.

Access Using Field Symbols

When you read table entries using READ or in a LOOP, you can assign them to a field symbol using the addition

```
... ASSIGNING <FS>
```

The field symbol <FS> points directly to the assigned line in memory. Unlike work areas, in which the contents of the line are only available indirectly, field symbols allow you to read and change table entries directly.

Remember when you access internal tables using field symbols that you must not change the contents of the key fields of sorted or hashed tables. If you try to assign a new value to a key field using a field symbol, a runtime error occurs. Note that you cannot use the SUM statement with field symbols, since the statement is always applied to work areas.

Advantages of Field Symbols

When you read from an internal table, there are no overheads for copying the table line to the work area. When you change an internal table with the MODIFY statement, you must first fill a work area with values, and then assign them to the internal table. If you work with field symbols instead, you do not have this overhead. This can improve performance if you have large or complex internal tables. It also makes it easier to process nested internal tables.

Overheads of READ

Note that internal overheads arise when you access internal tables using field symbols. After a READ statement with a field symbol, the system has to register the assignment. When you delete a table line to which a field symbol is pointing, the system also has to unassign the field symbol to prevent it from pointing to an undefined area.

When you read individual table lines, it is worth using field symbols with the READ statement for tables with a line width of 1000 bytes or more. If you also change the line using the MODIFY statement, using field symbols is worthwhile from a line width of 100 bytes onwards.

Overheads of LOOP

To minimize the overheads incurred by using field symbols in loop processing, the system does not register the assignment of each current line to the field symbol. Instead, it registers a general assignment between a line of the table and the field symbol. When the loop is finished, the line processed in the last loop pass is assigned to the field symbol.

Consequently, it is worth using field symbols in a LOOP when the internal table has as few as 10 lines. However, it is not possible to reassign the field symbol to another field or unassign it altogether within the loop. If you include the statements ASSIGN, UNASSIGN, or the ASSIGNING addition for the same field symbol within the loop block, a runtime error occurs.

Example

```
DATA: BEGIN OF LINE,
         COL1 TYPE I,
         COL2 TYPE I,
      END OF LINE.
```

```
DATA ITAB LIKE SORTED TABLE OF LINE WITH UNIQUE KEY COL1.

FIELD-SYMBOLS <FS> LIKE LINE OF ITAB.

DO 4 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

READ TABLE ITAB WITH TABLE KEY COL1 = 2 ASSIGNING <FS>.
<FS>-COL2 = 100.

READ TABLE ITAB WITH TABLE KEY COL1 = 3 ASSIGNING <FS>.
DELETE ITAB INDEX 3.

IF <FS> IS ASSIGNED.
  WRITE '<FS> is assigned!'.
ENDIF.

LOOP AT ITAB ASSIGNING <FS>.
  WRITE: / <FS>-COL1, <FS>-COL2.
ENDLOOP.
```

The output is:

```
1          1
2        100
4         16
```

The example fills a sorted table ITAB with 4 lines. The second line is assigned to the field symbol <FS> (which has the same type), and modified using it. The third line is assigned to <FS> and then deleted. Consequently, the logical expression in the IF statement is untrue. <FS> is used to display the table lines in the LOOP. Afterwards, it points to the third line of the table.

Using Header Lines as Work Areas

When you create an internal table object [Page 259] you can also declare a header line with the same name. You can use the header line as a work area when you process the internal table. The ABAP statements that you use with internal tables have short forms that you can use if your internal table has a header line. These statements automatically assume the header line as an implicit work area. The following table shows the statements that you must use for internal tables without a header line, and the equivalent statements that you can use for internal tables with a header line:

Operations without header line	Operations with header line
Operations for all Table Types	
`INSERT <wa> INTO TABLE <itab>.`	`INSERT TABLE ITAB.`
`COLLECT <wa> INTO <itab>.`	`COLLECT <itab>.`
`READ TABLE <itab> ... INTO <wa>.`	`READ TABLE <itab> ...`
`MODIFY TABLE <itab> FROM <wa> ...`	`MODIFY TABLE <itab> ...`
`MODIFY <itab> FROM <wa> ...WHERE ...`	`MODIFY <itab> ... WHERE ...`
`DELETE TABLE <itab> FROM <wa>.`	`DELETE TABLE <itab>.`
`LOOP AT ITAB INTO <wa> ...`	`LOOP AT ITAB ...`
Operations for Index Tables	
`APPEND <wa> TO <itab>.`	`APPEND <itab>.`
`INSERT <wa> INTO <itab> ...`	`INSERT <itab> ...`
`MODIFY <itab> FROM <wa> ...`	`MODIFY <itab> ...`

Using the header line as a work area means that you can use shorter statements; however, they are not necessarily easier to understand, since you cannot immediately recognize the origin and target of the assignment. Furthermore, the fact that the table and its header line have the same name can cause confusion in operations with entire internal tables [Page 264]. To avoid confusion, you should use internal tables with differently-named work areas.

Example

The following example shows two programs with the same function. One uses a header line, the other does not.

With header line:

```
TYPES: BEGIN OF LINE,
         COL1 TYPE I,
         COL2 TYPE I,
       END OF LINE.

DATA ITAB TYPE HASHED TABLE OF LINE WITH UNIQUE KEY COL1
               WITH HEADER LINE.
```

```
DO 4 TIMES.
  ITAB-COL1 = SY-INDEX.
  ITAB-COL2 = SY-INDEX ** 2.
  INSERT TABLE ITAB.
ENDDO.

ITAB-COL1 = 2.
READ TABLE ITAB FROM ITAB.

ITAB-COL2 = 100.
MODIFY TABLE ITAB.

ITAB-COL1 = 4.
DELETE TABLE ITAB.

LOOP AT ITAB.
  WRITE: / ITAB-COL1, ITAB-COL2.
ENDLOOP.
```

Without header line:

```
TYPES: BEGIN OF LINE,
         COL1 TYPE I,
         COL2 TYPE I,
       END OF LINE.

DATA: ITAB TYPE HASHED TABLE OF LINE WITH UNIQUE KEY COL1,
      WA LIKE LINE OF ITAB.

DO 4 TIMES.
  WA-COL1 = SY-INDEX.
  WA-COL2 = SY-INDEX ** 2.
  INSERT WA INTO TABLE ITAB.
ENDDO.

WA-COL1 = 2.
READ TABLE ITAB FROM WA INTO WA.

WA-COL2 = 100.
MODIFY TABLE ITAB FROM WA.

WA-COL1 = 4.
DELETE TABLE ITAB FROM WA.

LOOP AT ITAB INTO WA.
  WRITE: / WA-COL1, WA-COL2.
ENDLOOP.
```

The list, in both cases, appears as follows:

```
         1         1

         2       100

         3         9
```

The statements in the program that does not use a header line are easier to understand. As a further measure, you could have a further work area just to specify the key of the internal table, but to which no other values from the table are assigned.

Extracts

Since internal tables have fixed line structures, they are not suited to handle data sets with varying structures. Instead, you can use extract datasets for this purpose.

An extract is a sequential dataset in the memory area of the program. You can only address the entries in the dataset within a special loop. The index or key access permitted with internal tables is **not** allowed. You may only create one extract in any ABAP program. The size of an extract dataset is, in principle, unlimited. Extracts larger than 500KB are stored in operating system files. The practical size of an extract is up to 2GB, as long as there is enough space in the filesystem.

An extract dataset consists of a sequence of records of a pre-defined structure. However, the structure need not be identical for all records. In one extract dataset, you can store records of different length and structure one after the other. You need not create an individual dataset for each different structure you want to store. This fact reduces the maintenance effort considerably.

In contrast to internal tables, the system partly compresses extract datasets when storing them. This reduces the storage space required. In addition, you need not specify the structure of an extract dataset at the beginning of the program, but you can determine it dynamically during the flow of the program.

You can use control level processing with extracts just as you can with internal tables. The internal administration for extract datasets is optimized so that it is quicker to use an extract for control level processing than an internal table.

Procedure for creating an extract:

1. Define the record types that you want to use in your extract by declaring them as field groups. The structure is defined by including fields in each field group.

   Defining an Extract [Page 332]

2. Fill the extract line by line by extracting the required data.

   Filling an Extract with Data [Page 334]

3. Once you have filled the extract, you can sort it and process it in a loop. At this stage, you can no longer change the contents of the extract.

   Processing Extracts [Page 336]

Defining an Extract

To define an extract, you must first declare the individual records and then define their structure.

Declaring Extract Records as Field Groups

An extract dataset consists of a sequence of records. These records may have different structures. All records with the same structure form a record type. You must define each record type of an extract dataset as a field group, using the FIELD-GROUPS statement.

FIELD-GROUPS <fg>.

This statement defines a field group <fg>. A field group combines several fields under one name. For clarity, you should declare your field groups at the end of the declaration part of your program.

A field group does not reserve storage space for the fields, but contains pointers to existing fields. When filling the extract dataset with records, these pointers determine the contents of the stored records.

You can also define a special field group called HEADER:

FIELD-GROUPS HEADER.

This group is automatically placed before any other field groups when you fill the extract. This means that a record of a field group <fg> always contains the fields of the field group HEADER. When sorting the extract dataset, the system uses these fields as the default sort key.

Defining the Structure of a Field Group

To define the structure of a record, use the following statement to add the required fields to a field group:

INSERT $<f_1>$... $<f_n>$ INTO <fg>.

This statement defines the fields of field group <fg>. Before you can assign fields to a field group, you must define the field group <fg> using the FIELD-GROUPS statement. The fields in the field group must be global data objects in the ABAP program. You cannot assign a local data object defined in a procedure [Page 449] to a field group.

The INSERT statement, just as the FIELD-GROUPS statement, neither reserves storage space nor transfers values. You use the INSERT statement to create pointers to the fields $<f_i>$ in the field group <fg>, thus defining the structures of the extract records.

When you run the program, you can assign fields to a field group up to the point when you use this field group for the first time to fill an extract record. From this point on, the structure of the record is fixed and may no longer be changed. In short, as long as you have not used a field group yet, you can still extend it dynamically.

The special field group HEADER is part of every extract record. Consequently, you may not change HEADER once you have filled the first extract record.

A field may occur in several field groups; however, this means unnecessary data redundancy within the extract dataset. You do not need to define the structure of a field group explicitly with INSERT. If the field group HEADER is defined, an undefined field group consists implicitly of the fields in HEADER, otherwise, it is empty.

```
NODES: SPFLI, SFLIGHT.

FIELD-GROUPS: HEADER, FLIGHT_INFO, FLIGHT_DATE.

INSERT: SPFLI-CARRID SPFLI-CONNID SFLIGHT-FLDATE
    INTO HEADER,
    SPFLI-CITYFROM SPFLI-CITYTO
    INTO FLIGHT_INFO.
```

The program is linked to the logical database [Page 1163] F1S. The NODES statement declares the corresponding interface work areas [Page 130].

There are three field groups. The INSERT statement assigns fields to two of the field groups.

Filling an Extract with Data

Once you have declared the possible record types as field groups and defined their structure, you can fill the extract dataset using the following statements:

EXTRACT <fg>.

When the first EXTRACT statement occurs in a program, the system creates the extract dataset and adds the first extract record to it. In each subsequent EXTRACT statement, the new extract record is added to the dataset.

Each extract record contains exactly those fields that are contained in the field group <fg>, plus the fields of the field group HEADER (if one exists). The fields from HEADER occur as a sort key at the beginning of the record. If you do not explicitly specify a field group <fg>, the

EXTRACT

statement is a shortened form of the statement

EXTRACT HEADER.

When you extract the data, the record is filled with the current values of the corresponding fields.

As soon as the system has processed the first EXTRACT statement for a field group <fg>, the structure of the corresponding extract record in the extract dataset is fixed. You can no longer insert new fields into the field groups <fg> and HEADER. If you try to modify one of the field groups afterwards and use it in another EXTRACT statement, a runtime error occurs.

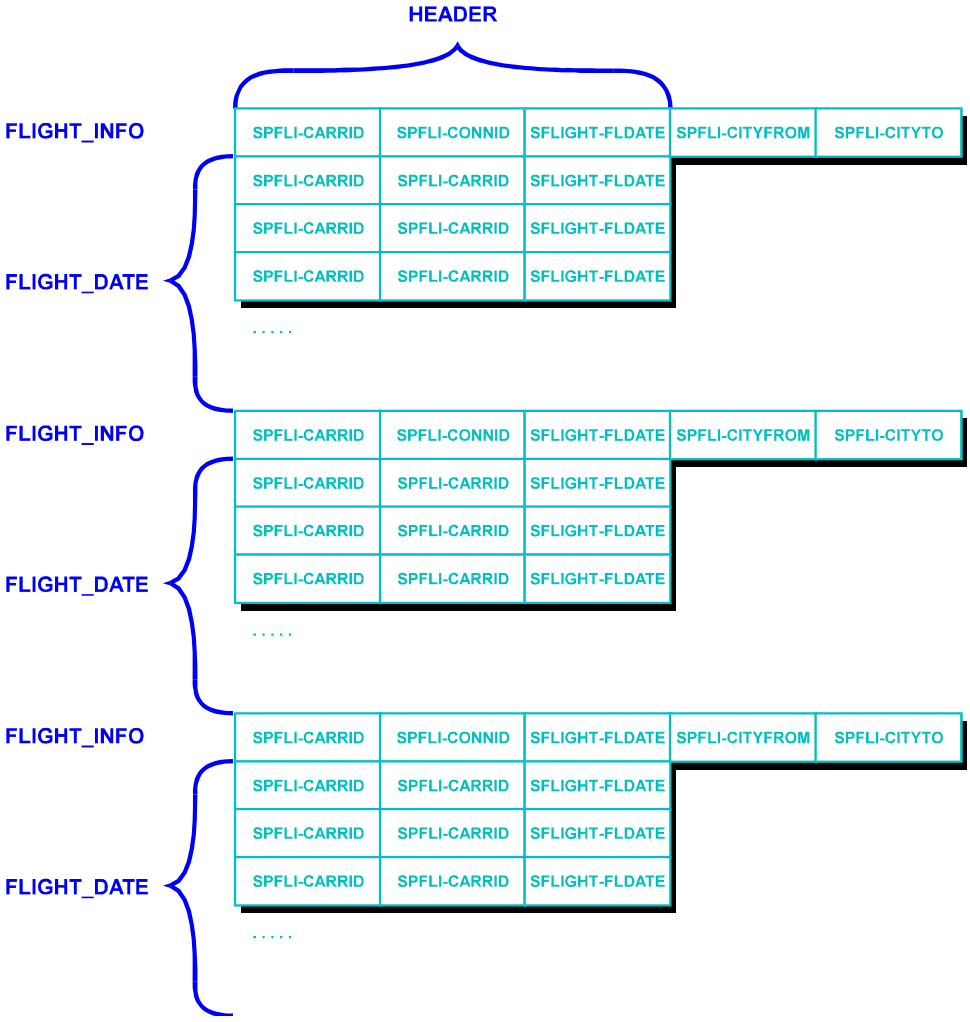
By processing EXTRACT statements several times using different field groups, you fill the extract dataset with records of different length and structure. Since you can modify field groups dynamically up to their first usage in an EXTRACT statement, extract datasets provide the advantage that you need not determine the structure at the beginning of the program.

Assume the following program is linked to the logical database [Page 1163] F1S.

```
REPORT DEMO.

NODES: SPFLI, SFLIGHT.

FIELD-GROUPS: HEADER, FLIGHT_INFO, FLIGHT_DATE.

INSERT: SPFLI-CARRID SPFLI-CONNID SFLIGHT-FLDATE
    INTO HEADER,
    SPFLI-CITYFROM SPFLI-CITYTO
    INTO FLIGHT_INFO.

START-OF-SELECTION.

GET SPFLI.
  EXTRACT FLIGHT_INFO.

GET SFLIGHT.
  EXTRACT FLIGHT_DATE.
```

There are three field groups. The INSERT statement assigns fields to two of the field groups. During the GET [Page 958] events, the system fills the extract dataset with two different record types. The records of the field group FLIGHT_INFO consist of five fields: SPFLI-CARRID, SPFLI-CONNID, SFLIGHT-FLDATE, SPFLI-CITYFROM,

and SPFLI-CITYTO. The first three fields belong to the prefixed field group HEADER. The records of the field group FLIGHT_DATE consist only of the three fields of field group HEADER. The following figure shows the structure of the extract dataset:



Processing Extracts

Once an extract dataset contains all of the required data, you can process it. When you have started processing it, you can no longer extract data into it.

Reading an Extract

Like internal tables, you can read the data in an extract dataset using a loop [Page 245].

```
LOOP.
...
  [AT FIRST | AT <fg_i> [WITH <fg_j>] | AT LAST.
   ...
   ENDAT.]
...
ENDLOOP.
```

When the LOOP statement occurs, the system stops creating the extract dataset, and starts a loop through the entries in the dataset. One record from the extract dataset is read in each loop pass. The values of the extracted fields are placed in the corresponding output fields within the loop. You can use several loops one after the other, but they cannot be nested. It is also no longer possible to use further EXTRACT statements within or after the loop. In both cases, a runtime error occurs.

In contrast to internal tables, extract datasets do not require a special work area or field symbol as an interface. Instead, you can process each record of the dataset within the loop using its original field names.

Loop control

If you want to execute some statements for certain records of the dataset only, use the control statements AT and ENDAT.

The system processes the statement blocks between the control statements for the different options of AT as follows:

- AT FIRST

  The system executes the statement block once for the first record of the dataset.

- AT <fg$_i$> [WITH <fgj>]

  The system processes the statement block, if the record type of the currently read extract record was defined using the field group <fg$_i$>. When using the WITH <fg$_j$> option, in the extract dataset, the currently read record of field group <fg$_i$> must be immediately followed by a record of field group <fg$_j$>.

- AT LAST

  The system executes the statement block once for the last record of the dataset.

You can also use the AT and ENDAT statements for control level processing [Page 343].

Assume the following program is linked to the logical database [Page 1163] F1S.

```
REPORT DEMO.

NODES: SPFLI, SFLIGHT.

FIELD-GROUPS: HEADER, FLIGHT_INFO, FLIGHT_DATE.
```

Reading an Extract

```
INSERT: SPFLI-CARRID SPFLI-CONNID SFLIGHT-FLDATE
    INTO HEADER,
    SPFLI-CITYFROM SPFLI-CITYTO
    INTO FLIGHT_INFO.

START-OF-SELECTION.

GET SPFLI.
  EXTRACT FLIGHT_INFO.

GET SFLIGHT.
  EXTRACT FLIGHT_DATE.

END-OF-SELECTION.

  LOOP.
    AT FIRST.
      WRITE / 'Start of LOOP'.
      ULINE.
    ENDAT.
    AT FLIGHT_INFO WITH FLIGHT_DATE.
      WRITE: / 'Info:',
               SPFLI-CARRID, SPFLI-CONNID, SFLIGHT-FLDATE,
               SPFLI-CITYFROM, SPFLI-CITYTO.
    ENDAT.
    AT FLIGHT_DATE.
      WRITE: / 'Date:',
                SPFLI-CARRID, SPFLI-CONNID, SFLIGHT-FLDATE.
    ENDAT.
    AT LAST.
   ULINE.
      WRITE / 'End of LOOP'.
    ENDAT.
  ENDLOOP.
```

The extract dataset is created and filled in the same way as shown in the example for Filling an Extract with Data [Page 334]. The data retrieval ends before the END-OF-SELECTION [Page 963] event, in which the dataset is read once using a loop.

The control statements AT FIRST and AT LAST instruct the system to write one line and one underscore line in the list, once at the beginning of the loop and once at the end.

The control statement AT <fg_i> tells the system to output the fields corresponding to each of the two record types. The WITH FLIGHT_DATE option means that the system only displays the records of field group FLIGHT_INFO if at least one record of field group FLIGHT_DATE follows; that is, if the logical database passed at least one date for a flight.

The beginning of the output list looks like this:

```
Start of LOOP

Info: AA   0017 ####/##/## ATLANTA              SAN FRANCISCO
Date: AA   0017 1996/01/09
Date: AA   0017 1996/01/10
Date: AA   0017 1996/01/11
Date: AA   0017 1996/01/12
Date: AA   0017 1996/01/13
Date: AA   0017 1996/01/14
Date: AA   0017 1996/01/15
Date: AA   0017 1996/01/16
Info: AA   0026 ####/##/## FRANKFURT            NEW YORK
Date: AA   0026 1996/01/09
Date: AA   0026 1996/01/10
Date: AA   0026 1996/01/11
Date: AA   0026 1996/01/12
Date: AA   0026 1996/01/13
Date: AA   0026 1996/01/14
Date: AA   0026 1996/01/15
Date: AA   0026 1996/01/16
Info: AA   0064 ####/##/## SAN FRANCISCO        NEW YORK
Date: AA   0064 1996/01/09
Date: AA   0064 1996/01/10
Date: AA   0064 1996/01/11
Date: AA   0064 1996/01/12
Date: AA   0064 1996/01/13
Date: AA   0064 1996/01/14
Date: AA   0064 1996/01/15
Date: AA   0064 1996/01/16
Date: AA   0064 1996/08/13
Info: DL   1699 ####/##/## NEW YORK             SAN FRANCISCO
Date: DL   1699 1996/01/09
Date: DL   1699 1996/01/10
```

The contents of the field SFLIGHT-FLDATE in the HEADER part of record type FLIGHT_INFO are displayed as pound signs (#). This is because the logical database fills all of the fields at that hierarchy level with the value HEX 00 when it finishes processing that level. This feature is important for sorting and for processing control levels [Page 343] in extract datasets.

Sorting an Extract

You can sort an extract dataset in much the same way as an internal table by using the following statement:

```
SORT [ASCENDING|DESCENDING] [AS TEXT] [STABLE]
     BY <f1> [ASCENDING|DESCENDING] [AS TEXT]
       ...
         <fn> [ASCENDING|DESCENDING] [AS TEXT].
```

The SORT statement terminates the creation of the extract dataset of a program and, at the same time, sorts its records. Without the BY option, the system sorts the dataset by the key specified in the HEADER field group.

You can sort an extract dataset as often as you like in a program, using any number of different keys. The only prerequisite is that all fields by which you want to sort are contained in the HEADER during the extraction process. You must not use the SORT statement between LOOP and ENDLOOP. However, you can sort and read the extract dataset in any sequence. No further EXTRACT statements may occur after the sort statement, otherwise a runtime error occurs.

You can define a different sort key by using the BY addition. The system then sorts the dataset according to the specified components <f_1> ... <f_n>. These components must either be fields of the HEADER field group or field groups containing only fields from the HEADER field group. The number of key fields is limited to 50. The sequence of the components <f_1> ... <f_n> determines the sort order. The system uses the options you specify before BY as a default for all fields specified behind BY. The options that you specify behind individual fields overwrite for these fields the options specified before BY.

You can define the sort direction using the DESCENDING or ASCENDING additions (ascending is the default direction). For character strings, you can use the AS TEXT addition to define the sort method. This forces an alphabetical sort, as with internal tables [Page 271]. If you want to sort an extract dataset alphabetically more than once, you should include an alphabetically-sortable field in the sort key instead of the text field for performance reasons. To fill this field, use the CONVERT statement [Page 168].

If you put AS TEXT before BY, the addition only applies to type C fields in the sort key. If you place AS TEXT after a field, the field must be of type C. If you place AS TEXT after a field group, the option only applies to the type C fields within the group.

This sorting process is not stable, that is, the old sequence of records with the same sort key must not necessarily be kept. To force a stable sort, use the STABLE addition.

If there is not enough main memory available to sort the data, the system writes data to an external auxiliary file during the sorting process. The name of the file is determined by the SAP profile parameter DIR_SORTTMP.

The SORT statement sorts by all of the fields in the sort key with the contents HEX 00 **before** all of the other entries. This is significant when you use logical databases [Page 1163]. When a logical database has finished reading a hierarchy level, it fills all of the fields at that level with the value HEX 00. Equally, if you use a field list in the GET [Page 958] statement (FIELDS addition), the logical database fills all of the fields not in the field list with HEX 00.

Each sorting process executed on the extract dataset using the SORT statement defines a control level. This is required for subsequent control level processing [Page 343].

Assume the following program is linked to the logical database [Page 1163] F1S.

```
REPORT DEMO.

NODES: SPFLI, SFLIGHT.

FIELD-GROUPS: HEADER, FLIGHT_INFO, FLIGHT_DATE.

INSERT: SPFLI-CARRID SPFLI-CONNID SFLIGHT-FLDATE
    INTO HEADER,
    SPFLI-CITYFROM SPFLI-CITYTO
    INTO FLIGHT_INFO.

START-OF-SELECTION.

GET SPFLI.
  EXTRACT FLIGHT_INFO.

GET SFLIGHT.
  EXTRACT FLIGHT_DATE.

END-OF-SELECTION.

  SORT DESCENDING.

  LOOP.
    AT FIRST.
      WRITE / 'Start of LOOP'.
      ULINE.
    ENDAT.
    AT FLIGHT_INFO WITH FLIGHT_DATE.
      WRITE: / 'Info:',
               SPFLI-CARRID, SPFLI-CONNID, SFLIGHT-FLDATE,
               SPFLI-CITYFROM, SPFLI-CITYTO.
    ENDAT.
    AT FLIGHT_DATE.
      WRITE: / 'Date:',
                SPFLI-CARRID, SPFLI-CONNID, SFLIGHT-FLDATE.
    ENDAT.
    AT LAST.
   ULINE.
      WRITE / 'End of LOOP'.
    ENDAT.
  ENDLOOP.
```

This example is identical with the example in the section Reading an Extract [Page 337], apart from the SORT DESCENDING statement. The SORT statement tells the system to sort the extract dataset in descending order by the three fields of the HEADER field group, before reading it using the loop. The end of the list looks like this:

```
Date: DL   1699 1996/01/10
Date: DL   1699 1996/01/09
Info: AA   0064 ####/##/## SAN FRANCISCO        NEW YORK
Date: AA   0064 1996/08/13
Date: AA   0064 1996/01/16
Date: AA   0064 1996/01/15
Date: AA   0064 1996/01/14
Date: AA   0064 1996/01/13
Date: AA   0064 1996/01/12
Date: AA   0064 1996/01/11
Date: AA   0064 1996/01/10
Date: AA   0064 1996/01/09
Info: AA   0026 ####/##/## FRANKFURT            NEW YORK
Date: AA   0026 1996/01/16
Date: AA   0026 1996/01/15
Date: AA   0026 1996/01/14
Date: AA   0026 1996/01/13
Date: AA   0026 1996/01/12
Date: AA   0026 1996/01/11
Date: AA   0026 1996/01/10
Date: AA   0026 1996/01/09
Info: AA   0017 ####/##/## ATLANTA              SAN FRANCISCO
Date: AA   0017 1996/01/16
Date: AA   0017 1996/01/15
Date: AA   0017 1996/01/14
Date: AA   0017 1996/01/13
Date: AA   0017 1996/01/12
Date: AA   0017 1996/01/11
Date: AA   0017 1996/01/10
Date: AA   0017 1996/01/09

End of LOOP
```

It is worth noting that the records with the value HEX 00 in the field SFLIGHT-FLDATE (undefined characters in the list) are sorted before the remaining records. This is done to preserve the hierarchy of the data from the logical database, independent of the sort sequence.

Processing Control Levels

When you sort an extract dataset, control levels are defined in it. For general information about control levels, refer to Processing Internal Tables in Loops [Page 299] The control level hierarchy of an extract dataset corresponds to the sequence of the fields in the HEADER field group. After sorting, you can use the AT statement within a loop to program statement blocks that the system processes only at a control break, that is, when the control level changes.

AT NEW <f> | AT END OF <f>.
...
ENDAT.

A control break occurs when the value of the field <f> or a superior field in the current record has a different value from the previous record (AT NEW) or the subsequent record (AT END). Field <f> must be part of the HEADER field group.

If the extract dataset is not sorted, the AT... ENDAT block is never executed. Furthermore, all extract records with the value HEX 00 in the field <f> are ignored when the control breaks are determined.

The AT... ENDAT blocks in a loop are processed in the order in which they occur. This sequence should be the same as the sort sequence. This sequence must not necessarily be the sequence of the fields in the HEADER field group, but can also be the one determined in the SORT statement.

If you have sorted an extract dataset by the fields <f1>, <f2>, ..., the processing of the control levels should be written between the other control statements as follows:

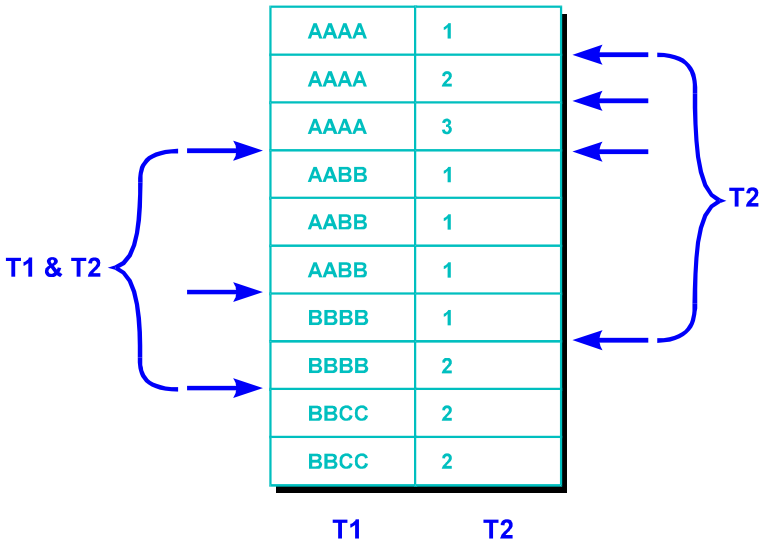
```
LOOP.
  AT FIRST.... ENDAT.
    AT NEW <f1>..... ENDAT.
      AT NEW <f2>..... ENDAT.
     ...
         AT <fgi>..... ENDAT.
         <single line processing without control statement>
    ...
      AT END OF <f2>.... ENDAT.
    AT END OF <f1>.... ENDAT.
  AT LAST..... ENDAT.
ENDLOOP.
```

You do not have to use all of the statement blocks listed here, but only the ones you require.

```
REPORT DEMO.

DATA: T1(4), T2 TYPE I.

FIELD-GROUPS: HEADER.

INSERT T2 T1 INTO HEADER.

T1 ='AABB'. T2 = 1. EXTRACT HEADER.
T1 ='BBCC'. T2 = 2. EXTRACT HEADER.
T1 ='AAAA'. T2 = 2. EXTRACT HEADER.
T1 ='AABB'. T2 = 1. EXTRACT HEADER.
```

```
T1 ='BBBB'. T2 = 2. EXTRACT HEADER.
T1 ='BBCC'. T2 = 2. EXTRACT HEADER.
T1 ='AAAA'. T2 = 1. EXTRACT HEADER.
T1 ='BBBB'. T2 = 1. EXTRACT HEADER.
T1 ='AAAA'. T2 = 3. EXTRACT HEADER.
T1 ='AABB'. T2 = 1. EXTRACT HEADER.

SORT BY T1 T2.

LOOP.

  AT FIRST.
    WRITE 'Start of LOOP'.
    ULINE.
  ENDAT.

  AT NEW T1.
    WRITE / '   New T1:'.
  ENDAT.

  AT NEW T2.
    WRITE / '   New T2:'.
  ENDAT.

  WRITE: /14 T1, T2.

  AT END OF T2.
    WRITE / 'End of T2'.
  ENDAT.

  AT END OF T1.
    WRITE / 'End of T1'.
  ENDAT.

  AT LAST.
    ULINE.
  ENDAT.

ENDLOOP.
```

This program creates a sample extract, containing the fields of the HEADER field group only. After the sorting process, the extract dataset has several control breaks for the control levels T1 and T2, which are indicated in the following figure:



In the loop, the system displays the contents of the dataset and the control breaks it encountered as follows:

```
Start of LOOP

   New T1:
   New T2:
               AAAA          1
End of T2
   New T2:
               AAAA          2
End of T2
   New T2:
               AAAA          3
End of T2
End of T1
   New T1:
   New T2:
               AABB          1
               AABB          1
               AABB          1
End of T2
End of T1
   New T1:
   New T2:
               BBBB          1
End of T2
   New T2:
               BBBB          2
End of T2
End of T1
   New T1:
   New T2:
               BBCC          2
               BBCC          2
End of T2
End of T1
```

Calculating Numbers and Totals

When you read a **sorted** extract dataset using LOOP, you can access two automatically-generated fields CNT(<f>) and SUM(<g>). These fields contain the number of different values and the sums of the numeric fields respectively. The system fills these fields at the end of a control level and after reading the last record of the dataset as follows:

- CNT(<f>)

  If <f> is a non-numeric field of the HEADER field group and the system sorted the extract dataset by <f>, CNT(<f>) contains the number of different values <f> assumed within the control level or entire dataset respectively.

- SUM(<g>)

  If <g> is a numeric field of the extract dataset, SUM (<g>) contains the total of the values of <g> within the control level or entire dataset respectively.

You can access these fields either within the processing blocks following AT END OF or in the processing block following AT LAST, after reading the entire dataset. If you try to access the fields CNT(<f>) and SUM(<g>) without first sorting the dataset, a runtime error may occur.

```
REPORT DEMO.

DATA: T1(4), T2 TYPE I.

FIELD-GROUPS: HEADER, TEST.

INSERT T2 T1 INTO HEADER.

T1 ='AABB'. T2 = 1. EXTRACT TEST.
T1 ='BBCC'. T2 = 2. EXTRACT TEST.
T1 ='AAAA'. T2 = 2. EXTRACT TEST.
T1 ='AABB'. T2 = 1. EXTRACT TEST.
T1 ='BBBB'. T2 = 2. EXTRACT TEST.
T1 ='BBCC'. T2 = 2. EXTRACT TEST.
T1 ='AAAA'. T2 = 1. EXTRACT TEST.
T1 ='BBBB'. T2 = 1. EXTRACT TEST.
T1 ='AAAA'. T2 = 3. EXTRACT TEST.
T1 ='AABB'. T2 = 1. EXTRACT TEST.

SORT BY T1 T2.

LOOP.
  WRITE: /20 T1, T2.

  AT END OF T2.
    ULINE.
    WRITE: 'Sum:', 20 SUM(T2).
    ULINE.
  ENDAT.

  AT END OF T1.
    WRITE: 'Different values:', (6) CNT(T1).
  ULINE.
  ENDAT.
```

```
  AT LAST.
  ULINE.
    WRITE: 'Sum:', 20 SUM(T2),
      / 'Different values:', (6) CNT(T1).
  ENDAT.

ENDLOOP.
```

This program creates a sample extract, containing the fields of the HEADER field group only. After sorting, the system outputs the contents of the dataset, the number of the different T1 fields, and the totals of the T2 fields at the end of each control level and at the end of the loop:

```
                  AAAA          1

Sum:                            1

                  AAAA          2

Sum:                            2

                  AAAA          3

Sum:                            3

Different values:     1

                  AABB          1
                  AABB          1
                  AABB          1

Sum:                            3

Different values:     1

                  BBBB          1

Sum:                            1

                  BBBB          2

Sum:                            2

Different values:     1

                  BBCC          2
                  BBCC          2

Sum:                            4

Different values:     1


Sum:                           16
Different values:     4
```

Formatting Data

This section shows how you can use internal tables and extracts, using an example of how to format data for display as a list. The data that you format is usually read from database tables or from an external file, or you can create your own generic data in the program. Formatting data for a list usually means sorting it, calculating sums, the number of items, and so on. The example of formatted data [Page 351] shows a list whose output data has been processed like this.

There are two ways of formatting data - during and after data retrieval.

Formatting Data During Retrieval

When you create your own datasets or read data from the database using Open SQL statements, the data may already be sufficiently formatted when you read it.

Formatting Data After Retrieval

When using logical databases to access database tables, when reading data from sequential files, or if the options of Open SQL are not comprehensive enough, the retrieved data often appears in a sequence and structure you need to refine. To refine this data later on , you need to store it in a temporary dataset. The refinement process is independent of the process of retrieving data. The flow of a program that can be directly executed [Page 945] is adapted to this procedure.

ABAP allows you to use internal tables and extracts to save temporary datasets.

Internal tables

It makes sense to use internal tables whenever the dataset has the same structure as the underlying data structure, and when you want to access individual entries individually.

Extract Datasets

It makes sense to use an extract dataset whenever you want to process large amounts of data entirely and repeatedly.

Formatting Data During Reading [Page 353]

Formatting Data Using Internal Tables [Page 355]

Formatting Data Using Extracts [Page 359]

Example of Formatted Data

For many report evaluations, the sequence in which you want to process data may differ from the sequence in which it is stored. Since the results of read operations reflect the sequence in which data is stored, you must re-sort all of the data you selected into the desired sequence.

A typical result of formatting data in the context of a flight reservation application is creating a list [Page 771] designed to contain booking information for each flight number. The possible connections are to be sorted by departure city, the flights by date, and the customer data by class and smoker/non-smoker. For each flight, the total number of passengers and the overall luggage weight need to be displayed.

A section of the resulting list should look like this:

Example of Formatted Data

```
LH  2407 from BERLIN          to FRANKFURT

Date: 1996/01/09   Book-ID              Smoker    Class

                   00000002                       C
                   00000011                       C
                   00000013                       C
                   00000014                       C
                   00000025                       C
                   00000007              X        C
                   00000022              X        C
                   00000006                       F
                   00000010                       F
                   00000016                       F
                   00000027                       F
                   00000030                       F
                   00000003              X        F
                   00000019              X        F
                   00000023              X        F
                   00000001                       Y
                   00000004                       Y
                   00000005                       Y
                   00000009                       Y
                   00000012                       Y
                   00000015                       Y
                   00000017                       Y
                   00000020                       Y
                   00000021                       Y
                   00000026                       Y
                   00000029                       Y
                   00000031                       Y
                   00000032                       Y
                   00000008              X        Y
                   00000018              X        Y
                   00000024              X        Y
                   00000028              X        Y

Number of bookings:   32
Total luggage weight: 622 KG

Date: 1996/01/10   Book-ID              Smoker    Class

                   00000007                       C
                   00000011                       C
```

Formatting Data During Reading

The most direct method of formatting data is to use the corresponding options of the SELECT statement. With this method, you must program the database access yourself. In addition, you must program a selection screen to offer the user the possibility of restricting the set of data to be read. Note that using nested SELECT loops is exceedingly **inefficient**. You should therefore place your data in blocks into internal tables or extracts, and process it from there.

```
REPORT DEMO.

DATA: SUM TYPE I, CNT TYPE I,
      WA_SPFLI TYPE SPFLI,
      WA_SFLIGHT TYPE SFLIGHT,
      WA_SBOOK TYPE SBOOK.

SELECT * FROM SPFLI INTO WA_SPFLI
                    ORDER BY CITYFROM CITYTO CONNID.

  SKIP.
  WRITE: / WA_SPFLI-CARRID,
           WA_SPFLI-CONNID,
           'from', (15) WA_SPFLI-CITYFROM,
           'to',   (15) WA_SPFLI-CITYTO.
  ULINE.

  SELECT * FROM SFLIGHT INTO WA_SFLIGHT
                        WHERE CARRID = WA_SPFLI-CARRID
                          AND CONNID = WA_SPFLI-CONNID
                        ORDER BY FLDATE.

    SKIP.
    WRITE: / 'Date:', WA_SFLIGHT-FLDATE.
    WRITE: 20 'Book-ID', 40 'Smoker', 50 'Class'.
    ULINE.

    SUM = 0.
    CNT = 0.
    SELECT * FROM SBOOK INTO WA_SBOOK
                        WHERE CARRID = WA_SFLIGHT-CARRID
                          AND CONNID = WA_SFLIGHT-CONNID
                          AND FLDATE = WA_SFLIGHT-FLDATE
                        ORDER BY CLASS SMOKER BOOKID.

      WRITE: / WA_SBOOK-BOOKID UNDER 'Book-ID',
               WA_SBOOK-SMOKER UNDER 'Smoker',
               WA_SBOOK-CLASS  UNDER 'Class'.

    SUM = SUM + WA_SBOOK-LUGGWEIGHT.
    CNT = CNT + 1.

    ENDSELECT.

    ULINE.
    WRITE: 'Number of bookings:  ', (3) CNT,
         / 'Total luggage weight:', (3) SUM, WA_SBOOK-WUNIT.
```

```
  ENDSELECT.
ENDSELECT.
```

This program creates the list as in the Example of Formatted Data [Page 351]. It uses the ORDER BY addition in the SELECT statement to sort the data.

Refining Data Using Internal Tables

When storing data in internal tables, you often use one internal table for each database you read. Each one contains some or all columns of the relevant database table. It is up to you whether you create an internal table with a flat structure for each database table or if you create, for example, internal tables with nested structures. If you have several tables, each one with a flat structure, you have to work with redundant key fields to link the tables. If, on the other hand, you use nested internal tables, you can store the data from the database tables hierarchically.

Saving and processing very large amounts of data in internal tables has disadvantages. If you divide up the data into different internal tables, processing it can be very runtime-intensive, since the tables have to be processed individually. Furthermore, it requires a lot of storage space, since internal tables are not stored in compressed form. The system may even need to store the dataset outside of its working memory. This means that processing it takes even longer.

Processing Data Using Flat Internal Tables

Assume the following program is linked to the logical database [Page 1163] F1S.

```
REPORT DEMO.

DATA: SUM TYPE I, CNT TYPE I.

NODES: SPFLI, SFLIGHT, SBOOK.

DATA: TAB_SPFLI   TYPE TABLE OF SPFLI,
      TAB_SFLIGHT TYPE TABLE OF SFLIGHT,
      TAB_SBOOK   TYPE TABLE OF SBOOK.

DATA: WA_SPFLI   LIKE LINE OF TAB_SPFLI,
      WA_SFLIGHT LIKE LINE OF TAB_SFLIGHT,
      WA_SBOOK   LIKE LINE OF TAB_SBOOK.

START-OF-SELECTION.

GET SPFLI.
  APPEND SPFLI TO TAB_SPFLI.

GET SFLIGHT.
  APPEND SFLIGHT TO TAB_SFLIGHT.

GET SBOOK.
  APPEND SBOOK TO TAB_SBOOK.

END-OF-SELECTION.

  SORT: TAB_SPFLI   BY CITYFROM CITYTO CONNID,
        TAB_SFLIGHT BY FLDATE,
        TAB_SBOOK   BY CLASS SMOKER BOOKID.

  LOOP AT TAB_SPFLI INTO WA_SPFLI.

    SKIP.
    WRITE: / WA_SPFLI-CARRID,
             WA_SPFLI-CONNID,
             'from', (15) WA_SPFLI-CITYFROM,
```

```
            'to',    (15) WA_SPFLI-CITYTO.
    ULINE.

    LOOP AT TAB_SFLIGHT INTO WA_SFLIGHT
                        WHERE CARRID = WA_SPFLI-CARRID
                          AND CONNID = WA_SPFLI-CONNID.

      SKIP.
      WRITE: / 'Date:', WA_SFLIGHT-FLDATE.
      WRITE: 20 'Book-ID', 40 'Smoker', 50 'Class'.
      ULINE.

      SUM = 0.
      CNT = 0.
      LOOP AT TAB_SBOOK INTO WA_SBOOK
                        WHERE CARRID = WA_SFLIGHT-CARRID
                          AND CONNID = WA_SFLIGHT-CONNID
                          AND FLDATE = WA_SFLIGHT-FLDATE.

        WRITE: / WA_SBOOK-BOOKID UNDER 'Book-ID',
                 WA_SBOOK-SMOKER UNDER 'Smoker',
                 WA_SBOOK-CLASS  UNDER 'Class'.

                 SUM = SUM + WA_SBOOK-LUGGWEIGHT.
                 CNT = CNT + 1.

      ENDLOOP.

      ULINE.
      WRITE: 'Number of bookings:  ', (3) CNT,
           / 'Total luggage weight:',
          (3) SUM, WA_SBOOK-WUNIT.

    ENDLOOP.

  ENDLOOP.
```

This program creates the same list as in the Example of Formatted Data [Page 351].

The GET [Page 958] events that retrieve the data are clearly separated from the sorting process. The three internal tables have exactly the same structure and contents as the corresponding database tables. The data is sorted and then displayed. The loop structure is exactly the same as that of the SELECT loops in the example from the Formatting Data During Reading [Page 353] section.

Formatting Data Using Nested Internal Tables

Assume the following program is linked to the logical database [Page 1163] F1S.

```
REPORT DEMO.

DATA: SUM TYPE I, CNT TYPE I.

NODES: SPFLI, SFLIGHT, SBOOK.

DATA: BEGIN OF WA_SBOOK,
        BOOKID     TYPE SBOOK-BOOKID,
        SMOKER     TYPE SBOOK-SMOKER,
```

```
          CLASS      TYPE SBOOK-CLASS,
          LUGGWEIGHT TYPE SBOOK-LUGGWEIGHT,
          WUNIT      TYPE SBOOK-WUNIT,
        END OF WA_SBOOK.

DATA: BEGIN OF WA_SFLIGHT,
        FLDATE TYPE SFLIGHT-FLDATE,
        SBOOK  LIKE TABLE OF WA_SBOOK,
      END OF WA_SFLIGHT.

DATA: BEGIN OF WA_SPFLI,
        CARRID   TYPE SPFLI-CARRID,
        CONNID   TYPE SPFLI-CONNID,
        CITYFROM TYPE SPFLI-CITYFROM,
        CITYTO   TYPE SPFLI-CITYTO,
        SFLIGHT  LIKE TABLE OF WA_SFLIGHT,
      END OF WA_SPFLI.

DATA TAB_SPFLI LIKE TABLE OF WA_SPFLI.

START-OF-SELECTION.

GET SPFLI.
  REFRESH WA_SPFLI-SFLIGHT.

GET SFLIGHT.
  REFRESH WA_SFLIGHT-SBOOK.

GET SBOOK.
  MOVE-CORRESPONDING SBOOK TO WA_SBOOK.
  APPEND WA_SBOOK TO WA_SFLIGHT-SBOOK.

GET SFLIGHT LATE.
  MOVE-CORRESPONDING SFLIGHT TO WA_SFLIGHT.
  APPEND WA_SFLIGHT TO WA_SPFLI-SFLIGHT.

GET SPFLI LATE.
  MOVE-CORRESPONDING SPFLI TO WA_SPFLI.
  APPEND WA_SPFLI TO TAB_SPFLI.

END-OF-SELECTION.

  SORT TAB_SPFLI BY CITYFROM CITYTO CONNID.

  LOOP AT TAB_SPFLI INTO WA_SPFLI.

    SKIP.
    WRITE: / WA_SPFLI-CARRID,
             WA_SPFLI-CONNID,
             'from', (15) WA_SPFLI-CITYFROM,
             'to',   (15) WA_SPFLI-CITYTO.
    ULINE.

    SORT WA_SPFLI-SFLIGHT BY FLDATE.

    LOOP AT WA_SPFLI-SFLIGHT INTO WA_SFLIGHT.

      SKIP.
      WRITE: / 'Date:', WA_SFLIGHT-FLDATE.
      WRITE: 20 'Book-ID', 40 'Smoker', 50 'Class'.
      ULINE.
```

```
      SORT WA_SFLIGHT-SBOOK BY CLASS SMOKER BOOKID.

      SUM = 0.
      CNT = 0.

      LOOP AT WA_SFLIGHT-SBOOK INTO WA_SBOOK.

        WRITE: / WA_SBOOK-BOOKID UNDER 'Book-ID',
                 WA_SBOOK-SMOKER UNDER 'Smoker',
                 WA_SBOOK-CLASS  UNDER 'Class'.

        SUM = SUM + WA_SBOOK-LUGGWEIGHT.
        CNT = CNT + 1.

      ENDLOOP.

      ULINE.
      WRITE: 'Number of bookings:  ', (3) CNT,
           / 'Total luggage weight:',
          (3) SUM, WA_SBOOK-WUNIT.

    ENDLOOP.

  ENDLOOP.
```

This program creates the same list as in the Example of Formatted Data [Page 351].

During the GET [Page 958] events, the system reads the data into the three-level internal table SORT_SPFLI which contains the substructure SFLIGHT and its substructure SBOOK. The sorting process takes place on the individual nesting levels.

This way of programming does not require key relations between the internal tables (no WHERE conditions), but it is more complex than using flat internal tables. And the increased internal administration effort has a negative effect on the storage space required as well as on the runtime.

Formatting Data Using Extracts

You can use a single extract to store entries from more than one database table, since the records in an extract can have different structures.

Assume the following program is linked to the logical database [Page 1163] F1S.

```
REPORT DEMO.

NODES: SPFLI, SFLIGHT, SBOOK.

FIELD-GROUPS: HEADER, FLIGHT_INFO, FLIGHT_BOOKING.

INSERT:
  SPFLI-CITYFROM SPFLI-CITYTO
  SPFLI-CONNID SFLIGHT-FLDATE
  SBOOK-CLASS SBOOK-SMOKER SBOOK-BOOKID INTO HEADER,
  SPFLI-CARRID                    INTO FLIGHT_INFO,
  SBOOK-LUGGWEIGHT SBOOK-WUNIT        INTO FLIGHT_BOOKING.

START-OF-SELECTION.

GET SPFLI.
  EXTRACT FLIGHT_INFO.

GET SFLIGHT.

GET SBOOK.
  EXTRACT FLIGHT_BOOKING.

END-OF-SELECTION.

  SORT.

  LOOP.
    AT FLIGHT_INFO.
      SKIP.
      WRITE: / SPFLI-CARRID,
               SPFLI-CONNID,
               'from', (15) SPFLI-CITYFROM,
               'to',   (15) SPFLI-CITYTO.
      ULINE.
    ENDAT.
    AT NEW SFLIGHT-FLDATE.
       SKIP.
       WRITE: / 'Date:', SFLIGHT-FLDATE.
       WRITE: 20 'Book-ID', 40 'Smoker', 50 'Class'.
       ULINE.
    ENDAT.
    AT FLIGHT_BOOKING.
       WRITE: / SBOOK-BOOKID UNDER 'Book-ID',
                SBOOK-SMOKER UNDER 'Smoker',
                SBOOK-CLASS  UNDER 'Class'.
    ENDAT.
    AT END OF SFLIGHT-FLDATE.
```

```
    ULINE.
    WRITE: 'Number of bookings:  ', (3) CNT(SBOOK-BOOKID),
        / 'Total luggage weight:',
           SUM(SBOOK-LUGGWEIGHT), SBOOK-WUNIT.
  ENDAT.
ENDLOOP.
```

This program creates the same list as in the Example of Formatted Data [Page 351].

The system creates three field groups and fills them with several fields. It then fills the extract in the EXTRACT statements in the GET [Page 958] events. Note that there is no EXTRACT statement for GET SFLIGHT, since the required field SFLIGHT_FLDATE is part of the HEADER field group and thus automatically extracted for each subordinate event GET SBOOK.

After retrieving the data, the system terminates the creation of the dataset when the SORT statement occurs, and sorts the dataset by the HEADER sort key. In the LOOP-ENDLOOP block, it writes the sorted extract dataset to the output list, using various AT ... ENDAT blocks and the fields CNT(...) and SUM(...).

Saving Data Externally

This section describes how to save and read data externally from ABAP programs. These techniques are additional to those for storing data in the central database in the R/3 System.

Saving Data Objects as Clusters [Page 362]

Working with Files [Page 384]

Saving Data Objects as Clusters

You can combine any number of internal data objects of any complexity from an ABAP program into a data cluster that you can then store either temporarily in ABAP memory or persistently in the database.

The following sections describe how to store data clusters in memory or in the database:

Data Clusters in ABAP Memory [Page 363]

Data Clusters in the Database [Page 368]

Data Clusters in ABAP Memory

You can store data clusters in ABAP memory. ABAP memory is a memory area within the internal session (roll area) of an ABAP program and any other program called from it using CALL TRANSACTION or SUBMIT.

ABAP memory is independent of the ABAP program or program module from which it was generated. In other words, an object saved in ABAP memory can be read from any other ABAP program in the same call chain. ABAP memory is not the same as the cross-transaction global SAP memory. For further information, refer to Passing Data Between Programs [Page 1032].

This allows you to pass data from one module to another over several levels of the program hierarchy. For example, you can pass data

- From an executable program (report) to another executable program called using SUBMIT.
- From a transaction to an executable program (report).
- Between dialog modules.
- From a program to a function module.

and so on.

The contents of the memory are released when you leave the transaction.

To save data objects in ABAP memory, use the statement EXPORT TO MEMORY.

Saving Data Objects in Memory [Page 364]

To read data objects from memory, use the statement IMPORT FROM MEMORY.

Reading Data Objects from Memory [Page 365]

To delete data clusters from memory, use the statement FREE MEMORY.

Deleting Data Clusters from Memory [Page 367]

Saving Data Objects in Memory

To read data objects from an ABAP program into ABAP memory, use the following statement:

Syntax

EXPORT <f_1> [FROM <g_1>] <f_2> [FROM <g_2>] ... TO MEMORY ID <key>.

This statement stores the data objects specified in the list as a cluster in memory. If you do not use the option FROM <f_i>, the data object <f_i> is saved under its own name. If you use the FROM <g_i> option, the data objet <g_i> is saved under the name <f_i>. The name <key> identifies the cluster in memory. It may be up to 32 characters long.

The EXPORT statement always completely overwrites the contents of any existing data cluster with the same name <key>.

If you are using internal tables with header lines, you can only store the table itself, not the header line. In the EXPORT statement, the table name is interpreted as the table. This is an exception to the general rule, under which statements normally interpret the table name as a table work area (see Choosing a Table Type [Page 276]).

```
PROGRAM SAPMZTS1.

DATA TEXT1(10) VALUE 'Exporting'.

DATA ITAB LIKE SBOOK OCCURS 10 WITH HEADER LINE.

DO 5 TIMES.
  ITAB-BOOKID = 100 + SY-INDEX.
  APPEND ITAB.
ENDDO.

EXPORT TEXT1
       TEXT2 FROM 'Literal'
  TO MEMORY ID 'text'.

EXPORT ITAB
  TO MEMORY ID 'table'.
```

In this example, the text fields TEXT1 and TEXT2 are stored in the ABAP memory of program SAPMZTS1 under the name “text”. The internal table ITAB is stored under the name “table”.

Reading Data Objects from Memory

To read data objects from ABAP memory into an ABAP program, use the following statement:

Syntax

IMPORT <f_1> [TO <g_1>] <f_2> [TO <g_2>] ... FROM MEMORY ID <key>.

This statement reads the data objects specified in the list from a cluster in memory. If you do not use the TO <g_i> option, the data object <f_i> in memory is assigned to the data object in the program with the same name. If you do use the option, the data object <f_i> is read from memory into the field <g_i>. The name <key> identifies the cluster in memory. It may be up to 32 characters long.

You do not have to read all of the objects stored under a particular name <key>. You can restrict the number of objects by specifying their names. If the memory does not contain any objects under the name <key>, SY-SUBRC is set to 4. If, on the other hand, there is a data cluster in memory with the name <key>, SY-SUBRC is always 0, regardless of whether it contained the data object <f_i>. If the cluster does not contain the data object <f_i>, the target field remains unchanged.

In this statement, the system does not check whether the structure of the object in memory is compatible with the structure into which you are reading it. The data is transported bit by bit. If the structures are incompatible, the data in the target field may be incorrect.

```
PROGRAM SAPMZTS1.

DATA TEXT1(10) VALUE 'Exporting'.

DATA ITAB LIKE SBOOK OCCURS 10 WITH HEADER LINE.

DO 5 TIMES.
  ITAB-BOOKID = 100 + SY-INDEX.
  APPEND ITAB.
ENDDO.

EXPORT TEXT1
       TEXT2 FROM 'Literal'
  TO MEMORY ID 'text'.

EXPORT ITAB
  TO MEMORY ID 'table'.

SUBMIT SAPMZTS2 AND RETURN.

SUBMIT SAPMZTS3.
```

The first part of this program is the same as the example in the section Saving Data Objects in Memory [Page 364]. In the example, the programs SAPMZTS1 and SAPMZTS2 are called using SUBMIT. You can create and maintain the programs called using the SUBMIT statement by double-clicking their names in the statement. For further information about the SUBMIT statement, refer to Calling Executable Programs (Reports) [Page 1018]

Example for SAPMZTS2:

```
PROGRAM SAPMZTS2.
```

```
DATA: TEXT1(10),
      TEXT3 LIKE TEXT1 VALUE 'Initial'.

IMPORT TEXT3 FROM MEMORY ID 'text'.
WRITE: / SY-SUBRC, TEXT3.

IMPORT TEXT2 TO TEXT1 FROM MEMORY ID 'text'.
WRITE: / SY-SUBRC, TEXT1.
```

Example for SAPMZTS3:

```
PROGRAM SAPMZTS3.

DATA JTAB LIKE SBOOK OCCURS 10 WITH HEADER LINE.

IMPORT ITAB TO JTAB FROM MEMORY ID 'table'.

LOOP AT JTAB.
  WRITE / JTAB-BOOKID.
ENDLOOP.
```

The output is displayed on two successive screens. It looks like this:

```
   0  Initial
   0  Literal
```

and

```
00000101
00000102
00000103
00000104
00000105
```

The program SAPMZTS2 attempts to read a data object TEXT3 from the data cluster “text”, which does not exist. TEXT3 therefore remains unchanged. The existing data object TEXT2 is placed in TEXT1. In both cases, SY-SUBRC is 0, since the cluster “text” contains data.

The program SAPMZTS3 reads the internal table ITAB from the cluster “table” into the internal table JTAB. Both tables have the same structure, namely that of the ABAP Dictionary table SBOOK.

Deleting Data Clusters from Memory

To delete data objects from ABAP memory, use the following statement:

Syntax

FREE MEMORY [ID <key>].

If you omit the addition ID <key>, the system deletes the entire memory, that is, all of the data clusters previously stored in ABAP memory using EXPORT. If you use the addition ID <key>, this statement only deletes the data cluster with the name <key>.

Only use the FREE MEMORY statement with the ID addition, since deleting the entire memory can also delete the memory contents of system routines.

```
PROGRAM SAPMZTST.

DATA: TEXT(10) VALUE '0123456789',
      IDEN(3)  VALUE 'XYZ'.

EXPORT TEXT TO MEMORY ID IDEN.

TEXT = 'xxxxxxxxxx'.
IMPORT TEXT FROM MEMORY ID IDEN.
WRITE: / SY-SUBRC, TEXT.

FREE MEMORY.

TEXT = 'xxxxxxxxxx'.
IMPORT TEXT FROM MEMORY ID IDEN.
WRITE: / SY-SUBRC, TEXT.
```

This produces the following output:

```
0  0123456789
4  xxxxxxxxxx
```

The FREE MEMORY statement deletes the data cluster “XYZ”. Consequently, SY-SUBRC is 4 after the following IMPORT statement, and the target field remains unchanged.

Data Clusters in the Database

You can store data clusters in special databases in the ABAP Dictionary. These are called ABAP cluster databases, and have a prescribed structure:

Cluster Databases [Page 369]

This method allows you to store complex data objects with deep structures in a single step, without having to adjust them to conform to the flat structure of a relational database. Your data objects are then available systemwide to every user. To read these objects from the database successfully, you must know their data types.

You can use cluster databases to store the results of analyses of data from the relational database. For example, if you want to create a list of your customers with the highest revenue, or an address list from the personnel data of all of your branches, you can write ABAP programs to generate the list and store it as a data cluster. To update the data cluster, you can schedule the program to run periodically as a background job. You can then write other programs that read from the data cluster and work with the results. This method can considerable reduce the response time of your system, since it means that you do not have to access the distributed data in the relational database tables each time you want to look at your list.

Data clusters are specific to ABAP. Although it is possible to read a cluster database using SQL statements, only ABAP can interpret the structure of the data cluster.

To save data objects in a cluster database, use the statement EXPORT TO DATABASE.

Saving Data Objects in Cluster Databases [Page 374]

To create a directory of data clusters and read data objects from cluster databases, use the statement IMPORT FROM DATABASE.

Creating a Directory of a Data Cluster [Page 376]

Reading Data Objects From Cluster Databases [Page 378]

To delete a data cluster from a cluster database, use the statement DELETE FROM DATABASE.

Deleting Data Clusters from Cluster Databases [Page 380]

For further information about how to access cluster databases using Open SQL, refer to:

Using Open SQL Statements to Access Cluster Databases [Page 382]

Cluster Databases

Cluster databases are special relational databases in the ABAP Dictionary that you can use to store data clusters. Their line structure is divided into a standard section, containing several fields, and one large field for the data cluster.

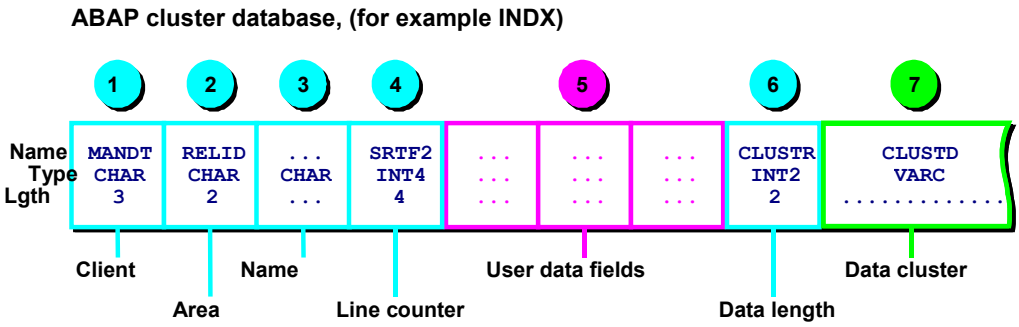
The following sections describe the rules for creating a cluster database, and introduce the system cluster database INDX:

Structure of a Cluster Database [Page 370]

Example of a Cluster Database [Page 372]

Structure of a Cluster Database

Cluster databases have the following structure:



The rules for the structure of a cluster database are listed below. The fields listed in steps 1 to 4 must be created as key fields. The data types listed are ABAP Dictionary types.

1. If the table is client-dependent, the first field must have the name MANDT, type CHAR, and length 3 bytes. It is used for the client number. The system fills the MANDT field when you save a data cluster - either automatically with the current client or with a client that you explicitly specify in the EXPORT statement.
2. The next field (the first field in the case of client-independent tables) must have the name RELID, type CHAR, and length 2 bytes. This contains an area ID. Cluster databases are divided into different areas. The system fills the field RELID with the area ID specified in the EXPORT statement when it saves a data cluster.
3. The next field has type CHAR and variable length. It contains the name <key> of the cluster, specified in the program with the addition ID in the EXPORT statement. Since the next field is aligned, the system may fill out the field RELID with up to three unused bytes. When you create your own cluster databases, you should plan for the length of this field.
4. The next field must have the name SRTF2, the type INT4, and length 4. Single data clusters can extend over several lines of the database table. Theoretically, up to 2**31 lines are possible. The field SRTF2 contains the current line number within the data cluster. Its value can be from 0 to 2**31 -1. The field is filled automatically by the system when you save a data cluster (see step 7).
5. After SRTF2, you can include any number of user data fields of any name and type. The system does not automatically fill these fields when you save a cluster. Instead, you must assign values to them explicitly in the program before the EXPORT statement. These fields normally contain administrative information such as program name and user ID.
6. The penultimate field of a line must have the name CLUSTR and the type INT2 with length 2. This contains the length of the data in the following field CLUSTD. The field is filled automatically by the system when you save a data cluster.
7. The last field in the line must have the name CLUSTD and the type VARC. You can define it with any length. It is usually around 1000 bytes long. This is the field in which the system saves the actual data in the data cluster. The data is saved in compressed form. If CLUSTD is not long enough to accommodate a data cluster, it is split across two or more lines. The lines are numbered in field SRTF2 (see step 4).

You can either create your own cluster databases by following the above rules (for further information, refer to the ABAP Dictionary [Ext.] documentation), or you can use the system cluster database INDX.

Example of a Cluster Database [Page 372]

Example of a Cluster Database

The database INDX is an example of a cluster database. It comes installed in the standard R/3 System, and is intended for customer use. The advantage of using this database is that you do not have to create your own. However, it does mean that all users can access, change, and delete the data that you store in it.

To display the structure of the database INDX, choose *Edit* → *More functions* → *Enter command* in the ABAP Editor, followed by SHOW INDX, or double-click INDX in a declarative statement such as TABLES.

Name	Key	Typ		Länge	/Dez
INDX-MANDT	X	CLNT	C	3	
INDX-RELID	X	CHAR	C	2	
INDX-SRTFD	X	CHAR	C	22	
INDX-SRTF2	X	INT4	X	4	
INDX-LOEKZ		CHAR	C	1	
INDX-SPERR		CHAR	C	1	
INDX-AEDAT		DATS	D	8	
INDX-USERA		CHAR	C	12	
INDX-PGMID		CHAR	C	8	
INDX-BEGDT		DATS	D	8	
INDX-ENDDT		DATS	D	8	
INDX-CLUSTR		INT2	X	2	
INDX-CLUSTD		LRAW	X	2886	

For each field, you see the ABAP Dictionary data type and the corresponding ABAP data type (that is, the data type of the components of the table work area created using the TABLES statement).

The first four fields are the key fields of the table INDX, and correspond precisely to the description in Structure of a Cluster Database [Page 370]. The third field for the cluster name has the name SRTDF and length 22 bytes. This means that <key> in the ID addition of the EXPORT statement can only contain 22 characters.

The next seven fields are not standard fields. They are used for user data:

- AEDAT: Date changed
- USERA: User name
- PGMID: Program name

The last two fields are an obligatory part of your structure. In table INDX, the field CLUSTD, which contains the actual data clusters, has the length 2886 bytes.

For examples of table INDX in use, refer to

Saving Data Objects in Cluster Databases [Page 374]

Saving Data Objects in Cluster Databases

To save data objects from an ABAP program in a cluster database, use the following statement:

Syntax

EXPORT <f_1> [FROM <g_1>] <f_2> [FROM <g_2>] ...
    TO DATABASE <dbtab>(<ar>) [CLIENT <cli>] ID <key>.

This statement stores the data objects specified in the list as a cluster in the cluster database <dbtab>. You must declare <dbtab> using a TABLES statement. If you do not use the option FROM <f_i>, the data object <f_i> is saved under its own name. If you use the FROM <g_i> option, the data objet <g_i> is saved under the name <f_i>.

<ar> is the two-character area ID for the cluster in the database
(refer to point to in Structure of a Cluster Database [Page 370]).

The name <key> identifies the data in the database. Its maximum length depends on the length of the name field in <dbtab>.
(See also point 3 in Structure of a Cluster Database [Page 370]).

The CLIENT <cli> option allows you to disable the automatic client handling of a client-specific cluster database, and specify the client yourself. The addition must always come directly after the name of the database.
(See also point 1 in Structure of a Cluster Database [Page 370]).

The EXPORT statement also transports the contents of the user fields of the table work area <dbtab> into the database table. You can fill these fields yourself beforehand.
(See also point 5 in Structure of a Cluster Database [Page 370]).

The EXPORT statement always completely overwrites the contents of any existing data cluster in the same area <ar> with the same name <key> in the same client <cli>.

If you are using internal tables with header lines, you can only store the table itself, not the header line. In the EXPORT statement, the table name is interpreted as the table. This is an exception to the general rule, under which statements normally interpret the table name as a table work area (see Choosing a Table Type [Page 276]).

```
PROGRAM SAPMZTS1.

TABLES INDX.

DATA: BEGIN OF ITAB OCCURS 100,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF ITAB.

DO 3000 TIMES.
  ITAB-COL1 = SY-INDEX.
  ITAB-COL2 = SY-INDEX ** 2.
  APPEND ITAB.
ENDDO.
```

```
INDX-AEDAT = SY-DATUM.
INDX-USERA = SY-UNAME.
INDX-PGMID = SY-REPID.

EXPORT ITAB TO DATABASE INDX(HK) ID 'Table'.

WRITE: '    SRTF2',
    AT 20 'AEDAT',
    AT 35 'USERA',
    AT 50 'PGMID'.
ULINE.

SELECT * FROM INDX WHERE RELID = 'HK'
                AND   SRTFD = 'Table'.

  WRITE: / INDX-SRTF2 UNDER 'SRTF2',
         INDX-AEDAT UNDER 'AEDAT',
         INDX-USERA UNDER 'USERA',
         INDX-PGMID UNDER 'PGMID'.

ENDSELECT.
```

This example fills an internal table ITAB with 3000 lines, assigns values to some of the user fields in INDX, and then exports ITAB to INDX.

Since INDX is a relational database, you can address it using Open SQL statements. The SELECT statement in the example uses appropriate WHERE conditions to select lines transferred to the database in the EXPORT statement.

The output of some of the database fields is as follows:

SRTF2	AEDAT	USERA	PGMID
0	07.12.1995	KELLERH	SAPMZTST
1	07.12.1995	KELLERH	SAPMZTST
2	07.12.1995	KELLERH	SAPMZTST
3	07.12.1995	KELLERH	SAPMZTST
4	07.12.1995	KELLERH	SAPMZTST
5	07.12.1995	KELLERH	SAPMZTST

This shows that the user fields AEDAT, USERA, and PGMID were transferred in the EXPORT statement, and that the data cluster contained in ITAB extends over 6 lines. If you change the figure 3000 in the DO statement, the number of lines occupied by the data cluster would also change.

Creating a Directory of a Data Cluster

To create a directory of a data cluster from an ABAP cluster database, use the following statement:

Syntax

```
IMPORT DIRECTORY INTO <dirtab>
       FROM DATABASE <dbtab>(<ar>)
       [CLIENT <cli>] ID <key>.
```

This creates a directory of the data objects belonging to a data cluster in the database <dbtab> in the internal table <dirtab>. You must declare <dbtab> using a TABLES statement.

To save a data cluster in a database, use the EXPORT TO DATABASE statement (refer to Saving Data Objects in a Cluster Database [Page 374]). For information about the structure of the database table <dbtab>, refer to the section Structure of Cluster Databases [Page 370].

For <ar>, enter the two-character area ID for the cluster in the database. The name <key> identifies the data in the database. Its maximum length depends on the length of the name field in <dbtab>. The CLIENT <cli> option allows you to disable the automatic client handling of a client-specific cluster database, and specify the client yourself. The addition must always come directly after the name of the database.

The IMPORT statement also reads the contents of the user fields from the database table.

If the system is able to create a directory, SY-SUBRC is set to 0, otherwise to 4.

The internal table <dirtab> must have the ABAP Dictionary structure CDIR. You can declare it using the TYPE addition in the DATA statement. CDIR has the following components:

Field name	Type	Description
NAME	CHAR	Name of the object in the cluster
OTYPE	CHAR	Object type: F: elementary field R: structure T: internal table
FTYPE	CHAR	Data type of object: Structured data types return type C
TFILL	INT4	Number of filled lines in internal tables
FLENG	INT2	Length of field or structure

```
PROGRAM SAPMZTS2.

TABLES INDX.

DATA DIRTAB LIKE CDIR OCCURS 10 WITH HEADER LINE.

IMPORT DIRECTORY INTO DIRTAB FROM DATABASE
                    INDX(HK) ID 'Table'.
```

```
IF SY-SUBRC = 0.
  WRITE: / 'AEDAT:', INDX-AEDAT,
         / 'USERA:', INDX-USERA,
         / 'PGMID:', INDX-PGMID.
  WRITE  / 'Directory:'.
  LOOP AT DIRTAB.
    WRITE: / DIRTAB-NAME,  DIRTAB-OTYPE, DIRTAB-FTYPE,
             DIRTAB-TFILL, DIRTAB-FLENG.
  ENDLOOP.
ELSE.
  WRITE 'Not found'.
ENDIF.
```

This example creates a directory of the contents of the data cluster created using the example program in the section Saving Data Objects in Cluster Databases [Page 374]. The output appears as follows:

```
AEDAT: 07.12.1995
USERA: KELLERH
PGMID: SAPMZTST
Directory:
ITAB                            T C     3.000     8
```

The directory DIRTAB contains one line, showing that the data cluster contains an internal table called ITAB, with 3000 lines, each of length 8.

Reading Data Objects From Cluster Databases

To read data objects from an ABAP cluster database into an ABAP program, use the following statement:

Syntax

IMPORT <f_1> [TO <g_1>] <f_2> [TO <g_2>] ...
FROM DATABASE <dbtab>(<ar>)
[CLIENT <cli>] ID <key>|MAJOR-ID <maid> [MINOR-ID <miid>].

This statement reads the data objects specified in the list from a cluster in the database <dbtab>. You must declare <dbtab> using a TABLES statement. If you do not use the TO <g_i> option, the data object <f_i> in the database is assigned to the data object in the program with the same name. If you do use the option, the data object <f_i> is read from the database into the field <g_i>.

To save a data cluster in a database, use the EXPORT TO DATABASE statement (refer to Saving Data Objects in a Cluster Database [Page 374]). For information about the structure of the database table <dbtab>, refer to the section Structure of Cluster Databases [Page 370].

For <ar>, enter the two-character area ID for the cluster in the database. The name <key> identifies the data in the database. Its maximum length depends on the length of the name field in <dbtab>. You can replace the ID <key> addition with MAJOR-ID <maid>. The system then selects the data cluster whose name corresponds to <maid>. If you also use the MINOR-ID <miid> addition, the system searches for the data cluster whose name is greater than or equal to <miid> starting at the position following the length of <maid>. The CLIENT <cli> option allows you to disable the automatic client handling of a client-specific cluster database, and specify the client yourself. The addition must always come directly after the name of the database.

The IMPORT statement also reads the contents of the user fields from the database table.

You do not have to read all of the objects stored under a particular name <key>. You can restrict the number of objects by specifying their names. If the database does not contain any objects that correspond to the key <ar>, <key>, and <cli>, SY-SUBRC is set to 4. If, on the other hand, there is a data cluster in the database with the same key, SY-SUBRC is always 0, regardless of whether it contained the data object <f_i>. If the cluster does not contain the data object <f_i>, the target field remains unchanged.

In this statement, the system does not check whether the structure of the object in the database is compatible with the structure into which you are reading it. If this is not the case, a runtime error occurs. Exceptions are fields with type C, which may occur at the end of a structured data object. These can be extended, shortened, added, or omitted.

```
PROGRAM SAPMZTS3.

TABLES INDX.

DATA: BEGIN OF JTAB OCCURS 100,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF JTAB.

IMPORT ITAB TO JTAB FROM DATABASE INDX(HK) ID 'Table'.
```

```
WRITE: / 'AEDAT:', INDX-AEDAT,
       / 'USERA:', INDX-USERA,
       / 'PGMID:', INDX-PGMID.

SKIP.
WRITE 'JTAB:'.

LOOP AT JTAB FROM 1 TO 5.
  WRITE: / JTAB-COL1, JTAB-COL2.
ENDLOOP.
```

This example is based on the example program in the section Saving Data Objects in Cluster Databases [Page 374], in which the internal table ITAB is stored in the cluster database INDX. Here, the contents are read from the cluster database into the internal table JTAB. The output, consisting of the user fields of INDX and the first five lines of JTAB, looks like this:

```
SY-SUBRC:     0
   AEDAT: 07.12.1995
   USERA: KELLERH
   PGMID: SAPMZTST

JTAB:
          1           1
          2           4
          3           9
          4          16
          5          25
```

Deleting Data Clusters from Cluster Databases

To delete data objects from a cluster database, use the following statement:

Syntax

DELETE FROM DATABASE <dbtab>(<ar>) [CLIENT <cli>] ID <key>.

This statement deletes the entire cluster in area <ar> with the name <key> from the database table <dbtab>. You must declare <dbtab> using a TABLES statement.

The CLIENT <cli> option allows you to disable the automatic client handling of a client-specific cluster database, and specify the client yourself. The addition must always come directly after the name of the database.

To save a data cluster in a database, use the EXPORT TO DATABASE statement (refer to Saving Data Objects in a Cluster Database [Page 374]). For information about the structure of the database table <dbtab>, refer to the section Structure of Cluster Databases [Page 370].

The DELETE statement deletes all of the lines in the cluster database over which the data cluster extends.

If the system is able to delete a data cluster with the specified key, SY-SUBRC is set to 0, otherwise to 4.

```
PROGRAM SAPMZTS4.

TABLES INDX.

DATA DIRTAB LIKE CDIR OCCURS 10.

IMPORT DIRECTORY INTO DIRTAB FROM DATABASE
                      INDX(HK) ID 'Table'.
WRITE: / 'SY-SUBRC, IMPORT:', SY-SUBRC.

DELETE FROM DATABASE INDX(HK) ID 'Table'.
WRITE: / 'SY-SUBRC, DELETE:', SY-SUBRC.

IMPORT DIRECTORY INTO DIRTAB FROM DATABASE
                      INDX(HK) ID 'Table'.
WRITE: / 'SY-SUBRC, IMPORT:', SY-SUBRC.
```

This example deletes the data cluster stored in the cluster database in the example from the section Saving Database Objects in Cluster Databases [Page 374]. If the data cluster exists when you start the program, the display looks like this:

```
SY-SUBRC, IMPORT:      0
SY-SUBRC, DELETE:      0
SY-SUBRC, IMPORT:      4
```

In the first IMPORT statement, the data cluster still exists. The DELETE statement is successful. In the second IMPORT statement, the data cluster no longer exists.

Open SQL Statements and Cluster Databases

Cluster databases are relational databases, defined in the ABAP Dictionary, that have a special use in ABAP. In principle, you can use Open SQL [Page 1041] to access cluster databases.

However, to use SQL statements properly with cluster databases, you must take into account their special structure (refer to Structure of a Cluster Database [Page 370]).

For example, there is little sense in reading the fields CLUSTR and CLUSTID using the SELECT statement, or changing them using UPDATE. These fields contain the coded data clusters, which can only properly be dealt with using EXPORT TO DATABASE and IMPORT FROM DATABASE.

You should only use the Open SQL statements UPDATE, MODIFY, and DELETE if the runtime of the corresponding data cluster statements is too long. Do not use the Open SQL statement INSERT at all in cluster databases.

You can use Open SQL statements to maintain the cluster database. For example, you can use SELECT statements to search the cluster database for particular clusters, using information from the user data fields (see example in the section Storing Data Objects in Cluster Databases [Page 374]). This is not possible with the IMPORT FROM DATABASE statement.

```
PROGRAM SAPMZTS5.

DATA COUNT TYPE I VALUE 0.

TABLES INDX.

SELECT * FROM INDX WHERE RELID = 'HK'
                   AND   SRTF2 = 0
                   AND   USERA = SY-UNAME.

DELETE FROM DATABASE INDX(HK) ID INDX-SRTFD.

IF SY-SUBRC = 0.
    COUNT = COUNT + 1.
  ENDIF.

ENDSELECT.

WRITE: / COUNT, 'Cluster(s) deleted'.
```

This example program deletes all data clusters in the area “HK” from table INDX that have the name of the current user in the field USERA. The SELECT statement fills the field SRTFD in the table work area INDX, which is used in the DELETE statement. By specifying SRTF2 = 0 in the WHERE clause, you ensure that each data cluster is only processed once.

Do not confuse the Open SQL DELETE statement with the DELETE statement used for data clusters (refer to Deleting Data Clusters from Cluster Databases [Page 380]). Always delete **all** of the lines in a data cluster, not just a selection.

The following example demonstrates how you can change the name and area of a data cluster in a database table using the Open SQL statement UPDATE. This task

would have required considerably more effort using the cluster statements EXPORT, IMPORT, and DELETE.

```
PROGRAM SAPMZTS5.

TABLES INDX.

DATA DIRTAB LIKE CDIR OCCURS 10 WITH HEADER LINE.

UPDATE INDX SET RELID = 'NW'
         SRTFD = 'Internal'
                     WHERE RELID = 'HK'
                     AND   SRTFD = 'Table'.

WRITE: / 'UPDATE:',
     / 'SY-SUBRC:', SY-SUBRC,
     / 'SY-DBCNT:', SY-DBCNT.

IMPORT DIRECTORY INTO DIRTAB FROM DATABASE
                          INDX(NW) ID 'Internal'.

WRITE: / 'IMPORT:',
     / 'SY-SUBRC:', SY-SUBRC.
```

This example changes the data cluster stored in the cluster database in the example from the section Saving Database Objects in Cluster Databases [Page 374]. If the data cluster exists when the program is run, and no other errors occur in the UPDATE statement, the output is as follows:

```
UPDATE:
SY-SUBRC:       0
SY-DBCNT:       6
IMPORT:
SY-SUBRC:       0
```

The UPDATE statement [Page 1094] changes the six lines of the database table INDX that belong to the specified data cluster. Afterwards, the IMPORT DIRECTORY statement finds the data cluster in the area “NW” under the name “Internal”.

Working with Files

ABAP allows you to use sequential files located on the application server or presentation server. You can use these files to buffer data, or as an interface between local programs and the R/3 System.

Working with Files on the Application Server [Page 385]

Working with Files on the Presentation Server [Page 416]

The physical addresses of files and file paths are platform-specific The R/3 System contains a function module and a range of transactions that allow you to work with platform-independent filenames.

Using Platform-Independent Filenames [Page 431]

Working with Files on the Application Server

In ABAP, there is a range of statements for processing data that is stored in sequential files on the application server instead of the database. The following sections describe

File Handling in ABAP [Page 386]

Writing Data to Files [Page 405]

Reading Data from Files [Page 407]

During sequential file operations, the system performs a range of automatic checks, some of which may lead to runtime errors.

Automatic Checks in File Operations [Page 409]

File Handling in ABAP

ABAP contains three statements for working with files:

- OPEN DATASET for opening files
- CLOSE DATASET for closing files
- DELETE DATASET for deleting files

Opening a File [Page 387]

Closing a File [Page 403]

Deleting a File [Page 404]

Opening a File

To open a file on the application server, use the OPEN DATASET statement. The basic form of this statement is described in the section

Basic Form of the OPEN DATASET Statement [Page 388].

It also has a range of additions for the following tasks:

ABAP-Specific Additions

Opening a File for Read Access [Page 389]

Opening a File for Write Access [Page 390]

Opening a File for Appending Data [Page 393]

Using Binary Mode [Page 395]

Using Text Mode [Page 397]

Opening a File at a Given Position [Page 399]

Operating System Additions

Executing Operating System Commands [Page 401]

Receiving System Messages [Page 402]

For details of other additions, refer to the keyword documentation in the ABAP Editor for the OPEN DATASET statement.

Basic Form of the OPEN DATASET Statement

To open a file on the application server, use the OPEN statement as follows:

Syntax

OPEN DATASET <dsn> [Additions].

This statement opens the file <dsn>. If you do not specify any additions for the mode, the file is opened in binary mode for reading. SY-SUBRC returns 0 if the system opens the file. Otherwise, SY-SUBRC is set to 8.

You enter the filename <dsn> either as a literal or as a field containing the actual name of the file. If you do not specify a path, the system opens the file in the directory in which the R/3 System is running on the application server. To open a file, the user under which the R/3 System is running must have the requisite authorizations at operating system level.

Filenames are platform-specific. You must therefore use file- and pathnames that conform to the rules of the operating system under which your R/3 System is running. However, you can also use logical filenames to ensure that your programs are not operating system-specific. For further information, refer to Using Platform-Independent Filenames [Page 431].

```
DATA FNAME(60).

FNAME = '/tmp/myfile'.

OPEN DATASET 'myfile'.

OPEN DATASET FNAME.
```

This example works as long as your R/3 System is running under UNIX. The program opens the file “myfile” in the directory in which the R/3 System is running, and also opens the file “myfile” in directory “/tmp”. However, you would have to change the filename for other operating systems. For example, for OpenVMS, you could write the following:

```
FNAME = '[TMP]myfile.BIN'

OPEN DATASET 'myfile.BIN'.
```

Opening a File for Read Access

To open a file for reading, use the FOR INPUT addition to the OPEN DATASET statement.

Syntax

OPEN DATASET <dsn> FOR INPUT.

The file must already exist, otherwise, the system sets SY-SUBRC to 8, and ignores the statement.

If the file exists and is already open (for read or write access, or for appending), the position is reset to the beginning of the file. However, it is good programming style to close files that are already open before you reopen them for a different operation (for further information about closing files, refer to Closing a File [Page 403]).

```
DATA FNAME(60) VALUE 'myfile'.

OPEN DATASET FNAME FOR INPUT.

IF SY-SUBRC = 0.
  WRITE / 'File opened'.
  .....
ELSE.
  WRITE / 'File not found'.
ENDIF.
```

This example opens the file “myfile” for reading.

Opening a File for Write Access

To open a file for writing, use the FOR OUTPUT addition to the OPEN DATASET statement.

Syntax

OPEN DATASET <dsn> FOR OUTPUT.

If the file does not already exist, it is created automatically. If it does already exist, but is closed, its contents are overwritten. If the file exists and is already open (for read or write access, or for appending), the position is reset to the beginning of the file. If the system can open the file <dsn> successfully, SY-SUBRC is set to 0. If not, it is set to 8.

```
DATA: MESS(60),
      FNAME(10) VALUE '/tmp'.

OPEN DATASET FNAME FOR OUTPUT MESSAGE MESS.

IF SY-SUBRC <> 0.
  WRITE: 'SY-SUBRC:', SY-SUBRC,
       / 'System Message:', MESS.
ENDIF.
```

If the R/3 System is ruining under UNIX, the output looks like this:

```
SY-SUBRC:     8
System Message: Is a directory
```

The system cannot open the file, since the name you specified is that of a directory.

The following program shows how the system sets the position when you open a file for writing. However, it is better programming style to close files that are already open before you reopen them for a different operation (for further information about closing files, refer to Closing a File [Page 403]).

```
DATA FNAME(60) VALUE 'myfile'.
DATA NUM TYPE I.

OPEN DATASET FNAME FOR OUTPUT.

DO 10 TIMES.
  NUM = NUM + 1.
  TRANSFER NUM TO FNAME.
ENDDO.

PERFORM INPUT.

OPEN DATASET FNAME FOR OUTPUT.

NUM = 0.
DO 5 TIMES.
  NUM = NUM + 10.
  TRANSFER NUM TO FNAME.
ENDDO.

PERFORM INPUT.
```

```
CLOSE DATASET FNAME.

OPEN DATASET FNAME FOR OUTPUT.

NUM = 0.
DO 5 TIMES.
  NUM = NUM + 20.
  TRANSFER NUM TO FNAME.
ENDDO.

PERFORM INPUT.

FORM INPUT.
  SKIP.
  OPEN DATASET FNAME FOR INPUT.
  DO.
    READ DATASET FNAME INTO NUM.
    IF SY-SUBRC <> 0.
      EXIT.
    ENDIF.
    WRITE / NUM.
  ENDDO.
ENDFORM.
```

The output appears as follows:

```
         1
         2
         3
         4
         5
         6
         7
         8
         9
        10

        10
        20
        30
        40
        50
         6
         7
         8
         9
        10

        20
        40
        60
        80
       100
```

This example performs the following steps using the file “myfile”:

1. It is opened for writing
2. It is filled with 10 integers (for information about the TRANSFER statement, refer to Writing Data to Files [Page 405])
3. It is then opened for reading. The position is reset accordingly to the beginning of the file.

4. It is read into the field NUM. For information about the READ DATASET statement, refer to Reading Data from Files [Page 407]. The values of NUM are displayed on the screen.
5. It is reopened for writing The position is reset to the beginning of the file.
6. It is filled with five integers, which overwrite the previous contents of the file.
7. It is then reopened for reading. The position is reset to the beginning of the file.
8. The file is read into the field NUM. The values of NUM are displayed on the screen.
9. It is closed (for information about the CLOSE DATASET statement, refer to Closing a File [Page 403]).
10. It is then reopened for writing. The system deletes the existing contents of the file.
11. It is filled with 5 integers.
12. It is then opened for reading. The position is reset to the beginning of the file.
13. The file is read into the field NUM. The values of NUM are displayed on the screen.

Opening a File for Appending Data

To open a file so that you can append data to the end of it, use the FOR APPENDING addition in the OPEN DATASET statement:

Syntax

OPEN DATASET <dsn> FOR APPENDING.

This statement opens a file to which you can append data. If the file does not already exist, it is created automatically. If it does exist, but is closed, the system opens it, and sets the position to the end of the file. If the file exists and is already open (for read or write access, or for appending), the position is set to the end of the file. SY-SUBRC is always 0.

It is good programming style to close files that are already open before you reopen them for a different operation (for further information about closing files, refer to Closing a File [Page 403]).

```
DATA FNAME(60) VALUE 'myfile'.

DATA NUM TYPE I.

OPEN DATASET FNAME FOR OUTPUT.
DO 5 TIMES.
  NUM = NUM + 1.
  TRANSFER NUM TO FNAME.
ENDDO.

OPEN DATASET FNAME FOR INPUT.

OPEN DATASET FNAME FOR APPENDING.
NUM = 0.
DO 5 TIMES.
  NUM = NUM + 10.
  TRANSFER NUM TO FNAME.
ENDDO.

OPEN DATASET FNAME FOR INPUT.
DO.
 READ DATASET FNAME INTO NUM.
 IF SY-SUBRC <> 0.
   EXIT.
 ENDIF.
 WRITE / NUM.
ENDDO.
```

The output appears as follows:

```
 1
 2
 3
 4
 5
10
20
```

```
30
40
50
```

This example opens the file “myfile” for write access and fills it with the five integers 1-5 (for further information about the TRANSFER statement, refer to Writing Data to Files [Page 405]). The next OPEN DATASET statement resets the position to the beginning of the file. Then, the file is opened for appending data (the position is set to the end of the file). Five integers between 10 and 50 are written into the file. Finally, the program reads the contents of the file, and displays them on the screen.

Using Binary Mode

To open a file in binary mode, use the IN BINARY MODE addition to the OPEN DATASET statement.

Syntax

OPEN DATASET <dsn> IN BINARY MODE [FOR ....].

If you read from or write to a file that is open in binary mode, the data is transferred byte by byte. The system does not interpret the contents of the file while it is being transferred. If you write the contents of a field to a file, the system transfers all of the bytes in the source field. When you transfer data from a file to a field, the number of bytes transferred depends on the length of the target field. If you then use another ABAP statement to address the target field, the system interprets the field contents according to the data type of the field.

```
DATA FNAME(60) VALUE 'myfile'.

DATA: NUM1     TYPE I,
      NUM2     TYPE I,
      TEXT1(4) TYPE C,
      TEXT2(8) TYPE C,
      HEX      TYPE X.

OPEN DATASET FNAME FOR OUTPUT IN BINARY MODE.

NUM1  = 111.
TEXT1 = 'TEXT'.
TRANSFER NUM1  TO FNAME.
TRANSFER TEXT1 TO FNAME.

OPEN DATASET FNAME FOR INPUT IN BINARY MODE.

READ DATASET FNAME INTO TEXT2.
WRITE / TEXT2.

OPEN DATASET FNAME FOR INPUT IN BINARY MODE.

READ DATASET FNAME INTO NUM2.
WRITE / NUM2.

OPEN DATASET FNAME FOR INPUT IN BINARY MODE.

SKIP.
DO.
  READ DATASET FNAME INTO HEX.
  If SY-SUBRC <> 0.
    EXIT.
  ENDIF.
  WRITE  HEX.
ENDDO.
```

The output appears as follows:

```
###oTEXT

111

00 00 00 6F 54 45 58 54
```

The program opens the file “myfile” in binary mode and writes the contents of the fields NUM1 and TEXT1 into the file. For information about the TRANSFER statement, refer to Writing Data to Files [Page 405]. The file is then opened for reading, and its entire contents are read into the field TEXT2. For information about the READ DATASET statement, refer to Reading Data from Files [Page 407]. The first four characters of the string TEXT2 are nonsense, since the corresponding bytes are the platform-specific representation of the number 111. The system tries to interpret all of the bytes as characters. However, this only works for the last four bytes. After the OPEN statement, the position is reset to the start of the file, and the first four bytes of the file are transferred into NUM2. The value of NUM2 is correct, since it has the same data type as NUM1. Finally, the eight bytes of the file are read into the field HEX. On the screen, you can see the hexadecimal representation of the file contents. The last four bytes are the ASCII representation of the characters in the word “TEXT”.

Using Text Mode

To open a file in text mode, use the IN TEXT MODE addition to the OPEN DATASET statement.

Syntax

OPEN DATASET <dsn> FOR .... IN TEXT MODE.

If you read from or write to a file that is open in text mode, the data is transferred line by line. The system assumes that the file has a line structure.

- In each TRANSFER statement, the system transfers all bytes (apart from spaces at the end) into the file, and places an end of line marker at the end. For information about the TRANSFER statement, refer to Writing Data to Files [Page 405].
- In each READ DATASET statement, the system reads all of the data up to the next end of line marker. For information about the READ DATASET statement, refer to Reading Data from Files [Page 407]. If the target field is too small, the line is truncated. If it is longer than the line in the file, it is filled with trailing spaces.

You should always use text mode if you want to write strings to files or where you know that an existing file has a line construction. You can, for example, use text mode to read files that you have created using any editor on your application server.

The following example works in R/3 Systems that are running on UNIX systems using ASCII.

```
DATA FNAME(60) VALUE 'myfile'.

DATA: TEXT(4),
      HEX TYPE X.

OPEN DATASET FNAME FOR OUTPUT IN TEXT MODE.

TRANSFER '12       ' TO FNAME.
TRANSFER '123456  9 ' TO FNAME.
TRANSFER '1234     ' TO FNAME.

OPEN DATASET FNAME FOR INPUT IN TEXT MODE.

READ DATASET FNAME INTO TEXT.
WRITE / TEXT.
READ DATASET FNAME INTO TEXT.
WRITE  TEXT.
READ DATASET FNAME INTO TEXT.
WRITE  TEXT.

OPEN DATASET FNAME FOR INPUT IN BINARY MODE.

SKIP.
DO.
  READ DATASET FNAME INTO HEX.
  If SY-SUBRC <> 0.
    EXIT.
  ENDIF.
  WRITE HEX.
ENDDO.
```

The output appears as follows:

```
12   1234 1234
31 32 0A 31 32 33 34 35 36 20 20 39 0A 31 32 33 34 0A
```

This example opens a file “myfile” for writing in text mode. Three literals with length 10 characters are written to it. After the file has been opened for reading in text mode, the lines are read into the field TEXT (length 4). The first line is filled with two trailing spaces. The last five characters of the second line are truncated. The structure of the file is displayed by opening it in binary mode and reading its contents into the hexadecimal field HEX. The numbers 31 - 36 are the ASCII codes for the digits 1 - 6. 20 is the code for the space character. The end of each line is marked by 0A. Note that any spaces at the end of strings are not written to the file.

Opening a File at a Given Position

To open a file at a particular position, use the AT POSITION addition in the OPEN DATASET statement.

Syntax

OPEN DATASET <dsn> [FOR ....] [IN ... MODE] AT POSITION <pos>.

This statement opens the file <dsn>, and prepares it for reading or writing from position <pos>. <pos> is the number of bytes from the beginning of the file. It is not possible to specify a position before the beginning of the file.

It makes the most sense to specify a position when you are working in binary mode, since the physical representation of a text file depends on the operating system.

```
DATA FNAME(60) VALUE 'myfile'.

DATA NUM TYPE I.

OPEN DATASET FNAME FOR OUTPUT AT POSITION 16.
DO 5 TIMES.
  NUM = NUM + 1.
  TRANSFER NUM TO FNAME.
ENDDO.

OPEN DATASET FNAME FOR INPUT.
DO 9 TIMES.
  READ DATASET FNAME INTO NUM.
  WRITE / NUM.
ENDDO.

OPEN DATASET FNAME FOR INPUT AT POSITION 28.
SKIP.
DO 2 TIMES.
  READ DATASET FNAME INTO NUM.
  WRITE / NUM.
ENDDO.
```

The output appears as follows:

```
0
0
0
0
1
2
3
4
5

4
5
```

This example opens the file “myfile” in binary mode (this is the default mode). It sets the starting position to 16 for writing to the file, and 28 for reading from it. The example also shows that it only makes sense to specify positions for reading or writing integers if the position is divisible by four.

Executing Operating System Commands

If your system is running under UNIX or WINDOWS NT, you can execute an operating system command in the OPEN DATASET statement using the FILTER addition:

Syntax

OPEN DATASET <dsn> FILTER <filt>.

The operating system command in the field <filt> is processed when the file <dsn> is opened.

The following example works under UNIX:

```
DATA DSN(20) VALUE '/usr/test.Z'.
OPEN DATASET DSN FOR OUTPUT FILTER 'compress'.
.....
OPEN DATASET DSN FOR INPUT FILTER 'uncompress'.
```

The first OPEN statement opens the file ‘/usr/test.Z’ so that the displayed data can be read into it in compressed form.

The second OPEN statement opens the file ‘/usr/test.Z’ so that the data is decompressed when it is read from the file.

Receiving Operating System Messages

To receive the system message sent by the operating system when you try to open a file, use the MESSAGE addition in the OPEN DATASET statement:

Syntax

OPEN DATASET <dsn> MESSAGE <msg>.

This statement imports the message from the operating system and places it in the variable <msg>.

You can use this addition along with the system field SY-SUBRC for error handling.

```
DATA: MESS(60),
      FNAME(10) VALUE 'hugo.xyz'.

OPEN DATASET FNAME MESSAGE MESS.

IF SY-SUBRC <> 0.
  WRITE: 'SY-SUBRC:', SY-SUBRC,
       / 'System Message:', MESS.
ENDIF.
```

If the R/3 System is running under UNIX and the file “hugo.xyz” does not exist, this program would display the following:

```
SY-SUBRC:     8
System Message: No such file or directory
```

Closing a File

To close a file on the application server, use the close statement.

Syntax

CLOSE DATASET <dsn>.

This statement closes the file <dsn>. The naming convention is described in the section Opening a File [Page 387].

You only need to close a file if you want to delete its contents the next time you open it for write access. For further information and an example, refer to Opening a File for Write Access [Page 390].

However, to avoid errors, and to make your programs easier to read, you should always close a file before the next OPEN DATASET statement. The CLOSE statement divides your program into logical blocks, and makes them easier to maintain.

```
DATA FNAME(60) VALUE 'myfile'.

OPEN DATASET FNAME FOR OUTPUT.
 .....
CLOSE FNAME.

OPEN DATASET FNAME FOR INPUT.
 .....
CLOSE FNAME.

OPEN DATASET FNAME FOR INPUT AT POSITION <pos>.
 .....
CLOSE FNAME.
```

The CLOSE statement is not obligatory in this example, but it does improve the layout of the program.

Deleting a File

To delete a file from the application server, use the DELETE DATASET statement:

Syntax

DELETE DATASET <dsn>.

This statement deletes the file <dsn>. The naming convention is described in the section Opening a File [Page 387].

If the system deletes the file <dsn> successfully, SY-SUBRC is set to 0. If not, it is set to 4.

```
DATA FNAME(60) VALUE 'myfile'.

OPEN DATASET FNAME FOR OUTPUT.

OPEN DATASET FNAME FOR INPUT.
IF SY-SUBRC = 0.
  WRITE / 'File found'.
ELSE.
  WRITE / 'File not found'.
ENDIF.

DELETE DATASET FNAME.

OPEN DATASET FNAME FOR INPUT.
IF SY-SUBRC = 0.
  WRITE / 'File found'.
ELSE.
  WRITE / 'File not found'.
ENDIF.
```

The output appears as follows:

```
File found

File not found
```

In this example, the file “myfile” is opened for write access as long as it does not already exist. The system finds the file when it is opened for read access. After the DELETE DATASET statement, the system can no longer find the file.

Writing Data to Files

To write data to a file on the application server, use the TRANSFER statement:

Syntax

TRANSFER <f> to <dsn> [LENGTH <len>].

This statement writes the values of the field <f> into the file <dsn>. You can specify the transfer mode in the OPEN DATASET statement. If you have not already opened the file for writing, the system tries to open it either in binary mode, or using the additions from the last OPEN DATASET statement. However, it is good practice only to open files using the OPEN DATASET statement. For further information about the OPEN DATASET statement and the naming conventions for files, refer to Opening a File [Page 387]. <f> can have an elementary data type, but may also be a structure, as long as it does not contain an internal table. You cannot write internal tables into files in a single step.

You can specify the length of the data you want to transfer using the LENGTH addition. The system then transfers the first <len> bytes into the file. If <len> is too short, excess bytes are truncated. If <len> is greater than the length of the field, the system adds trailing blanks.

The following program shows how you can write internal tables into a file:

```
DATA FNAME(60) VALUE 'myfile'.

TYPES: BEGIN OF LINE,
         COL1 TYPE I,
         COL2 TYPE I,
       END OF LINE.

TYPES  ITAB TYPE LINE OCCURS 10.

DATA: LIN TYPE LINE,
      TAB TYPE ITAB.

DO 5 TIMES.
  LIN-COL1 = SY-INDEX.
  LIN-COL2 = SY-INDEX ** 2.
  APPEND LIN TO TAB.
ENDDO.

OPEN DATASET FNAME FOR OUTPUT.
LOOP AT TAB INTO LIN.
  TRANSFER LIN TO FNAME.
ENDLOOP.
CLOSE DATASET FNAME.

OPEN DATASET FNAME FOR INPUT.
DO.
  READ DATASET FNAME INTO LIN.
  IF SY-SUBRC <> 0.
    EXIT.
  ENDIF.
  WRITE: / LIN-COL1, LIN-COL2.
ENDDO.
```

```
CLOSE DATASET FNAME.
```

The output is:

```
1            1
2            4
3            9
4           16
5           25
```

In this example, a structure LIN and an internal table TAB with line type LINE are created. Once the internal table TAB has been filled, it is written line by line into the file “myfile”. The file is then read into the structure LIN, whose contents are displayed on the screen.

The following example works in R/3 Systems that are running on UNIX systems using ASCII.

```
DATA FNAME(60) VALUE 'myfile'.

DATA: TEXT1(4) VALUE '1234',
      TEXT2(8) VALUE '12345678',
      HEX  TYPE X.

OPEN DATASET FNAME FOR OUTPUT IN TEXT MODE.
TRANSFER: TEXT1 TO FNAME LENGTH 6,
          TEXT2 TO FNAME LENGTH 6.
CLOSE DATASET FNAME.

OPEN DATASET FNAME FOR INPUT.
DO.
  READ DATASET FNAME INTO HEX.
  IF SY-SUBRC <> 0.
    EXIT.
  ENDIF.
  WRITE  HEX.
ENDDO.
CLOSE DATASET FNAME.
```

The output is:

```
31 32 33 34 20 20 0A 31 32 33 34 35 36 0A
```

This example writes the strings TEXT1 and TEXT2 to the file “myfile” in text mode. The output length is set to 6. By reading the file byte by byte into the hexadecimal field, you can see how it is stored. The numbers 31 to 36 are the ASCII codes for the digits 1 to 6. 20 is a space, and 0A marks the end of the line. TEXT1 is filled with two trailing spaces. Two characters are truncated from TEXT2.

Reading Data from Files

To read data from a file on the application server, use the READ DATASET statement:

Syntax

READ DATASET <dsn> INTO <f> [LENGTH <len>].

This statement reads data from the file <dsn> into the variable <f>. In order to determine into which variable you should read data from a file, you need to know the structure of the file.

You can specify the transfer mode in the OPEN DATASET statement. If you have not already opened the file for reading, the system tries to open it either in binary mode, or using the additions from the last OPEN DATASET statement. However, it is good practice only to open files using the OPEN DATASET statement. For further information about the OPEN DATASET statement and the naming conventions for files, refer to Opening a File [Page 387].

If the system was able to read data successfully, SY-SUBRC is set to 0. When the end of the file is reached, SY-SUBRC is set to 4. If the file could not be opened, SY-SUBRC is set to 8.

If you are working in binary mode, you can use the LENGTH addition to find out the length of the data transferred to <f>. The system sets the value of the variable <len> to this length.

```
DATA FNAME(60) VALUE 'myfile'.

DATA: TEXT1(12) VALUE 'abcdefghijkl',
      TEXT2(5),
      LENG TYPE I.

OPEN DATASET FNAME FOR OUTPUT IN BINARY MODE.
TRANSFER TEXT1 TO FNAME.
CLOSE DATASET FNAME.

OPEN DATASET FNAME FOR INPUT IN BINARY MODE.
DO.
  READ DATASET FNAME INTO TEXT2 LENGTH LENG.
  WRITE: / SY-SUBRC, TEXT2, LENG.
  IF SY-SUBRC <> 0.
    EXIT.
  ENDIF.
ENDDO.
CLOSE DATASET FNAME.
```

The output is:

```
0 abcde          5
0 fghij          5
4 kl###          2
```

This example fills the file “myfile” with 12 bytes from the field TEXT1. It is then read into the field TEXT2 in 5-byte portions. Note here that the system fills up the last three bytes of TEXT2 with zeros after the end of the file has been reached. The number of bytes transferred is contained in the field LENG.

If you are working in text mode, you can use the LENGTH addition to find out the length of the current line in the file. The system sets the value of the variable <len> to the length of the line.

The system calculates this by counting the number of bytes between the current position and the next end of line marker in the file.

```
DATA FNAME(60) VALUE 'myfile'.

DATA: TEXT1(4) VALUE '1234    ',
      TEXT2(8) VALUE '12345678',
      TEXT3(2),
      LENG TYPE I.

OPEN DATASET FNAME FOR OUTPUT IN TEXT MODE.
  TRANSFER: TEXT1 TO FNAME,
            TEXT2 TO FNAME.
CLOSE DATASET FNAME.

OPEN DATASET FNAME FOR INPUT IN TEXT MODE.
  DO 2 TIMES.
    READ DATASET FNAME INTO TEXT3 LENGTH LENG.
    WRITE: / TEXT3, LENG.
  ENDDO.
CLOSE DATASET FNAME.
```

The output appears as follows:

```
12              4

12              8
```

This example writes the strings TEXT1 and TEXT2 to the file “myfile” in text mode. They are then read into the string TEXT3 (length 2). The amount of memory occupied by the lines is read into the field LENG.

Automatic Checks in File Operations

The R/3 System automatically performs the following checks in operations with sequential files:

It checks the authorization object S_DATASET to see whether the current program may access the specified file.

(See also Authorization Checks for Programs and Files [Page 410].)

It checks in table SPTH to see whether the specified file is registered for file access from ABAP. Table SPTH also allows you to check the user's authorization.

(see also General Checks for File Access [Page 413].)

Authorization Checks for Programs and Files

BC - Benutzer und Rollen [Ext.]

When you access sequential files on the application server using the following statements:

- OPEN DATASET
- READ DATASET
- TRANSFER
- DELETE DATASET

the system automatically checks the user's authorization against the authorization object S_DATASET.

This object allows you to assign authorization for **particular** files from **particular** programs. You can also assign the authorization to use operating system commands as a file filter.

Do not use S_DATASET to control general access rights to files from ABAP, or user-dependent authorization checks. Instead, use table SPTH (see also General Checks for Accessing Files [Page 413]).

The Authorization Object S_DATASET

The object S_DATASET consists of the following fields:

- ABAP program name

  Name of the ABAP program from which access is allowed. This allows you to restrict file access to a few programs specifically for that task.

- Activity

  The possible values are:

  33: Read file normally

  34: Write to or delete file normally

  A6: Read file with filter (operating system command)

  A7: Write to file with filter (operating system command)

- File name

  Name of the operating system file. This allows you to restrict the files to which the user has access.

For more information about authorization objects, refer to the Users and Authorizations documentation.

If the result of the automatic authorization check is negative, a runtime error occurs. You should therefore check the authorization in your ABAP program **before** accessing the file using the function module AUTHORITY_CHECK_DATASET.

The Function Module AUTHORITY_CHECK_DATASET

This function module allows you to check whether the user is authorized to access a file before the system tries to open it. This preempts a possible runtime error that can otherwise occur in the automatic authorization check.

The function module has the following import parameters:

- PROGRAM

  Name of the ABAP program from which the file is to be opened. If you do not specify a program name, the system assumes the current program.

- ACTIVITY

  Access type, with the following possible values:

  – READ: Read file

  – WRITE: Change file

  – READ_WITH_FILTER: Read file using filter functions

  – WRITE_WITH_FILTER: Change file using filter functions

  – DELETE: Delete file

  These values are defined as constants in the type group SABC as follows:

```
TYPE-POOL SABC .

CONSTANTS:
  SABC_ACT_READ(4)                VALUE 'READ',
  SABC_ACT_WRITE(5)               VALUE 'WRITE',
  SABC_ACT_READ_WITH_FILTER(16)   VALUE 'READ_WITH_FILTER',
  SABC_ACT_WRITE_WITH_FILTER(17) VALUE 'WRITE_WITH_FILTER',
  SABC_ACT_DELETE(6)              VALUE 'DELETE',
  SABC_ACT_INIT(4)                VALUE 'INIT',
  SABC_ACT_ACCEPT(6)              VALUE 'ACCEPT',
  SABC_ACT_CALL(4)                VALUE 'CALL'.
```

- FILENAME

  Name of the file that you want to access.

```
TYPE-POOLS SABC.

.....

CALL FUNCTION 'AUTHORITY_CHECK_DATASET'
   EXPORTING  PROGRAM        = SY-REPID
              ACTIVITY       = SABC_ACT_READ
              FILENAME       = '/tmp/sapv01'
   EXCEPTIONS NO_AUTHORITY    = 1
              ACTIVITY_UNKNOWN = 2.

......
```

  This function module call finds out whether the current program may access the file ‘/tmp/sapv01’.

General Checks for File Access

When you access sequential files on the application server using the following statements

- OPEN DATASET
- TRANSFER
- DELETE DATASET

the system automatically checks against table SPTH. This table regulates general read and write access from ABAP to files, and whether files should be included in security procedures.

In table SPTH, you can prevent read or write access to **generically-specified** files, **independently** of the R/3 authorization concept. For all other files (that is, those for which read and write access is allowed according to table SPTH), you **can** use the R/3 authorization concept to check authorizations. To enable you to do this, you can specify authorization groups in table SPTH for program-independent user authorization checks.

SPTH contains the following columns for this purpose:

- PATH

  This column contains generic filenames. This means that the files on the application server to which an entry in this column applies, retain the attributes specified in the remaining columns of the line.

  Suppose SPTH contains the following three entries in the column PATH:

  *

  /tmp

  /tmp/myfile

  The entries are then valid as follows:

  – First line: All files on the application server apart from the path ‘/tmp’
  – Second line: All files on the application server in the path ‘/tmp’ apart from the file ‘/tmp/myfile’
  – Third line: The application server file ‘/tmp/myfile’

- SAVEFLAG

  This column is a flag that you set using ‘X’.

  If the flag is set, the files specified in the PATH column are included in security procedures.

- FS_NOREAD

  This column is a flag that you set using ‘X’.

  If the flag is set, the files specified in the PATH column may not be accessed **at all** from ABAP. This flag overrides all user authorizations. If you set FS_NOREAD, FS_NOWRITE is also automatically set.

If the flag is not set, it is possible to access the files from ABAP if the authorization checks are successful (see also the FSBRGRU column and Authorization Check for Particular Programs and Files [Page 410]).

- FS_NOWRITE

  This column is a flag that you set using 'X'.

  If the flag is set, the files specified in the PATH column may not be accessed **for changing** from ABAP. This flag overrides all user authorizations.

  If the flag is not set, it is possible to change the files from ABAP if the authorization checks are successful (see also the FSBRGRU column and Authorization Check for Particular Programs and Files [Page 410]).

- FSBRGRU

  This column contains the names of authorization groups.

  An authorization group corresponds to the first field (RS_BRGRU) of the authorization object S_PATH. You can use the second field of the authorization object S_PATH (ACTVT) to check whether the user has authorization to read (value 3) or change (value 2) the files in he authorization group.

  Entries in FSBRGRU specify groups of files on the application server. You can control the access to files by assigning authorizations for the authorization object S_PATH.

  Unlike authorization checks using the authorization object S_DATASET (see Authorization Checks for Particular Programs and Files [Page 410]), the authorization check against the authorization object S_PATH is independent of the ABAP program used to access the files. Furthermore, the check is not restricted to individual files. Instead, it extends to all of the generically-specified files in the PATH column.

If there is no entry in the column FSBRGRU, the files in the column PATH are not assigned to an authorization group, and there is no authorization check against the authorization object S_PATH.

If the automatic check for a file access fails, a runtime error occurs.

Suppose the table SPTH contains the following entries:

PATH	S A V EF LA G	FS_ NO RE AD	FS_N OWR ITE	F S E F C F U
*		X	X	
/tmp				

/tmp/files	X	F I L E

With these settings, ABAP programs cannot access any files on the application server apart from those in the path '(/tmp').

All ABAP programs can read from and write to the files in that path.

Only users with authorizations for the authorization group FILE can use ABAP program that read from or write to files in the path '/tmp/files'. These files are also included in the security procedure.

With the above table entries, the following program extract would cause a runtime error for any user:

```
DATA: FNAME(60).
FNAME = '/system/files'.
OPEN DATASET FNAME FOR OUTPUT.
```

Working with Files on the Presentation Server

To work with files on the presentation server, you use special function modules. Internal tables serve as an interface between your programs and the function modules. You can use these function modules to:

Write Data to the Presentation Server with a User Dialog [Page 417]

Write Data to the Presentation Server without a User Dialog [Page 420]

Read Data from the Presentation Server with a User Dialog [Page 423]

Read Data from the Presentation Server without a User Dialog [Page 426]

Check Files on the Presentation Server [Page 428]

The physical names of the files depend on the operating system of the presentation server. You can, however, make your ABAP programs platform-independent by using logical filenames. For further information, refer to Using Platform-Independent Filenames [Page 431].

Writing Data to Presentation Server (Dialog)

To write data from an internal table to the presentation server using a user dialog, use the function module DOWNLOAD. The most important parameters are listed below. For more information, refer to the function module documentation in the Function Builder (Transaction SE37).

Important Import Parameters

Parameter	Function
BIN_FILESIZE	File length for binary files
CODEPAGE	Only for download under DOS: Value IBM
FILENAME	Filename (default value for user dialog)
FILETYPE	File type (default value for user dialog)
ITEM	Title for dialog box
MODE	Write mode (blank = overwrite, 'A' = append)

Use the FILETYPE parameter to specify the transfer mode. Possible values:

- BIN

  Binary files: You must specify the file length. The internal table must consist of a single column with data type X.

- ASC

  ASCII files:

- DAT

  Excel files: The columns are separated using tabs. The lines are separated with line breaks.

- WK1

  Excel and Lotus files: The files are saved in a WK1 spreadsheet.

Important Export Parameters

Parameter	Function
ACT_FILENAME	File name (as entered in the user dialog)
ACT_FILETYPE	File type (as entered in the user dialog)
FILESIZE	Number of bytes transferred

Tables Parameters

Parameter	Function
DATA_TAB	Internal table containing data

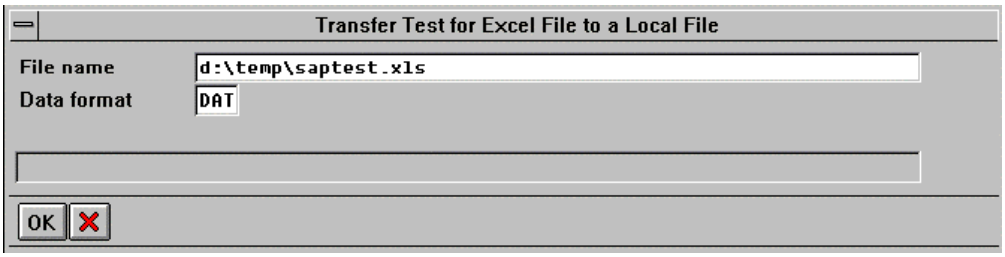
Exceptions Parameters

Parameter	Function
INVALID_FILESIZE	Invalid parameter BIN_FILESIZE
INVALID_TABLE_WIDTH	Invalid table structure
INVALID_TYPE	Value of FILETYPE parameter is incorrect

Suppose the presentation server is running under Windows NT, and you have written the following program:

```
PROGRAM SAPMZTST.

DATA: FNAME(128), FTYPE(3), FSIZE TYPE I.

TYPES: BEGIN OF LINE,
         COL1 TYPE I,
         COL2 TYPE I,
       END OF LINE.

TYPES  ITAB TYPE LINE OCCURS 10.

DATA: LIN TYPE LINE,
      TAB TYPE ITAB.

DO 5 TIMES.
  LIN-COL1 = SY-INDEX.
  LIN-COL2 = SY-INDEX ** 2.
  APPEND LIN TO TAB.
ENDDO.

CALL FUNCTION 'DOWNLOAD'

    EXPORTING
      CODEPAGE            = 'IBM'
      FILENAME            = 'd:\temp\saptest.xls'
      FILETYPE            = 'DAT'
      ITEM                = 'Test for Excel File'

    IMPORTING
      ACT_FILENAME        = FNAME
      ACT_FILETYPE        = FTYPE
      FILESIZE            = FSIZE

    TABLES
      DATA_TAB            = TAB

    EXCEPTIONS
      INVALID_FILESIZE    = 1
      INVALID_TABLE_WIDTH = 2
      INVALID_TYPE        = 3.

WRITE: 'SY-SUBRC:', SY-SUBRC,
    / 'Name    :', (60) FNAME,
    / 'Type    :', FTYPE,
    / 'Size    :', FSIZE.
```

The program displays the following dialog box:



Here, the user can change the default values. When the user chooses *Transfer*, the system writes the data from the internal table TAB, filled in the program, into the file D:\temp\saptest.xls. If the file already exists, the system asks the user whether it should replace the existing version. The system inserts tabs between the columns, and line breaks at the end of each line.

The output appears as follows:

```
SY-SUBRC:     0
Name    : d:\temp\saptest.xls
Type    : DAT
Size    :         27
```

You can now open the file D:\temp\saptest.xls using MS Excel on the presentation server. It looks like this:

	A	B	C
1	1	1	
2	2	4	
3	3	9	
4	4	16	
5	5	25	
6			

Writing Data to Presentation Server (no Dialog)

To write data from an internal table to the presentation server without using a user dialog, use the function module WS_DOWNLOAD. The most important parameters are listed below. For more information, refer to the function module documentation in the Function Builder (Transaction SE37).

Important Import Parameters

Parameter	Function
BIN_FILESIZE	File length for binary files
CODEPAGE	Only for download under DOS: Value IBM
FILENAME	Filename
FILETYPE	File type
MODE	Write mode (blank = overwrite, 'A' = append)

Use the FILETYPE parameter to specify the transfer mode. Possible values:

- BIN

  Binary files: You must specify the file length. The internal table must consist of a single column with data type X.

- ASC

  ASCII files:

- DAT

  Excel files: The columns are separated using tabs. The lines are separated with line breaks.

- WK1

  Excel and Lotus files: The files are saved in a WK1 spreadsheet.

Export Parameter

Parameter	Function
FILELENGTH	Number of bytes transferred

Tables Parameters

Parameter	Function
DATA_TAB	Internal table containing data

Exceptions Parameters

Parameter	Function
FILE_OPEN_ERROR	Unable to open the file
FILE_WRITE_ERROR	Unable to write to file
INVALID_FILESIZE	Invalid parameter BIN_FILESIZE
INVALID_TABLE_WIDTH	Invalid table structure
INVALID_TYPE	Value of FILETYPE parameter is incorrect

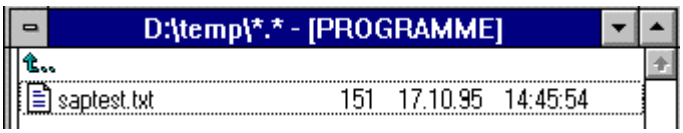
Suppose the presentation server is running under Windows NT, and you have written the following program:

```
PROGRAM SAPMZTST.

DATA: FLENGTH TYPE I.

DATA TAB(80) OCCURS 5.

APPEND 'This is the first line of my text. ' TO TAB.
APPEND 'The second line.                   ' TO TAB.
APPEND '      The third line.              ' TO TAB.
APPEND  '         The fourth line.        ' TO TAB.
APPEND  '           Fifth and final line. ' TO TAB.

CALL FUNCTION 'WS_DOWNLOAD'

    EXPORTING
      CODEPAGE           = 'IBM'
      FILENAME           = 'd:\temp\saptest.txt'
      FILETYPE           = 'ASC'

    IMPORTING
      FILELENGTH         = FLENGTH

    TABLES
      DATA_TAB           = TAB

    EXCEPTIONS
      FILE_OPEN_ERROR    = 1
      FILE_WRITE_ERROR   = 2
      INVALID_FILESIZE   = 3
      INVALID_TABLE_WIDTH = 4
      INVALID_TYPE       = 5.

WRITE: 'SY-SUBRC   :', SY-SUBRC,
    / 'File length:', FLENGTH.
```

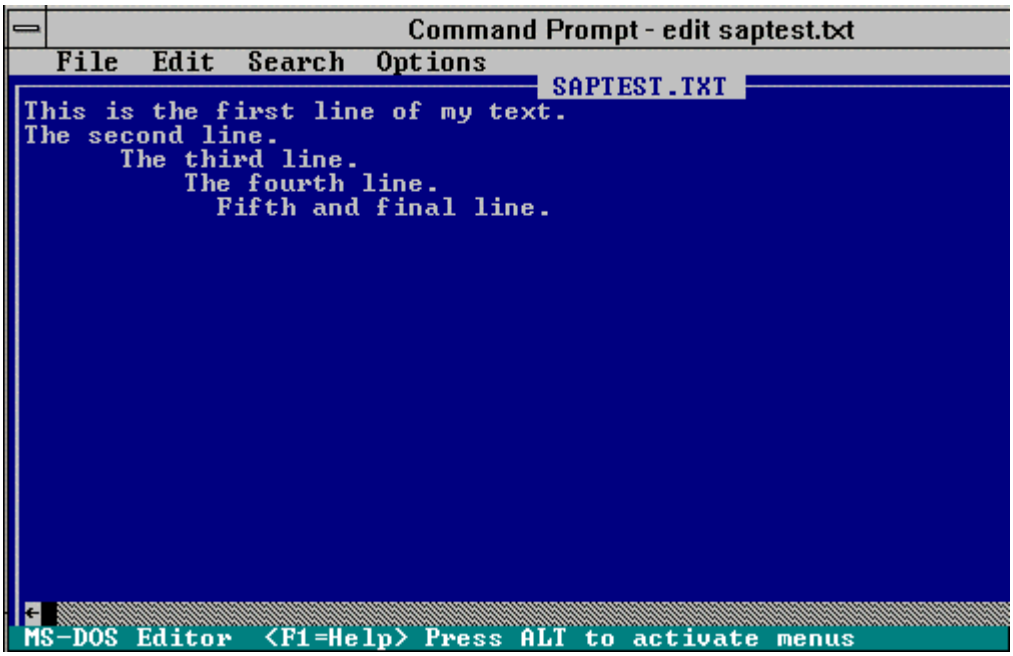
The output appears as follows:

```
SY-SUBRC    :       0
File length:         151
```

The system has written the five lines of the table TAB into the ASCII file D:\temp\saptest.txt. You can now use the Windows Explorer to check that the file exists:



You can now open the file D:\temp\saptest.xls using any editor on the presentation server. Here, we have used the DOS Editor.


Command Prompt - edit saptest.txt
File Edit Search Options
SAPTEST.TXT
This is the first line of my text.
The second line.
The third line.
The fourth line.
Fifth and final line.
MS-DOS Editor <F1=Help> Press ALT to activate menus

Reading Data from Presentation Server (Dialog)

To read data from the presentation server into an internal table with a user dialog, use the function module UPLOAD. The most important parameters are listed below. For more information, refer to the function module documentation in the Function Builder (Transaction SE37).

Important Import Parameters

Parameters	Function
CODEPAGE	Only for upload under DOS: Value IBM
FILENAME	Filename (default value for user dialog)
FILETYPE	File type (default value for user dialog)
ITEM	Title for dialog box

Use the FILETYPE parameter to specify the transfer mode. Possible values:

- BIN

  Binary files

- ASC

  ASCII files: Text files with end of line markers.

- DAT

  Excel files, saved as text files with columns separated by tabs and lines separated by line breaks.

- WK1

  Excel and Lotus files saved as WK1 spreadsheets.

Important Export Parameters

Parameters	Function
FILESIZE	Number of bytes transferred
ACT_FILENAME	Filename (as entered in the user dialog)
ACT_FILETYPE	File type (as entered in the user dialog)

Tables Parameters

Parameters	Function
DATA_TAB	Internal table (target for the import)

Exceptions Parameters

Parameters	Function
CONVERSION_ERROR	Error converting data
INVALID_TABLE_WIDTH	Invalid table structure
INVALID_TYPE	Incorrect FILETYPE parameter

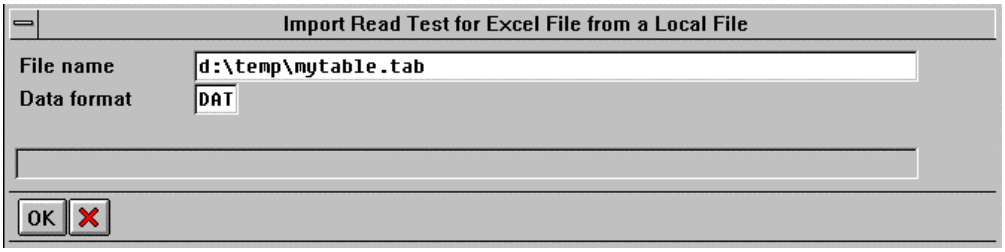
Suppose the presentation server is running under Windows NT, and contains the following Excel file:

	A	B	C
1	Billy	the	Kid
2	My	Fair	Lady
3	Herman	the	German
4	Conan	the	Barbarian

If this table is saved as a text file “D:\temp\mytable.txt” with tabs between the columns, the following program can read the table:

```
PROGRAM SAPMZTST.

DATA: FNAME(128), FTYPE(3), FSIZE TYPE I.

TYPES: BEGIN OF LINE,
         COL1(10) TYPE C,
         COL2(10) TYPE C,
         COL3(10) TYPE C,
       END OF LINE.

TYPES  ITAB TYPE LINE OCCURS 10.

DATA: LIN TYPE LINE,
      TAB TYPE ITAB.

CALL FUNCTION 'UPLOAD'

    EXPORTING
        CODEPAGE          = 'IBM'
        FILENAME          = 'd:\temp\mytable.txt'
        FILETYPE          = 'DAT'
        ITEM              = 'Read Test for Excel File'

    IMPORTING
        FILESIZE          =  FSIZE
        ACT_FILENAME      =  FNAME
        ACT_FILETYPE      =  FTYPE

    TABLES
        DATA_TAB          =  TAB

    EXCEPTIONS
        CONVERSION_ERROR    = 1
        INVALID_TABLE_WIDTH = 2
        INVALID_TYPE        = 3.

WRITE: 'SY-SUBRC:', SY-SUBRC,
    / 'Name    :', (60) FNAME,
    / 'Type    :', FTYPE,
    / 'Size    :', FSIZE.

SKIP.
LOOP AT TAB INTO LIN.
  WRITE: / LIN-COL1, LIN-COL2, LIN-COL3.
ENDLOOP.
```

The program displays the following dialog box:



Here, the user can change the default values. When the user chooses Transfer, the system imports the data from the file D:\temp\mytable.txt into the internal table TAB.

The output appears as follows:

```
SY-SUBRC:     0
Name    : d:\temp\mytable.txt
Type    : DAT
Size    :          69

Billy       the         Kid
My          Fair        Lady
Herman      the         German
Conan       the         Barbarian
```

The contents of the internal table TAB are exactly the same as the contents of the original Excel table.

Reading Data from Presentation Server (no Dialog)

To read data from the presentation server into an internal table without a user dialog, use the function module WS_UPLOAD. The most important parameters are listed below. For more information, refer to the function module documentation in the Function Builder (Transaction SE37).

Important Export Parameters

Parameters	Function
CODEPAGE	Only for upload under DOS: Value IBM
FILENAME	Filename
FILETYPE	File type

Use the FILETYPE parameter to specify the transfer mode. Possible values:

- BIN

  Binary files

- ASC

  ASCII files: Text files with end of line markers.

- DAT

  Excel files, saved as text files with columns separated by tabs and lines separated by line breaks.

- WK1

  Excel and Lotus files saved as WK1 spreadsheets.

Export Parameters

Parameters	Function
FILELENGTH	Number of bytes transferred

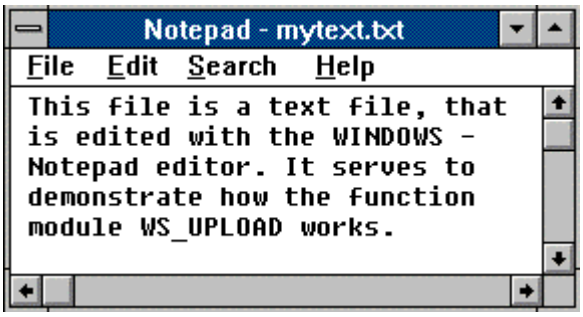
Tables Parameters

Parameters	Function
DATA_TAB	Internal table (target for the import)

Exception Parameters

Parameters	Function
CONVERSION_ERROR	Error converting data
FILE_OPEN_ERROR	Unable to open the file
FILE_READ_ERROR	Unable to read the file
INVALID_TABLE_WIDTH	Invalid table structure
INVALID_TYPE	Value of FILETYPE parameter is incorrect

Suppose the presentation server is running under Windows NT, and contains the following text file:



The following program reads the text file:

```
PROGRAM SAPMZTST.

DATA: FLENGTH TYPE I.

DATA: TAB(80) OCCURS 5 WITH HEADER LINE.

CALL FUNCTION 'WS_UPLOAD'

    EXPORTING
       CODEPAGE          = 'IBM'
       FILENAME          = 'd:\temp\mytext.txt'
       FILETYPE          = 'ASC'

    IMPORTING
       FILELENGTH        =  FLENGTH

    TABLES
       DATA_TAB          =  TAB

    EXCEPTIONS
       CONVERSION_ERROR    = 1
       FILE_OPEN_ERROR     = 2
       FILE_READ_ERROR     = 3
       INVALID_TABLE_WIDTH = 4
       INVALID_TYPE        = 5.

WRITE: 'SY-SUBRC:', SY-SUBRC,
    / 'Length  :', FLENGTH.

SKIP.
LOOP AT TAB.
  WRITE: / TAB.
ENDLOOP.
```

The output appears as follows:

```
SY-SUBRC:     0
Length  :       145

This file is a text file, that
is edited with the WINDOWS -
Notepad editor. It serves to
demonstrate how the function
module WS_UPLOAD works.
```

Checking Files on the Presentation Server

To retrieve information about files on the presentation server and the presentation server operating system, use the function module WS_QUERY. The most important parameters are listed below. For more information, refer to the function module documentation in the Function Builder (Transaction SE37).

Important Import Parameters

Parameter	Function
FILENAME	Filename for query commands 'FE', 'FL', and 'DE'
QUERY	Query command

The import parameter QUERY defines the query. Important query commands:

- CD: Returns the current directory
- EN: Returns environment variables
- FL: Returns the length of the file specified in FILENAME
- FE: Returns whether the file specified in FILENAME exists
- DE: Returns whether the directory specified in FILENAME exists
- WS: Returns the windowing system of the presentation server
- OS: Returns the operating system of the presentation server

Export Parameter

Parameter	Function
RETURN	Result of the query ('0' = no, '1' = yes)

Exception Parameter

Parameter	Function
INV_QUERY	QUERY or FILENAME contains an incorrect value

Suppose the presentation server is running under Windows NT, and the file SYSTEM.INI exists as shown below:

The following program returns some of the attributes of the operating system and of the file:

```
PROGRAM SAPMZTST.

DATA:  FNAME(60), RESULT(30), FLENGTH TYPE I.

FNAME = 'C:\WINNT35\SYSTEM.INI'.
```

```
CALL FUNCTION 'WS_QUERY'
   EXPORTING
      QUERY      = 'OS'
   IMPORTING
      RETURN     = RESULT
   EXCEPTIONS
      INV_QUERY   = 1.

IF SY-SUBRC = 0.
  WRITE: / 'Operating System:', RESULT.
ENDIF.

CALL FUNCTION 'WS_QUERY'
   EXPORTING
      QUERY      = 'WS'
   IMPORTING
      RETURN     = RESULT
   EXCEPTIONS
      INV_QUERY   = 1.

IF SY-SUBRC = 0.
  WRITE: / 'Windows:', RESULT.
ENDIF.

CALL FUNCTION 'WS_QUERY'
   EXPORTING
      FILENAME    = FNAME
      QUERY      = 'FE'
   IMPORTING
      RETURN     = RESULT
   EXCEPTIONS
      INV_QUERY   = 1.

IF SY-SUBRC = 0.
  WRITE: / 'File exists ?', RESULT.
ENDIF.

CALL FUNCTION 'WS_QUERY'
   EXPORTING
      FILENAME    = FNAME
      QUERY      = 'FL'
   IMPORTING
      RETURN     = FLENGTH
   EXCEPTIONS
      INV_QUERY   = 1.

IF SY-SUBRC = 0.
  WRITE: / 'File Length:', FLENGTH.
ENDIF.
```

Output:

```
Operating System: NT

Windows: WN32

File exists ? 1

File Length:        210
```

The windowing system WN32 is the windows system of WINDOWS NT. For information about the abbreviations used, place the cursor on the QUERY field in the documentation screen and choose *Help*.

Using Platform-Independent Filenames

The file names that you use in ABAP statements for processing files are physical names. This means that they must be syntactically correct filenames for the operating system under which your R/3 System is running. Once you have created a file from an ABAP program with a particular name and path, you can find the same file using the same name and path at operating system level.

Since the naming conventions for files and paths differ from operating system to operating system, ABAP programs are only portable from one operating system to another if you use the tools described below.

To make programs portable, the R/3 System has a concept of logical filenames and paths. These are linked to physical files and paths. The links are created in special tables, which you can maintain according to your own requirements. In an ABAP program, you can then use the function module FILE_GET_NAME to generate a physical filename from a logical one.

Maintaining platform-independent filenames is part of Customizing. For a full description, choose *Tools* → *Business Engineer* → *Customizing*, followed by *Implement. projects* → *SAP Reference IMG*. On the next screen, choose *Basis Components* → *System Administration* → *Platform-independent File Names.*

For a more detailed description of the function module FILE_GET_NAME, enter its name on the initial screen of the Function Builder and choose *Goto* → *Documentation*. On the next screen, choose *Function module doc.*

Another way of maintaining platform-independent filenames is to use the Transaction FILE. The following sections provide an overview of the transaction.

Once you have started the transaction, you can use the options in the *Navigation* group box to maintain the above tables as described in the following sections:

Maintaining Syntax Groups [Page 432]

Assigning Operating Systems to Syntax Groups [Page 433]

Creating and Defining Logical Paths [Page 435]

Creating and Defining Logical Filenames [Page 437]

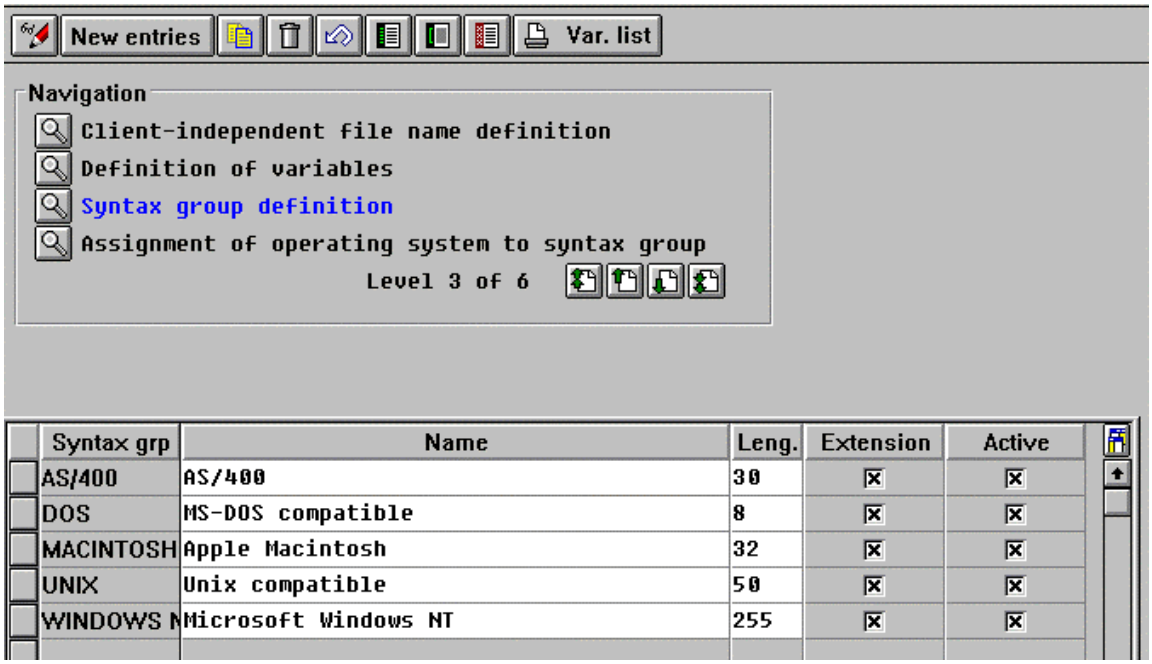
There is also an explanation of how to use the function module FILE_GET_NAME to convert logical filenames in ABAP programs into physical filenames.

Using Logical Files in ABAP Programs [Page 438]

For further information about platform-dependent filenames, refer to the Platform Independent Filename Assignment [Ext.] section of the Extended Applications Function Library documentation.

Maintaining Syntax Groups

A syntax group contains the names of all operating systems that use the same syntax to assign filenames. To maintain a syntax group, choose *Syntax group definition* from the *Navigation* group box in Transaction FILE. The following screen appears:

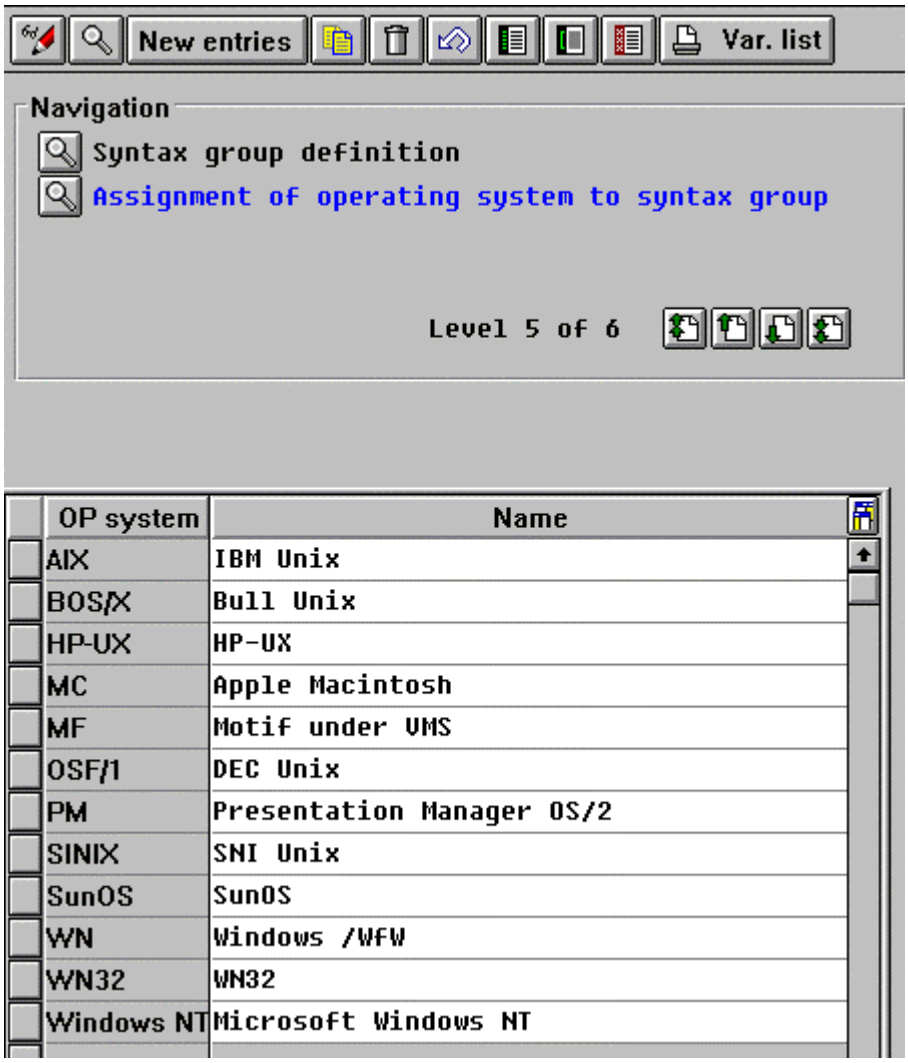


Syntax grp	Name	Leng.	Extension	Active
AS/400	AS/400	30	☒	☒
DOS	MS-DOS compatible	8	☒	☒
MACINTOSH	Apple Macintosh	32	☒	☒
UNIX	Unix compatible	50	☒	☒
WINDOWS N	Microsoft Windows NT	255	☒	☒

The screen contains a list of existing syntax groups. To create a new group, choose *New entries*. For further information, refer to Assigning Operating Systems to Syntax Groups [Page 433].

Assigning Operating Systems to Syntax Groups

In the *Navigation* group box of Transaction FILE, choose *Assignment of operating system to syntax group*. The following screen appears:

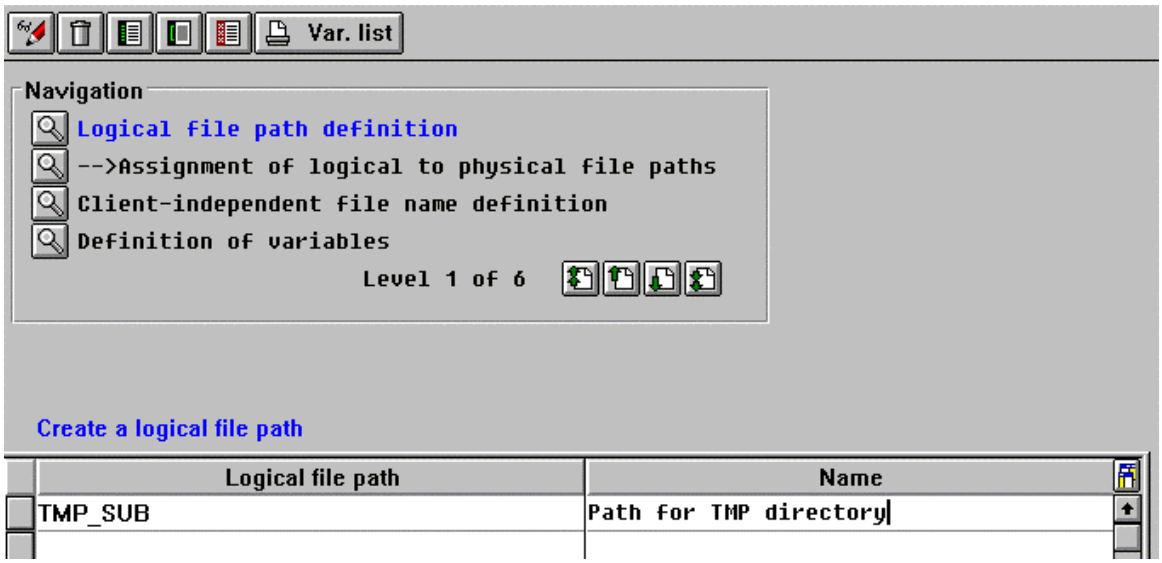


This is a list of the operating systems currently supported. To add a new entry, choose *New entries*. To assign an existing entry to a syntax group, select an operating system and choose *Detail*. The following screen appears:

The operating system HP-UX is now assigned to the syntax group UNIX.

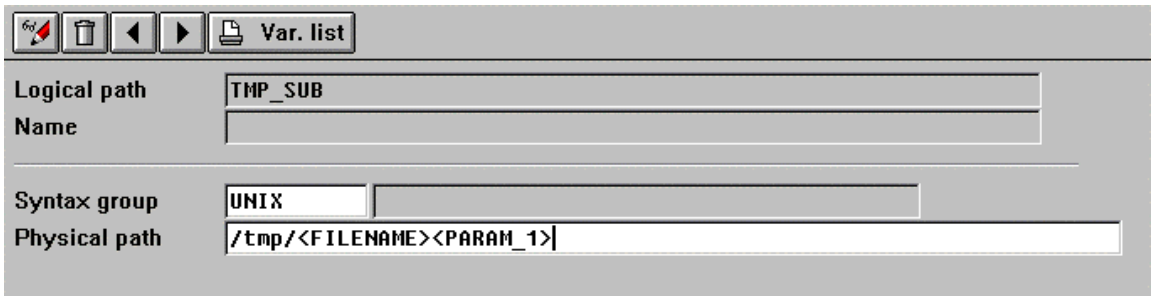
Creating and Defining Logical Paths

You can link a logical path to any logical filename. The logical path provides a platform-specific physical path for any logical filename. To create a logical filename, choose *Logical file path definition* from the Navigation group box in Transaction FILE. On the screen, you can then define a new logical path as shown below:



Save the logical path.

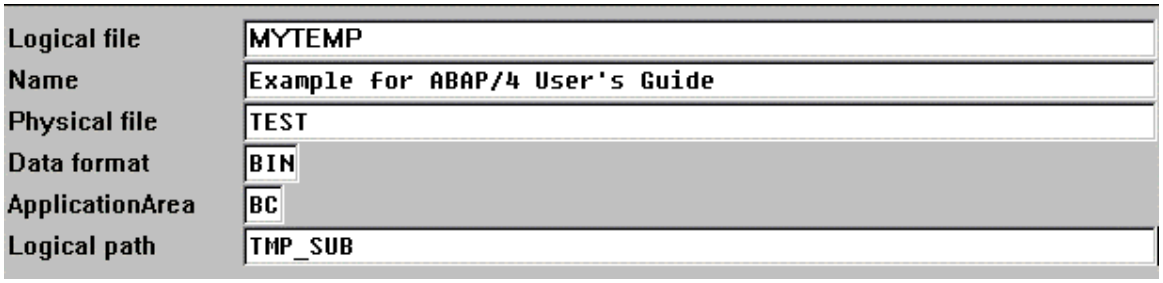
In the Navigation group box, choose *Assignment of physical paths to logical path*. This allows you to link logical paths to physical paths and syntax groups. You can now either choose a logical path and maintain its link to a syntax group, or choose *New entries* to create new links. The physical path for the syntax group UNIX might be defined as follows:



You can display lists of possible entries for the *Logical path* and *Syntax group* fields. When you create the physical path, you can use reserved words (enclosed in angled brackets). These are replaced by the appropriate values at runtime. To display a list of reserved words, place the cursor on the input field and choose *Help*. The physical path must always contain the reserved word <FILENAME>. It is replaced at runtime by the logical filename used by the logical path. The value of the reserved word <PARAM_1>, used in the previous example, is an import parameter of the function module FILE_GET_NAME.

Creating and Defining Logical Filenames

To create a logical filename, choose *Logical filename definition, client-independent* from the Navigation group box in Transaction FILE, then choose *New entries*. You define logical filenames as displayed below:



You can either define a logical filename and link it to a logical path (as displayed here), or you can enter the full physical filename in the *Physical file* field. In the latter case, the logical filename is only valid for one operating system. The rules for entering the complete physical filename are the same as for the definition of the physical path for the logical file. To display further information and a list of reserved words, choose *Help*.

If you link a logical path to a logical file, the logical file is valid for all syntax groups that have been maintained for that logical path. The filename specified under *Physical file* replaces the reserved word <FILENAME> in the physical paths that are assigned to the logical path. To make the name independent of the operating system, use names that begin with a letter, contain up to 8 letters, and do not contain special characters.

Save your changes.

Using Logical Files in ABAP Programs

To create a physical file name from a logical file name in your ABAP programs, use the function module FILE_GET_NAME. To insert the function module call in your program, choose *Edit* → *Insert statement* from the ABAP Editor screen. A dialog box appears. Select *Call Function* and enter FILE_GET_NAME. The parameters of this function module are listed below.

Import parameters

Parameters	Function
CLIENT	The maintenance tables for the logical files and paths are client-dependent. Therefore, the desired client can be imported. The current client is stored in the system field SY-MANDT.
LOGICAL_FILENAME	Enter the logical file name in upper case letters that you want to convert.
OPERATING_SYSTEM	You can import any operating system that is contained in the list in Transaction SF04 (see Assigning Operating Systems to Syntax Groups [Page 433]). The physical file name will be created according to the syntax group to which the operating system is linked. The default parameter is the value of the system field SY-OPSYS.
PARAMETER_1 PARAMETER_2	If you specify these import parameters, the reserved words <PARAM_1> and <PARAM_2> in the physical path names will be replaced by the imported values.
USE_PRESENTATION_SERVER	With this flag you can decide whether to import the operating system of the presentation server instead of the operating system imported by the parameter OPERATING_SYSTEM.
WITH_FILE_EXTENSION	If you set this flag unequal to SPACE, the file format defined for the logical file name is appended to the physical file name.

Export Parameters

Parameters	Function
EMERGENCY_FLAG	If this parameter is unequal to SPACE, no physical name is defined in the logical path. An emergency physical name was created from table FILENAME and profile parameter DIR_GLOBAL.
FILE_FORMAT	This parameter is the file format defined for the logical file name. You can use this parameter, for example, to decide in which mode the file should be opened.
FILE_NAME	This parameter is the physical file name that you can use with the ABAP statements for working with files.

Exception Parameters

Parameters	Function
FILE_NOT_FOUND	This exception is raised if the logical file is not defined.
OTHERS	This exception is raised if other errors occur.

Suppose the logical file MYTEMP and the logical path TMP_SUB are defined as in the preceding topics and we have the following program:

```
DATA: FLAG,
      FORMAT(3),
      FNAME(60).

WRITE SY-OPSYS.

CALL FUNCTION 'FILE_GET_NAME'

    EXPORTING
        LOGICAL_FILENAME       = 'MYTEMP'
        OPERATING_SYSTEM       = SY-OPSYS
        PARAMETER_1            = '01'

    IMPORTING
        EMERGENCY_FLAG         = FLAG
        FILE_FORMAT            = FORMAT
        FILE_NAME              = FNAME

    EXCEPTIONS
        FILE_NOT_FOUND         = 1
        OTHERS                 = 2.

IF SY-SUBRC = 0.
  WRITE: /  'Flag      :', FLAG,
         / 'Format    :', FORMAT,
         / 'Phys. Name:', FNAME.
ENDIF.
```

The output appears as follows:

```
HP-UX

FLAG       :

FORMAT     : BIN

Phys. Name: /tmp/TEST01
```

In this example, the R/3 System is running under the operating system HP-UX, which is member of the syntax group UNIX. The logical file name MYTEMP with the logical path TMP_SUB is converted into a physical file name /tmp/TEST01 as defined for the syntax group UNIX. The field FNAME can be used in the further flow of the program to work with file TEST01 in directory /tmp.

Suppose we have a logical file name EMPTY with the physical file name TEST, connected to a logical path that has no specification of a physical path. If you replace the EXPORTING parameter 'MYTEMP' with 'EMPTY' in the above example, the output appears as follows:

```
HP-UX

FLAG       : X

FORMAT     :

Phys. Name: /usr/sap/S11/SYS/global/TEST
```

The system created an emergency file name, whose path depends on the installation of the current R/3 System.

Modularization Techniques

All ABAP programs are modular in structure and made up of processing blocks (see Structure of ABAP Programs [Page 44]). There are two kinds of processing blocks, those that are called from outside a program by the ABAP runtime system, and those that can be called by ABAP statements in ABAP programs.

Processing blocks that are called using the ABAP runtime system:

- Event blocks
- Dialog modules

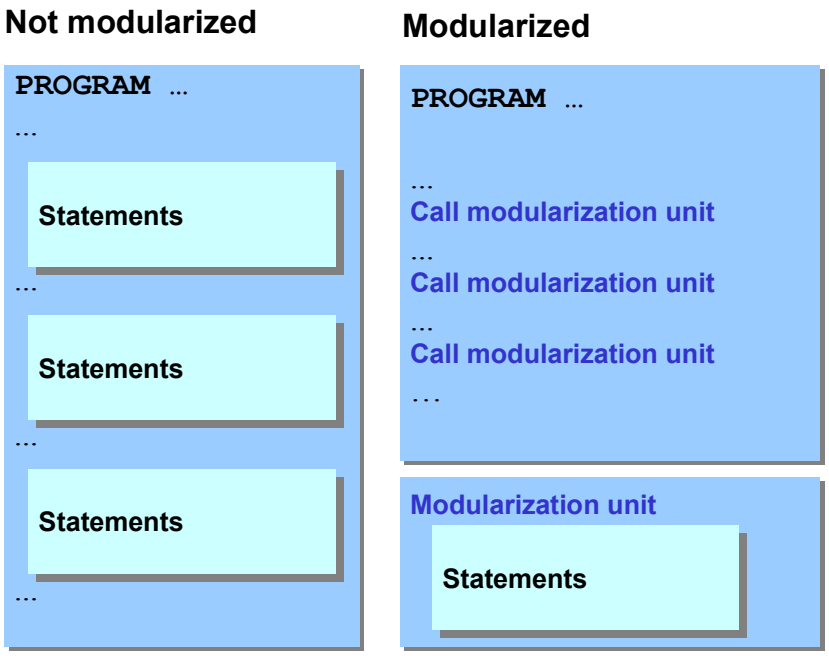
Processing blocks that are called from ABAP programs:

- Subroutines
- Function modules
- Methods

The processing blocks that you call from ABAP programs are called **procedures**.

As well as modularization in processing blocks, ABAP allows you to **modularize source code** by placing ABAP statements either in local macros or global include programs.

This section of the documentation describes both modularization in source code modules [Page 443] and in procedures [Page 449]. This kind of modularization makes ABAP programs easier to read and maintain, as well as avoiding redundancy, increasing the reusability of source code, and encapsulating data.

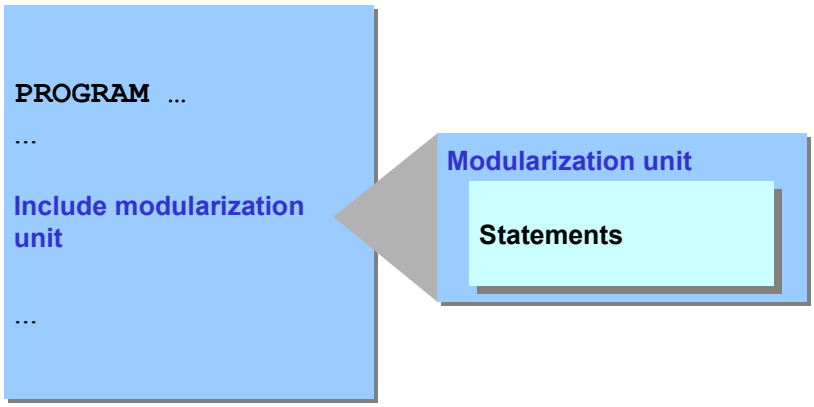


Modularization Techniques

Splitting up ABAP programs into event blocks and dialog modules is designed to help the general flow of the programs. This is discussed in ABAP Program Processing [Page 937].

Source Code Modules

When you modularize source code, you place a sequence of ABAP statements in a module. Then, instead of placing all of the statements in your main program, you just call the module.



When the program is generated, the source code in the modularization unit is treated as though it were actually physically present in the main program. Source code modules help you to avoid repeatedly writing the same set of statements and to make your programs easier to read and understand. They are not used to modularize tasks and functions. You should use procedures for this purpose.

ABAP contains two kinds of source code modules: Local modules are called macros [Page 444], and cross-program modules are called include programs [Page 447].

Macros

If you want to reuse the same set of statements more than once in a program, you can include them in a macro. For example, this can be useful for long calculations or complex WRITE statements. You can only use a macro within the program in which it is defined, and it can only be called in lines of the program following its definition.

The following statement block defines a macro <macro>:

DEFINE <macro>.

  <statements>

END-OF-DEFINITION.

You must specify complete statements between DEFINE and END-OF-DEFINITION. These statements can contain up to nine placeholders (&1, &2, ..., &9). You must define the macro **before** the point in the program at which you want to use it.

Macros **do not** belong to the definition part of the program. This means that the DEFINE...END-OF-DEFINITION block is not interpreted before the processing blocks in the program. At the same time, however, macros **are not operational statements** that are executed within a processing block at runtime. When the program is generated, macro definitions are not taken into account at the point at which they are defined. For this reason, they do not appear in the overview of the structure of ABAP programs [Page 44].

A macro definition inserts a form of shortcut at **any** point in a program and can be used at **any subsequent point** in the program. As the programmer, you must ensure that the macro definition occurs in the program before the macro itself is used. Particular care is required if you use both macros and include programs, since not all include programs are included in the syntax check (exception: TOP include).

To use a macro, use the following form:

<macro> [<p1> <p2> ... <p9>].

When the program is generated, the system replaces <macro> by the defined statements and each placeholder &i by the parameter <p_i>. You can use macros within macros. However, a macro cannot call itself.

```
DATA: RESULT TYPE I,
      N1     TYPE I VALUE 5,
      N2     TYPE I VALUE 6.

DEFINE OPERATION.
  RESULT = &1 &2 &3.
  OUTPUT   &1 &2 &3 RESULT.
END-OF-DEFINITION.

DEFINE OUTPUT.
  WRITE: / 'The result of &1 &2 &3 is', &4.
END-OF-DEFINITION.

OPERATION 4 + 3.
OPERATION 2 ** 7.
OPERATION N2 - N1.
```

The produces the following output:

The result of 4 + 3 is 7

The result of 2 ** 7 is 128

The result of N2 - N1 is 1

Here, two macros, OPERATION and OUTPUT, are defined. OUTPUT is nested in OPERATION. OPERATION is called three times with different parameters. Note how the placeholders &1, &2, ... are replaced in the macros.

The following example shows that a macro definition only works in the program lines following its definition. Do **not** copy it!

Suppose we have a program with a subroutine TEST:

```
PROGRAM MACRO_TEST.

...

FORM TEST.
  WRITE '...'.
ENDFORM.
```

Suppose we then changed the program to the following:

```
PROGRAM MACRO_TEST.

...

FORM TEST.

DEFINE MACRO.
    WRITE '...'.
  ENDFORM.
END-OF-DEFINITION.

MACRO.
```

Inserting the macro changes **nothing** in the generated form of the program. Processing blocks - here a subroutine - are always indivisible. We could also write the program as follows:

```
PROGRAM MACRO_TEST.

...

DEFINE MACRO.
    WRITE '...'.
  ENDFORM.
END-OF-DEFINITION.

...

FORM TEST.

MACRO.
```

The most essential feature of a macro definition is that it should occur **before** the macro is used.

Include Programs

Include programs are global R/3 Repository objects. They are solely for modularizing source code, and have no parameter interface.

They have the following functions:

- **Library**: Include programs allow you to use the same source code in different programs. For example, this can be useful if you have lengthy data declarations that you want to use in different programs.
- **Order**. Include programs allow you to manage complex programs in an orderly way. Function groups and module pools use include programs to store parts of the program that belong together. The ABAP Workbench supports you extensively when you create such complex programs by creating the include programs automatically and by assigning them unique names. A special include is the TOP include of a program. If you name it according to the naming convention, it is always included in program navigation and in the syntax check.

Creating Your Own Include Programs

If you create an include program yourself, you must assign it the type I in its program attributes. You can also create or change an include program by double-clicking on the name of the program after the INCLUDE statement in your ABAP program. If the program exists, the ABAP Workbench navigates to it. If it does not exist, the system creates it for you.

An include program cannot run independently, but must be built into other programs. Include programs can contain other includes.

The only restrictions for writing the source code of include programs are:

- Include programs cannot call themselves.
- Include programs must contain complete statements.

You must ensure that the statements of your include program fit logically into the source code of the programs from which it is called. Choosing *Check* while editing an include program in the ABAP Editor is normally not sufficient for this.

***INCLUDE INCL_TST.

TEXT = 'Hello!'.

Here, the syntax check reports an error because the field TEXT is not declared. However, you can include INCL_TST in any program in which a field called TEXT with the correct type has been declared.

For the syntax check to produce valid results, you must check the program in which the include occurs. The exception to this is the TOP include, the contents of which are always included when you check the syntax of another include.

Using Include Programs

To use an include program in another program, enter the statement

INCLUDE <incl>.

Include Programs

The INCLUDE statement has the same effect as copying the source code of the include program <incl> into the program. In the syntax check, the contents of the include program are also analyzed. Include programs are not loaded at runtime, but are expanded when the program is generated. Once the program has been generated, the load version contains static versions of all of its includes. If you subsequently change an include program, the programs that use it are automatically regenerated.

The INCLUDE statement must be the only statement on a line and cannot extend over several lines.

Suppose we have the following program:

```
***INCLUDE STARTTXT.

WRITE: / 'Program started by', SY-UNAME,
       / 'on host', SY-HOST,
         'date:', SY-DATUM, 'time:', SY-UZEIT.
ULINE.
```

We can then include this program in any other ABAP program to display a standard list header.

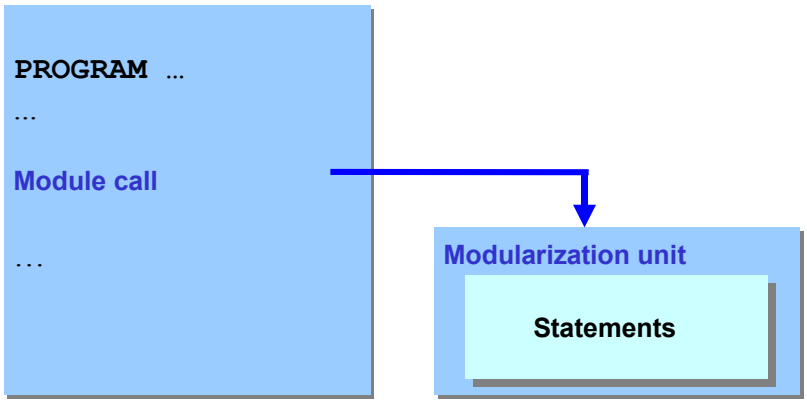
```
PROGRAM SAPMZTST.
INCLUDE STARTTXT.

.....
```

This could produce the following output:

```
Program started by KELLERH

on host ds0025   date: 03/19/1998 time: 09:00:39
________________________________________________________________

.................
```

Procedures

Procedures contain a set of statements, and are called from other ABAP programs.



You define procedures in ABAP programs. When the program is generated, they remain as standalone modules. You can call procedures in the program in which they are defined, or from external programs. Procedures have an **interface** for passing data, and can also contain **local data**.

ABAP contains the following kinds of procedures:

- Subroutines [Page 451]

  Subroutines are principally for local modularization, that is, they are generally called from the program in which they are defined. You can use subroutines to write functions that are used repeatedly within a program. You can define subroutines in any ABAP program.

- Function Modules [Page 480]

  Function modules are for global modularization, that is, they are always called from a different program. Function modules contain functions that are used in the same form by many different programs. They are important in the R/3 System for encapsulating processing logic and making it reusable. Function modules must be defined in a function group, and can be called from any program.

- Methods [Page 1291]

  Methods describe the functions and behavior of classes and their instances in ABAP Objects. Methods must be defined in classes. When you call them, you must observe certain special rules of object-oriented programming.

You can call procedures either internally or externally. If you call procedures externally, it is important that you understand how memory is organized in the R/3 System, how screens are processed, and how interface work areas are used.

Organization of External Procedure Calls [Page 494]

Subroutines

Subroutines are procedures that you can define in any ABAP program and also call from any program. Subroutines are normally called internally, that is, they contain sections of code or algorithms that are used frequently **locally**. If you want a function to be reusable throughout the system, use a function module.

Defining Subroutines [Page 452]

Calling Subroutines [Page 466]

Defining Subroutines

A subroutine is a block of code introduced by FORM and concluded by ENDFORM.

FORM <subr> [USING ... [VALUE(]<p_i>[)] [TYPE <t>|LIKE <f>]...]
[CHANGING... [VALUE(]<p_i>[)] [TYPE <t>|LIKE <f>]...].

...

ENDFORM.

<subr> is the name of the subroutine. The optional additions USING and CHANGING define the **parameter interface**. Like any other processing block, subroutines cannot be nested. You should therefore place your subroutine definitions at the end of the program, especially for executable programs (type 1). In this way, you eliminate the risk of accidentally ending an event block in the wrong place by inserting a FORM...ENDFORM block.

Data Handling in Subroutines

Global Data from the Main Program [Page 453]

Local Data in the Subroutine [Page 455]

The Parameter Interface [Page 459]

Terminating Subroutines

Terminating a Subroutine [Page 464]

Global Data from the Main Program

Subroutines can access all of the global data in the program in which they are defined (main program). You therefore do not need to define a parameter interface if you do not want to change any data in the subroutine, or if very little data is involved.

```
FORM HEADER.
  WRITE: / 'Program started by', SY-UNAME,
         / 'on host', SY-HOST,
           'date:', SY-DATUM, 'time:', SY-UZEIT.
  ULINE.
ENDFORM.
```

This example creates a subroutine called HEADER, which, like the example of an include program [Page 447], displays a list header.

However, if you want subroutines to perform complex operations on data without affecting the global data in the program, you should define a parameter interface through which you can pass exactly the data you need. In the interests of good programming style and encapsulation, you should **always** use a parameter interface, at least when the subroutine changes data.

Protecting Global Data Objects Against Changes

To prevent the value of a global data object from being changed inside a subroutine, use the following statement:

LOCAL <f>.

This statement may only occur between the FORM and ENDFORM statements. With LOCAL, you can preserve the values of global data objects which cannot be hidden by a data declaration inside the subroutine.

For example, you cannot declare a table work area that is defined by the TABLES statement with another TABLES statement inside a subroutine. If you want to use the table work area locally, but preserve its contents outside the subroutine, you must use the LOCAL statement.

```
PROGRAM FORM_TEST.

TABLES SFLIGHT.

PERFORM TABTEST1.
WRITE: / SFLIGHT-PLANETYPE, SFLIGHT-PRICE.

PERFORM TABTEST2.
WRITE: / SFLIGHT-PLANETYPE, SFLIGHT-PRICE.

FORM TABTEST1.
  SFLIGHT-PLANETYPE = 'A310'.
  SFLIGHT-PRICE = '150.00'.
  WRITE: / SFLIGHT-PLANETYPE, SFLIGHT-PRICE.
ENDFORM.
```

```
FORM TABTEST2.
  LOCAL SFLIGHT.
  SFLIGHT-PLANETYPE = 'B747'.
  SFLIGHT-PRICE = '500.00'.
  WRITE: / SFLIGHT-PLANETYPE, SFLIGHT-PRICE.
ENDFORM.
```

When you run the program, the following is displayed:

```
A310        150.00

A310        150.00

B747        500.00

A310        150.00
```

The program creates a table work area SFLIGHT for the database table SFLIGHT. Different values are assigned to the table work area SFLIGHT in TABTEST1 and TABTEST2. While the values assigned in TABTEST1 are valid globally, the values assigned in TABTEST2 are only valid locally.

Local Data in the Subroutine

Data declarations in procedures create local data types and objects that are only visible within that procedure. There are two kinds of data types and objects - dynamic and static. Dynamic data objects only exist while the subroutine is running, while static objects still exist after the subroutine has finished running, and retain their values until the next time the subroutine is called. You can also declare local field symbols. A special kind of data object for subroutines are copies of global data on a local data stack. You define and address them using field symbols.

Dynamic Local Data Types and Objects

Local data types and objects declared in subroutines using the TYPES and DATA statements are deleted when the subroutine ends, and recreated each time the routine is called.

Every subroutine has its own local namespace. If you declare a local data type or object with the same name as a global data type or object, the global type or object cannot be addressed from within the subroutine. Local data types or data objects hide identically named global data types or objects. This means that if you use the name of a data type or object in the subroutine, you always address a locally declared object - if this exists - and otherwise a globally declared one. To avoid this, you must assign other names to local types and objects. For example, you might name all of your local data starting with ‘L_’.

```
PROGRAM FORM_TEST.

TYPES WORD(10) TYPE C.
DATA  TEXT TYPE WORD.

TEXT = '1234567890'. WRITE / TEXT.

PERFORM DATATEST.

WRITE / TEXT.

FORM DATATEST.
  TYPES WORD(5) TYPE C.
  DATA  TEXT TYPE WORD.
  TEXT = 'ABCDEFGHJK'. WRITE / TEXT.
ENDFORM.
```

When you run the program, the following is displayed:

```
1234567890

ABCDE

1234567890
```

In this example, a data type WORD and a data object TEXT with type WORD are declared globally in the main program. After assigning a value to TEXT and writing it to the list, the internal subroutine DATATEST is called. Inside the subroutine, a data type WORD and a data object TEXT with type WORD are declared locally. They hide the global type and object. Only after the subroutine is finished are the global definitions are valid again.

Static Local Data Objects

If you want to keep the value of a local data object after exiting the subroutine, you must use the STATICS statement to declare it instead of the DATA statement. With STATICS you declare a data object that is globally defined, but only locally visible from the subroutine in which it is defined.

```
PROGRAM FORM_TEST.

PERFORM DATATEST1.
PERFORM DATATEST1.

SKIP.

PERFORM DATATEST2.
PERFORM DATATEST2.

FORM DATATEST1.
  TYPES F_WORD(5) TYPE C.
  DATA  F_TEXT TYPE F_WORD VALUE 'INIT'.
  WRITE F_TEXT.
  F_TEXT = '12345'.
  WRITE F_TEXT.
ENDFORM.

FORM DATATEST2.
  TYPES    F_WORD(5) TYPE C.
  STATICS  F_TEXT TYPE F_WORD VALUE 'INIT'.
  WRITE F_TEXT.
  F_TEXT = 'ABCDE'.
  WRITE F_TEXT.
ENDFORM.
```

When you run the program, the following is displayed:

```
INIT  12345 INIT  12345

INIT  ABCDE ABCDE ABCDE
```

In this example, two similar subroutines DATATEST1 and DATATEST2 are defined. In DATATEST2, the STATICS statement is used instead of the DATA statement to declare the data object F_TEXT. During each call of DATATEST1, F_TEXT is initialized again, but it keeps its value for DATATEST2. The VALUE option of the STATICS statement functions only during the first call of DATATEST2.

Local Field Symbols

All field symbols declared in a subroutine using the FIELD-SYMBOLS [Page 203] statement are local. The following rules apply to local field symbols:

- You cannot address local field symbols outside the subroutine.
- When you call the subroutine, any local field symbols are unassigned, even if you assigned a field to them the last time the subroutine was called.
- Local field symbols can have the same names as global field symbols. If they do, they hide the global field symbols within the subroutine.

- You can also declare structured field symbols locally. They can have local structures and you can assign local fields to them.

Local Copies of Global Fields

In a subroutine, you can create local copies of global data on the local stack. To do this, use a local field symbol and the following variant of the ASSIGN statement:

ASSIGN LOCAL COPY OF <f> TO <FS>.

The system places a copy of the specified global field <f> on the stack. In the subroutine, you can access and change this copy without changing the global data by addressing the field symbol <FS>.

You can use the LOCAL COPY OF addition with all variants of the ASSIGN statement except ASSIGN COMPONENT.

Other variants of the ASSIGN statement that are used in subroutines are:

ASSIGN LOCAL COPY OF INITIAL <f> TO <FS>.

This statement creates an initialized copy of the global field <f> on the stack without copying the field contents.

ASSIGN LOCAL COPY OF INITIAL LINE OF <itab> TO <FS>.

This statement creates an initial copy of the line of a global internal table <itab> on the stack.

ASSIGN LOCAL COPY OF INITIAL LINE OF (<f>) TO <FS>.

This statement creates an initial copy of the line of a global internal table <itab> on the stack. The internal table is specified dynamically as the contents of the field <f>.

```
DATA TEXT(5) VALUE 'Text1'.
PERFORM ROUTINE.
WRITE TEXT.
FORM ROUTINE.
  FIELD-SYMBOLS <FS>.
  ASSIGN LOCAL COPY OF TEXT TO <FS>.
  WRITE <FS>.
  <FS> = 'Text2'.
  WRITE <FS>.
  ASSIGN TEXT TO <FS>.
  WRITE <FS>.
  <FS> = 'Text3'.
ENDFORM.
```

The output is:

Text1 Text2 Text1 Text3

By assigning TEXT to <FS> in the subroutine ROUTINE in the program FORMPOOL, you place a copy of the field TEXT on the local data stack. By addressing <FS>, you can read and change this copy. The global field TEXT is not affected by operations on the local field. After you have assigned the field to the field symbol without the LOCAL COPY OF addition, it points directly to the global field. Operations with the field symbol then affect the global field.

The Parameter Interface

The USING and CHANGING additions in the FORM statement define the formal parameters of a subroutine. The sequence of the additions is fixed. Each addition can be followed by a list of any number of formal parameters. When you call a subroutine, you must fill all formal parameters with the values from the actual parameters. At the end of the subroutine, the formal parameters are passed back to the corresponding actual parameters.

Within a subroutine, formal parameters behave like dynamic local data. You can use them in the same way as normal local data objects that you would declare with the DATA statement. They mask global data objects with the same name. The value of the parameters at the start of the subroutine is the value passed from the corresponding actual parameter.

Subroutines can have the following formal parameters:

Parameters Passed by Reference

You list these parameters after USING or CHANGING without the VALUE addition:

FORM <subr> USING ... <p_i> [TYPE <t>|LIKE <f>] ...
CHANGING ... <p_i> [TYPE <t>|LIKE <f>] ...

The formal parameter occupies no memory of its own. During a subroutine call, only the address of the actual parameter is transferred to the formal parameter. The subroutine works with the field from the calling program. If the value of the formal parameter changes, the contents of the actual parameter in the calling program also change.

For calling by reference, USING and CHANGING are equivalent. For documentation purposes, you should use USING for input parameters which are not changed in the subroutine, and CHANGING for output parameters which are changed in the subroutine.

To avoid the value of an actual parameter being changed automatically, you must pass it by value.

Input Parameters That Pass Values

You list these parameters after USING with the VALUE addition:

FORM <subr> USING ... VALUE(<p_i>) [TYPE <t>|LIKE <f>] ...

The formal parameter occupies its own memory space. When you call the subroutine, the value of the actual parameter is passed to the formal parameter. If the value of the formal parameter changes, this has no effect on the actual parameter.

Output Parameters That Pass Values

You list these parameters after CHANGING with the VALUE addition:

FORM <subr> CHANGING ... VALUE(<p_i>) [TYPE <t>|LIKE <f>] ...

The formal parameter occupies its own memory space. When you call the subroutine, the value of the actual parameter is passed to the formal parameter. If the subroutine concludes successfully, that is, when the ENDFORM statement occurs, or when the subroutine is terminated through a CHECK or EXIT statement, the current value of the formal parameter is copied into the actual parameter.

If the subroutine terminates prematurely due to an error message, no value is passed. It only makes sense to terminate a subroutine through an error message in the PAI processing of a

screen, that is, in a PAI module, in the AT SELECTION-SCREEN event, or after an interactive list event.

Specifying the Type of Formal Parameters

Formal parameters can have any valid ABAP data type. You can specify the type of a formal parameter, either generically or fully, using the TYPE or LIKE addition. If you specify a generic type, the type of the formal parameter is either partially specified or not specified at all. Any attributes that are not specified are inherited from the corresponding actual parameter when the subroutine is called. If you specify the type fully, all of the technical attributes of the formal parameter are defined with the subroutine definition.

The following remarks about specifying the types of parameters also apply to the parameters of other procedures (function modules and methods).

If you have specified the type of the formal parameters, the system checks that the corresponding actual parameters are compatible when the subroutine is called. For internal subroutines, the system checks this in the syntax check. For external subroutines, the check cannot occur until runtime.

By specifying the type, you ensure that a subroutine always works with the correct data type. Generic formal parameters allow a large degree of freedom when you call subroutines, since you can pass data of any type. This restricts accordingly the options for processing data in the subroutine, since the operations must be valid for all data types. For example, assigning one data object to another may not even be possible for all data types. If you specify the types of subroutine parameters, you can perform a much wider range of operations, since only the data appropriate to those operations can be passed in the call. If you want to process structured data objects component by component in a subroutine, you must specify the type of the parameter.

Specifying Generic Types

The following types allow you more freedom when using actual parameters. The actual parameter need only have the selection of attributes possessed by the formal parameter. The formal parameter adopts its remaining unnamed attributes from the actual parameter.

Type specification	Check for actual parameters
No type specification TYPE ANY	The subroutine accepts actual parameters of any type. The formal parameter inherits all of the technical attributes of the actual parameter.
TYPE C, N, P, or X	The subroutine only accepts actual parameters with the type C, N, P, or X. The formal parameter inherits the field length and DECIMALS specification (for type P) from the actual parameter.
TYPE TABLE	The system checks whether the actual parameter is a standard internal table. This is a shortened form of TYPE STANDARD TABLE (see below).
TYPE ANY TABLE	The system checks whether the actual parameter is an internal table. The formal parameter inherits all of the attributes (line type, table type, key) from the actual parameter.
TYPE INDEX TABLE	The system checks whether the actual parameter is an index table (standard or sorted table). The formal parameter inherits all of the attributes (line type, table type, key) from the actual parameter.
TYPE STANDARD TABLE	The system checks whether the actual parameter is a standard internal table. The formal parameter inherits all of the attributes (line type, key) from the actual parameter.

TYPE SORTED TABLE	The system checks whether the actual parameter is a sorted table. The formal parameter inherits all of the attributes (line type, key) from the actual parameter.
TYPE HASHED TABLE	The system checks whether the actual parameter is a hashed table. The formal parameter inherits all of the attributes (line type, key) from the actual parameter.

Note that formal parameters inherit the attributes of their corresponding actual parameters dynamically at runtime, and so they cannot be identified in the program code. For example, you cannot address an inherited table key statically in a subroutine, but you probably can dynamically.

```
TYPES: BEGIN OF LINE,
         COL1,
         COL2,
       END OF LINE.
DATA: WA TYPE LINE,
      ITAB TYPE HASHED TABLE OF LINE WITH UNIQUE KEY COL1,
      KEY(4) VALUE 'COL1'.
WA-COL1 = 'X'. INSERT WA INTO TABLE ITAB.
WA-COL1 = 'Y'. INSERT WA INTO TABLE ITAB.
PERFORM DEMO USING ITAB.
FORM DEMO USING P TYPE ANY TABLE.
  ...
  READ TABLE P WITH TABLE KEY (KEY) = 'X' INTO WA.
  ...
ENDFORM.
```

The table key is addressed dynamically in the subroutine. However, the static address

```
READ TABLE P WITH TABLE KEY COL1 = 'X' INTO WA.
```

is syntactically incorrect, since the formal parameter P does not adopt the key of table ITAB until runtime.

Specifying Full Types

When you use the following types, the technical attributes of the formal parameters are fully specified. The technical attributes of the actual parameter must correspond to those of the formal parameter.

Type specification	Technical attributes of the formal parameter
TYPE D, F, I, or T	The formal parameter has the technical attributes of the predefined elementary type
TYPE <type>	The formal parameter has the type <type> This is a data type defined within the program using the TYPES statement, or a type from the ABAP Dictionary
TYPE REF TO <cif>	The formal parameter is a reference variable (ABAP Objects) for the class or interface <cif>
TYPE LINE OF <itab>	The formal parameter has the same type as a line of the internal table <itab> defined using a TYPES statement or defined in the ABAP Dictionary
LIKE <f>	The formal parameter has the same type as an internal data object <f> or structure, or a database table from the ABAP Dictionary

The Parameter Interface

When you use a formal parameter that is fully typed, you can address its attributes statically in the program, since they are recognized in the source code.

Structured Formal Parameters

Since formal parameters can take any valid ABAP data type, they can also take structures and internal tables with a structured line type, as long as the type of the formal parameter is **fully specified**. You can address the components of the structure statically in the subroutine.

Generic Structures

If you pass a structured actual parameter generically to a formal parameter whose type is not correctly specified, you cannot address the components of the structure statically in the subroutine. For internal tables, this means that only line operations are possible.

To access the components of a generically passed structure, you must use field symbols, and the assignment

ASSIGN COMPONENT <idx>|<name> OF STRUCTURE <s> TO <FS>. [Page 213]

<idx> is interpreted as the component number and the contents of <name> are interpreted as a component name in the generic structure <s>.

```
DATA: BEGIN OF LINE,
        COL1 VALUE 'X',
        COL2 VALUE 'Y',
      END OF LINE.
DATA COMP(4) VALUE 'COL1'.
PERFORM DEMO USING LINE.
FORM DEMO USING P TYPE ANY.
  FIELD-SYMBOLS <FS>.
  ASSIGN COMPONENT COMP OF STRUCTURE P TO <FS>.
  WRITE <FS>.
  ASSIGN COMPONENT 2 OF STRUCTURE P TO <FS>.
  WRITE <FS>.
ENDFORM.
```

The output is:

```
X Y
```

The components COL1 and COL2 of the structure of P (passed generically) are assigned to the field symbol <FS>. You cannot address the components directly **either statically or dynamically**.

Fitting Parameters into Structures

Instead of using TYPE or LIKE, you can specify the type of a structure as follows:

... $<p_i>$ [STRUCTURE <s>] ...

where <s> is a local structure in the program (data object, not a type) or a **flat** structure from the ABAP Dictionary. The formal parameter is structured according to <s>, and you can address its individual components in the subroutine. When the actual parameter is passed, the system only checks to ensure that the actual parameter is at least as long as the structure. STRUCTURE therefore allows you to force a structured **view** of any actual parameter.

```
DATA: BEGIN OF LINE,
        COL1,
        COL2,
      END OF LINE.
DATA TEXT(2) VALUE 'XY'.
PERFORM DEMO USING TEXT.
FORM DEMO USING P STRUCTURE LINE.
  WRITE: P-COL1, P-COL2.
ENDFORM.
```

The output is:

```
X Y
```

The string TEXT is fitted into the structure LINE.

The TABLES Addition

To ensure compatibility with previous releases, the following addition is still allowed before the USING and CHANGING additions:

FORM <subr> TABLES ... <itab$_i$> [TYPE <t>|LIKE <f>] ...

The formal parameters <itab$_i$> are defined as standard internal tables **with header lines**. If you use an internal table without header line as the corresponding actual parameter for a formal parameter of this type, the system creates a **local header line** in the subroutine for the formal parameter. If you pass an internal table with a header line, the table body and the table work area are passed to the subroutine. Formal parameters defined using TABLES cannot be passed by reference. If you want to address the components of structured lines, you must specify the type of the TABLES parameter accordingly.

From Release 3.0, you should use USING or CHANGING instead of the TABLES addition for internal tables, although for performance reasons, you should not pass them by value.

Terminating Subroutines

A subroutine normally ends at the ENDFORM statement. However, you can terminate them earlier by using the EXIT or CHECK statement. When you terminate a subroutine with EXIT or CHECK, the current values of the output parameters (CHANGING parameters passed by value) are passed back to the corresponding actual parameter.

Use EXIT to terminate a subroutine unconditionally. The calling program regains control at the statement following the PERFORM statement.

```
PROGRAM FORM_TEST.
PERFORM TERMINATE.
WRITE  'The End'.
FORM TERMINATE.
  WRITE '1'.
  WRITE '2'.
  WRITE '3'.
  EXIT.
  WRITE '4'.
ENDFORM.
```

The produces the following output:
1 2 3 The End
In this example, the subroutine TERMINATE is terminated after the third WRITE statement.

Use CHECK to terminate a subroutine conditionally. If the logical expression in the CHECK statement is untrue, the subroutine is terminated, and the calling program resumes processing after the PERFORM statement.

```
PROGRAM FORM_TEST.

DATA: NUM1 TYPE I,
      NUM2 TYPE I,
      RES  TYPE P DECIMALS 2.

NUM1 = 3. NUM2 = 4.
PERFORM DIVIDE USING NUM1 NUM2 CHANGING RES.

NUM1 = 5. NUM2 = 0.
PERFORM DIVIDE USING NUM1 NUM2 CHANGING RES.

NUM1 = 2. NUM2 = 3.
PERFORM DIVIDE USING NUM1 NUM2 CHANGING RES.

FORM DIVIDE USING N1 N2 CHANGING R.
  CHECK N2 NE 0.
  R = N1 / N2.
  WRITE: / N1, '/', N2, '=', R.
ENDFORM.
```

The produces the following output:

```
3 /           4 =           0.75
```

```
2 /           3 =           0.67
```

In this example, the system leaves the subroutine DIVIDE during the second call after the CHECK statement because the value of N2 is zero.

If the EXIT or CHECK statement occurs within a loop in a subroutine, it applies to the loop, and not to the subroutine. EXIT and CHECK terminate loops in different ways (see Loops [Page 245]), but behave identically when terminating subroutines.

Calling Subroutines

You call subroutines using the statement

PERFORM... [USING ... <p_i>...]
[CHANGING... <p_i>...].

Subroutines can call other subroutines (nested calls) and may also call themselves (recursive calls). Once a subroutine has finished running, the calling program carries on processing after the PERFORM statement. You can use the USING and CHANGING additions to supply values to the parameter interface of the subroutine.

Naming Subroutines [Page 467]

Passing Parameters to Subroutines [Page 470]

Naming Subroutines

With the PERFORM statement, you can call subroutines which are coded in the same ABAP program (internal calls), or subroutines which are coded in other ABAP programs (external calls).

You can also specify the name of the subroutine dynamically at runtime, and call subroutines from a list.

Internal Subroutine Calls

To call a subroutine defined in the same program, you need only specify its name in the PERFORM statement:

PERFORM <subr> [USING ... <p_i>...]
[CHANGING... <p_i>...].

The internal subroutine can access all of the global data of the calling program.

```
PROGRAM FORM_TEST.
DATA: NUM1 TYPE I,
      NUM2 TYPE I,
      SUM  TYPE I.
NUM1 = 2. NUM2 = 4.
PERFORM ADDIT.
NUM1 = 7. NUM2 = 11.
PERFORM ADDIT.
FORM ADDIT.
  SUM = NUM1 + NUM2.
  PERFORM OUT.
ENDFORM.
FORM OUT.
  WRITE: / 'Sum of', NUM1, 'and', NUM2, 'is', SUM.
ENDFORM.
```

The produces the following output:

```
Sum of        2 and        4 is        6
Sum of        7 and       11 is       18
```

In this example, two internal subroutines ADDIT and OUT are defined at the end of the program. ADDIT is called from the program and OUT is called from ADDIT. The subroutines have access to the global fields NUM1, NUM2, and SUM.

External subroutine calls

The principal function of subroutines is for modularizing and structuring local programs. However, subroutines can also be called externally from other ABAP programs. In an extreme case, you might have an ABAP program that contained nothing but subroutines. These programs cannot run on their own, but are used by other ABAP programs as pools of external subroutines.

However, if you want to make a function available throughout the system, you should use function modules instead of external subroutines. You create function modules in the ABAP Workbench using the Function Builder. They are stored in a central library, and have a defined release procedure.

You can encapsulate functions and data in the attributes and methods of classes in ABAP Objects. For any requirements that exceed pure functions, you can use global classes instead of external subroutines.

Naming Subroutines

When you call a subroutine externally, you must know the name of the program in which it is defined:
PERFORM <subr>(<prog>) [USING ... <p_i>...]
[CHANGING... <p_i>...] [IF FOUND].
You specify the program name <prog> statically. You can use the IF FOUND option to prevent a runtime error from occurring if the program <prog> does not contain a subroutine <sub>. In this case, the system simply ignores the PERFORM statement. When you call an external subroutine, the system loads the whole of the program containing the subroutine into the internal session of the calling program (if it has not already been loaded). For further information, refer to Memory Structure of an ABAP Program [Page 66]. In order to save memory space, you should keep the number of subroutines called in different programs to a minimum.

Suppose a program contains the following subroutine:

```
PROGRAM FORMPOOL.
FORM HEADER.
  WRITE: / 'Program started by', SY-UNAME,
         / 'on host', SY-HOST,
           'date:', SY-DATUM, 'time:', SY-UZEIT.
  ULINE.
ENDFORM.
```

The subroutine can then be called from another program as follows:

```
PROGRAM FORM_TEST.
PERFORM HEADER(FORMPOOL) IF FOUND.
```

In this example, no data is passed between calling program and subroutine.

Specifying Subroutines Dynamically

You can specify the name of a subroutine and, in the case of external calls, the name of the program in which it occurs, dynamically as follows:

PERFORM (<fsubr>)[IN PROGRAM (<fprog>)][USING ... <p_i>...]
[CHANGING... <p_i>...]
[IF FOUND].

The names of the subroutine and the external program are the contents of the fields <fsubr> and <fprog> respectively. By using the option IF FOUND, you can prevent a runtime error from being triggered if <fprog> does not contain a subroutine with the name <fsubr>. If you omit the parentheses, this variant of the PERFORM statement behaves like the static variant.

Suppose a program contains the following subroutines:

```
PROGRAM FORMPOOL.
FORM SUB1.
  WRITE: / 'Subroutine 1'.
ENDFORM.
FORM SUB2.
  WRITE: / 'Subroutine 2'.
ENDFORM.
```

Dynamic subroutine specification:

```
PROGRAM FORM_TEST.
DATA: PROGNAME(8) VALUE 'FORMPOOL',
      SUBRNAME(8).
```

```
SUBRNAME = 'SUB1'.
PERFORM (SUBRNAME) IN PROGRAM (PROGNAME) IF FOUND.
SUBRNAME = 'SUB2'.
PERFORM (SUBRNAME) IN PROGRAM (PROGNAME) IF FOUND.
```

The produces the following output:

```
Subroutine 1
Subroutine 2
```

The character field PROGNAME contains the name of the program, in which the subroutines are contained. The names of the subroutines are assigned to the character field SUBRNAME.

Calling Subroutines from a List

You can call a subroutine from a list as follows:

PERFORM <idx> OF <$subr_1$> <$subr_2$>.... <$subr_n$>.

The system calls the subroutine specified in the subroutine list in position <idx>. You can only use this variant of the PERFORM statement for internal subroutine calls, and **only for subroutines without a parameter interface**. The field <idx> can be a variable or a literal.

```
PROGRAM FORM_TEST.
DO 2 TIMES.
  PERFORM SY-INDEX OF SUB1 SUB2.
ENDDO.
FORM SUB1.
  WRITE / 'Subroutine 1'.
ENDFORM.
FORM SUB2.
  WRITE / 'Subroutine 2'.
ENDFORM.
```

The produces the following output:

```
Subroutine 1
Subroutine 2
```

In this example, the two internal subroutines, SUB1 and SUB2, are called consecutively from a list.

Passing Parameters to Subroutines

If a subroutine has a parameter interface [Page 459], you must supply values to **all** of the formal parameters in its interface when you call it. You list the actual parameters after the USING or CHANGING addition in the PERFORM statement.

When you pass the values, the **sequence** of the actual parameters in the PERFORM statement is crucial. The value of the first actual parameter in the list is passed to the first formal parameter, the second to the second, and so on. The additions USING and CHANGING have exactly the same meaning. You only need to use one or the other. However, for documentary reasons, it is a good idea to divide the parameters in the same way in which they occur in the interface definition.

Actual parameters can be any data objects or field symbols of the calling program whose technical attributes are compatible with the type specified for the corresponding formal parameter. When you specify the actual parameters, note that any that you pass by reference to a formal parameter, and any that you pass by value to an output parameter, can be changed by the subroutine. You should therefore ensure that only data objects that you want to be changed appear in the corresponding position of the actual parameter list.

If a subroutine contains TABLES parameters in its interface, you must specify them in a TABLES addition of the PERFORM statement before the USING and CHANGING parameters. TABLES parameters are only supported to ensure compatibility with earlier releases, and should no longer be used.

You can specify actual parameters with **variable** offset and length specifications. Offset specifications for actual parameters function as offset specifications for field symbols. You can select memory areas that lie outside the boundaries of the specified actual parameter.

```
PROGRAM FORM_TEST.
DATA: A1 TYPE P DECIMALS 3,
      A2 TYPE I,
      A3 TYPE D,
      A4 TYPE SPFLI-CARRID,
      A5 TYPE C.
...
PERFORM SUBR USING A1 A2 A3 A4 A5.
...
PERFORM SUBR CHANGING A1 A2 A3 A4 A5.
...
PERFORM SUBR USING A1 A2 A3
             CHANGING A4 A5.
...
FORM SUBR USING
            VALUE(F1) TYPE P
            VALUE(F2) TYPE I
            F3        LIKE A3
          CHANGING
            VALUE(F4) TYPE SPFLI-CARRID
            F5.
...
ENDFORM.
```

This example defines a subroutine SUBR with a parameter interface consisting of five formal parameter F1 to F5. The subroutine is called internally three times. The actual parameters are the data objects A1 to A5. **The three subroutine calls are all equally valid.** There are further PERFORM statements that are also equally valid, so long as the sequence of the actual parameters remains unchanged. In each call, A1 is passed to F1, A2 to F2, and so on. When the subroutine ends, A3, A4, and A5 receive the values of F3, F4, and F5 respectively. The third of the subroutine calls documents in the program what the parameter interface of the subroutine shows, namely that only A4 and A5 are changed. Whether A3 is changed depends on the way in which the subroutine is programmed.

The following example shows how generically-typed formal parameters inherit their technical attributes from their corresponding actual parameters.

```
REPORT FORMTEST.
DATA:
  DATE1      TYPE D,           DATE2      TYPE T,
  STRING1(6) TYPE C,           STRING2(8) TYPE C,
  NUMBER1    TYPE P DECIMALS 2,  NUMBER2    TYPE P,
  COUNT1     TYPE I,           COUNT2     TYPE I.
PERFORM TYPETEST USING DATE1 STRING1 NUMBER1 COUNT1.
SKIP.
PERFORM TYPETEST USING DATE2 STRING2 NUMBER2 COUNT2.
FORM TYPETEST USING NOW
                    TXT TYPE C
                    VALUE(NUM) TYPE P
                    INT TYPE I.
  DATA: T.
  DESCRIBE FIELD NOW TYPE T.
  WRITE: / 'Type of NOW is', T.
  DESCRIBE FIELD TXT LENGTH T.
  WRITE: / 'Length of TXT is', T.
  DESCRIBE FIELD NUM DECIMALS T.
  WRITE: / 'Decimals of NUM are', T.
  DESCRIBE FIELD INT TYPE T.
  WRITE: / 'Type of INT is', T.
ENDFORM.
```

The produces the following output:

```
TYPE of NOW is D
Length of TXT is 6
Decimals of NUM are 2
Type of INT is I
TYPE of NOW is T
Length of TXT is 8
Decimals of NUM are 0
Type of INT is I
```

An internal subroutine TYPETEST is called twice with different actual parameters. All actual and formal parameters are compatible and no error message occurs during the syntax check. Had you declared COUNT2 with type F instead of type I, the syntax check would have returned an error, since the formal parameter INT is specified with type I. The formal parameters with generic types adopt different technical attributes depending on their corresponding technical attributes.

Examples of Subroutines

Example of Passing Parameters by Reference

```
PROGRAM FORM_TEST.

DATA: NUM1 TYPE I,
      NUM2 TYPE I,
      SUM  TYPE I.

NUM1 = 2. NUM2 = 4.
PERFORM ADDIT USING NUM1 NUM2 CHANGING SUM.

NUM1 = 7. NUM2 = 11.
PERFORM ADDIT USING NUM1 NUM2 CHANGING SUM.

FORM ADDIT
      USING ADD_NUM1
            ADD_NUM2
      CHANGING ADD_SUM.

  ADD_SUM = ADD_NUM1 + ADD_NUM2.
  PERFORM OUT USING ADD_NUM1 ADD_NUM2 ADD_SUM.

ENDFORM.

FORM OUT
      USING OUT_NUM1
            OUT_NUM2
            OUT_SUM.

  WRITE: / 'Sum of', OUT_NUM1, 'and', OUT_NUM2, 'is', OUT_SUM.

ENDFORM.
```

The produces the following output:

```
Sum of       2 and       4 is       6

Sum of       7 and      11 is      18
```

In this example, the actual parameters NUM1, NUM2, and SUM are passed by reference to the formal parameters of the subroutine ADDIT. After changing ADD_SUM, the latter parameters are then passed to the formal parameters OUT_NUM1, OUT_NUM2, and OUT_SUM of the subroutine OUT.

Input parameters which are changed in the subroutine are also changed in the calling program. To prevent this, you must pass the parameter by value in a USING addition.

Example of passing parameters by reference

```
PROGRAM FORM_TEST.
```

```
DATA: NUM  TYPE I VALUE 5,
      FAC  TYPE I VALUE 0.

PERFORM FACT USING NUM CHANGING FAC.

WRITE: / 'Factorial of', NUM, 'is', FAC.

FORM FACT
       USING VALUE(F_NUM)
       CHANGING F_FACT.

  F_FACT = 1.
  WHILE F_NUM GE 1.
    F_FACT = F_FACT * F_NUM.
    F_NUM = F_NUM - 1.
  ENDWHILE.

ENDFORM.
```

The produces the following output:

```
Factorial of         5 is        120
```

To ensure that an input parameter is not changed in the calling program, even if it is changed in the subroutine, you can pass data to a subroutine by value. In this example, the factorial of a number NUM is calculated. The input parameter NUM is passed to the formal parameter F_NUM of the subroutine. Although F_NUM is changed in the subroutine, the actual parameter NUM keeps its old value. The output parameter FAC is passed by reference.

Example of output parameters

```
PROGRAM FORM_TEST.

DATA: OP1  TYPE I,
      OP2  TYPE I,
      RES  TYPE I.

OP1 = 3.
OP2 = 4.

PERFORM MULTIP
         USING OP1 OP2
         CHANGING RES.

WRITE: / 'After subroutine:',
       / 'RES=' UNDER 'RES=', RES.

FORM MULTIP
       USING VALUE(O1)
             VALUE(O2)
     CHANGING VALUE(R).

    R = O1 * O2.
    WRITE: / 'Inside subroutine:',
           /  'R=', R, 'RES=', RES.

ENDFORM.
```

Examples of Subroutines

The produces the following output:

```
Inside subroutine:
R=        12 RES=         0
After subroutine:
          RES=        12
```

To return a changed formal parameter once the subroutine has finished successfully, you can use a CHANGING parameter and pass the parameter by reference. In this example, the actual parameters OP1 and OP2 are passed by value in the USING addition to the formal parameters O1 and O2. The actual parameter RES is passed by value to the formal parameter R using CHANGING. By writing R and RES onto the screen from within the subroutine, it is demonstrated that RES has not changed its contents before the ENDFORM statement. After returning from the subroutine, its contents have changed.

Example of passing structures

```
PROGRAM FORM_TEST.

TYPES: BEGIN OF LINE,
         NAME(10)   TYPE C,
         AGE(2)     TYPE N,
         COUNTRY(3) TYPE C,
       END OF LINE.

DATA WHO TYPE LINE.

WHO-NAME = 'Karl'. WHO-AGE = '10'. WHO-COUNTRY = 'D'.

PERFORM COMPONENTS CHANGING WHO.

WRITE: / WHO-NAME, WHO-AGE, WHO-COUNTRY.

FORM COMPONENTS
     CHANGING VALUE(PERSON) TYPE LINE.

  WRITE: / PERSON-NAME, PERSON-AGE, PERSON-COUNTRY.
  PERSON-NAME = 'Mickey'.
  PERSON-AGE = '60'.
  PERSON-COUNTRY = 'USA'.

ENDFORM.
```

The produces the following output:

```
Karl       10 D

MICKEY     60 USA
```

The actual parameter WHO with the user-defined, structured data type LINE is passed to the formal parameter PERSON. The formal parameter PERSON is typed with TYPE LINE. Since LINE is a user-defined data type, the type of PERSON is completely specified. The subroutine accesses and changes the components of PERSON. They are then returned to the components of WHO in the calling program.

Example of passing internal tables

```
PROGRAM FORM_TEST.

DATA: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE STANDARD TABLE OF LINE.

PERFORM FILL CHANGING ITAB.

PERFORM OUT  USING    ITAB.

FORM FILL CHANGING F_ITAB LIKE ITAB.

  DATA F_LINE LIKE LINE OF F_ITAB.

  DO 3 TIMES.
    F_LINE-COL1 = SY-INDEX.
    F_LINE-COL2 = SY-INDEX ** 2.
    APPEND F_LINE TO F_ITAB.
  ENDDO.

ENDFORM.

FORM OUT USING VALUE(F_ITAB) LIKE ITAB.

  DATA F_LINE LIKE LINE OF F_ITAB.

  LOOP AT F_ITAB INTO F_LINE.
    WRITE: / F_LINE-COL1, F_LINE-COL2.
  ENDLOOP.

ENDFORM.
```

The produces the following output:

```
1       1

2       4

3       9
```

You can define the types of the formal parameters of the parameter interfaces of procedures as internal tables. In the example, the subroutines FILL and OUT each have one formal parameter defined as an internal table. An internal table without header line is passed to the subroutines. Each subroutine declares a work area F_LINE as a local data object. Were ITAB a table with a header line, you would have to replace ITAB with ITAB[] in the PERFORM and FORM statements.

Example of the TABLES parameter

This example is provided for completeness. The TABLES parameter is only supported for the sake of compatibility and should not be used.

```
PROGRAM FORM_TEST.
```

Examples of Subroutines

```
TYPES: BEGIN OF LINE,
         COL1 TYPE I,
         COL2 TYPE I,
       END OF LINE.

DATA: ITAB TYPE STANDARD TABLE OF LINE WITH HEADER LINE,
      JTAB TYPE STANDARD TABLE OF LINE.

PERFORM FILL TABLES ITAB.

MOVE ITAB[] TO JTAB.

PERFORM OUT TABLES JTAB.

FORM FILL TABLES F_ITAB LIKE ITAB[].

  DO 3 TIMES.
    F_ITAB-COL1 = SY-INDEX.
    F_ITAB-COL2 = SY-INDEX ** 2.
    APPEND F_ITAB.
  ENDDO.

ENDFORM.

FORM OUT TABLES F_ITAB LIKE JTAB.

  LOOP AT F_ITAB.
    WRITE: / F_ITAB-COL1, F_ITAB-COL2.
  ENDLOOP.

ENDFORM.
```

The produces the following output:

```
1       1

2       4

3       9
```

In this example, an internal table ITAB is declared with a header line and an internal table JTAB is declared without a header line. The actual parameter ITAB is passed to the formal parameter F_ITAB of the subroutine FILL in the TABLES addition. The header line is passed with it. After the body of the table has been copied from ITAB to JTAB, the actual parameter is passed to the formal parameter F_ITAB of the subroutine OUT using the TABLES addition. The header line F_ITAB, which is not passed, is generated automatically in the subroutine.

Shared Data Areas

For the sake of completeness, this section describes a technique that allows you to access the global data of the calling program from an external subroutine. To do this, you declare a common data area for the calling program and the program containing the subroutine. All interface work areas [Page 130] declared using TABLES and NODES behave like the common data area.

Like external subroutines themselves, you should use common data areas very sparingly. The rules [Page 494] for accessing common data areas can become very complicated if you call function modules from nested subroutines.

You declare a common data area in all programs concerned using the statement:

```
DATA: BEGIN OF COMMON PART [<name>],

    END OF COMMON PART [<name>].
```

Between the two DATA statements, you declare all of the data you want to include in the common data area.

The common part declaration must be exactly the same in all of the programs in which you want to use it. It is therefore a good idea to put the declaration in an include program.

You can use several common parts in one program. In this case, you must assign a name <name> to each common part. If you only have one common part in each program, you do not have to specify a name. To avoid conflicts between programs that have different common part declarations, you should always assign unique names to common parts.

Assume an include program INCOMMON contains the declaration of a common part NUMBERS. The common part comprises three numeric fields: NUM1, NUM2, and SUM:

```
***INCLUDE INCOMMON.

DATA: BEGIN OF COMMON PART NUMBERS,
        NUM1 TYPE I,
        NUM2 TYPE I,
        SUM  TYPE I,
      END OF COMMON PART NUMBERS.
```

The program FORMPOOL includes INCOMMON and contains the subroutines ADDIT and OUT:

```
PROGRAM FORMPOOL.

INCLUDE INCOMMON.

FORM ADDIT.
  SUM = NUM1 + NUM2.
  PERFORM OUT.
ENDFORM.

FORM OUT.
  WRITE: / 'Sum of', NUM1, 'and', NUM2, 'is', SUM.
ENDFORM.
```

A calling program FORM_TEST includes INCOMMON and calls the subroutine ADDIT from the program FORMPOOL.

```
PROGRAM FORM_TEST.

INCLUDE INCOMMON.

NUM1 = 2. NUM2 = 4.
PERFORM ADDIT(FORMPOOL).

NUM1 = 7. NUM2 = 11.
PERFORM ADDIT(FORMPOOL).
```

This produces the following output:

```
Sum of           2 and           4 is           6

Sum of           7 and          11 is          18
```

The subroutines in the program FORMPOOL access the global data of the shared data area.

Assume a program FORMPOOL that contains two subroutines TABTEST1 and TABTEST2 as follows:

```
PROGRAM FORMPOOL.

TABLES SFLIGHT.

FORM TABTEST1.
  SFLIGHT-PLANETYPE = 'A310'.
  SFLIGHT-PRICE = '150.00'.
  WRITE: / SFLIGHT-PLANETYPE, SFLIGHT-PRICE.
ENDFORM.

FORM TABTEST2.
  LOCAL SFLIGHT.
  SFLIGHT-PLANETYPE = 'B747'.
  SFLIGHT-PRICE = '500.00'.
  WRITE: / SFLIGHT-PLANETYPE, SFLIGHT-PRICE.
ENDFORM.
```

A program FORM_TEST calls TABTEST1 and TABTEST2 as follows:

```
PROGRAM FORM_TEST.

TABLES SFLIGHT.

PERFORM TABTEST1(FORMPOOL).
WRITE: / SFLIGHT-PLANETYPE, SFLIGHT-PRICE.

PERFORM TABTEST2(FORMPOOL).
WRITE: / SFLIGHT-PLANETYPE, SFLIGHT-PRICE.
```

SAPMZTST then produces the following output:

```
A310                 150.00

A310                 150.00

B747                 500.00
```

```
A310                150.00
```

All of the programs can access the table work area SFLIGHT, declared with the TABLES statement. In the subroutine TABTEST2, the global work area is protected against changes in the subroutine by the LOCAL statement.

Function Modules

Function modules are procedures that are defined in function groups (special ABAP programs with type F) and can be called from any ABAP program. Function groups act as containers for function modules that logically belong together. You create function groups and function modules in the ABAP Workbench using the Function Builder [Ext.].

Function modules allow you to encapsulate and reuse global **functions** in the R/3 System. They are stored in a central library. The R/3 System contains a wide range of predefined function modules that you can call from any ABAP program. Function modules also play an important role in database updates [Page 1276] and in remote communications [Ext.] between R/3 Systems or between an R/3 System and a non-SAP system.

Unlike subroutines, you do not define function modules in the source code of your program. Instead, you use the Function Builder [Ext.]. The actual ABAP interface definition remains hidden from the programmer. You can define the input parameters of a function module as optional. You can also assign default values to them. Function modules also support exception handling. This allows you to catch certain errors while the function module is running. You can test function modules without having to include them in a program using the Function Builder [Ext.].

The Function Builder [Ext.] also has a release process for function modules. This ensures that incompatible changes cannot be made to any function modules that have already been released. This applies particularly to the interface. Programs that use a released function module will not cease to work if the function module is changed.

Function Groups [Page 481]

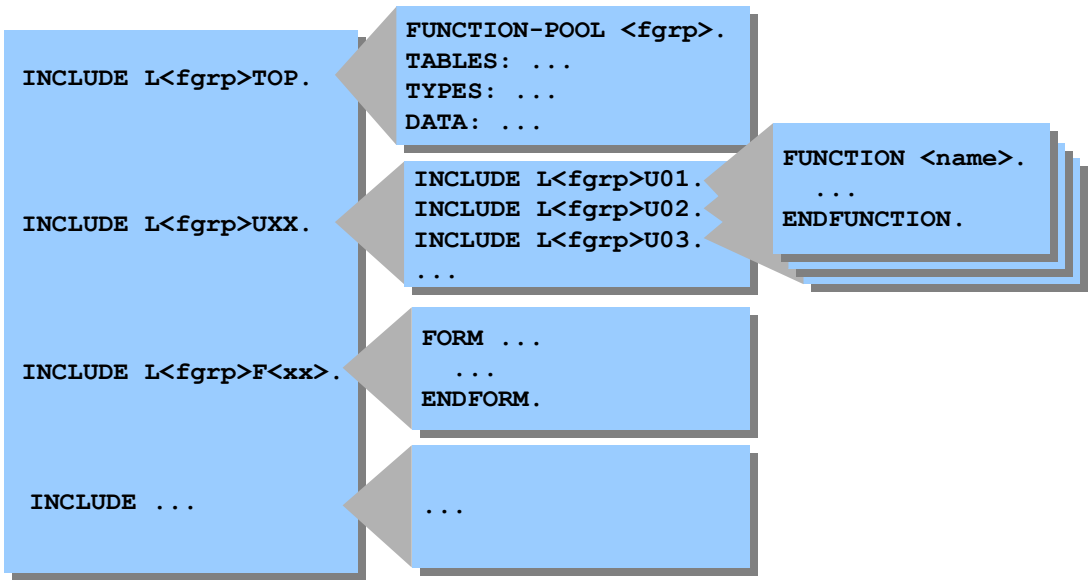
Calling Function Modules [Page 483]

Creating Function Modules [Page 488]

Function Groups

Function groups are containers for function modules. You cannot execute a function group. When you call an function module, the system loads the whole of its function group into the internal session of the calling program (if it has not already been loaded). For further information, refer to Organization of External Procedure Calls [Page 494].

The following diagram shows the structure of a function group: The name of a function group can be up to 26 characters long. This is used by the system to create the components of the group (main program and corresponding include programs). When you create a function group or function module in the Function Builder [Ext.] , the main program and include programs are generated automatically.



Main program SAPL<fgrp> Include programs

Th
e main program SAPL<fgrp> contains nothing but the INCLUDE statements for the following include programs:

- L<fgrp>TOP. This contains the FUNCTION-POOL statement (equivalent for a function group of the REPORT or PROGRAM statement) and global data declarations for the entire function group.
- L<fgrp>UXX. This contains further INCLUDE statements for the include programs L<fgrp>U01, L<fgrp>U02, ... These includes contain the actual function modules.
- The include programs L<fgrp>F01, L<fgrp>F02, ... can contain the coding of subroutines that can be called with internal subroutine calls from all function modules of the group.

The creation of these include programs is supported from the ABAP Workbench by forward navigation (for example creation of a subroutine include by double clicking on the name of a subroutine in a PERFORM statement within a function module).

Function Groups

You cannot declare a COMMON PART in a function group. Function groups have their own table work areas (TABLES). Function groups encapsulate data. In this respect, they are a precursor of ABAP Objects (see From Function Groups to Objects [Page 1296]).

All of the function modules in a function group can access the global data of the group. For this reason, you should place all function modules that use the same data in a single function group. For example, if you have a set of function modules that all use the same internal table, you could place them in a function group containing the table definition in its global data.

Like executable programs (type 1) and module pools (type M), function groups can contain screens, selection screens, and lists. User input is processed either in dialog modules or in the corresponding event blocks in the main program of the function group. There are special include programs in which you can write this code. In this way, you can use function groups to encapsulate single screens or screen sequences.

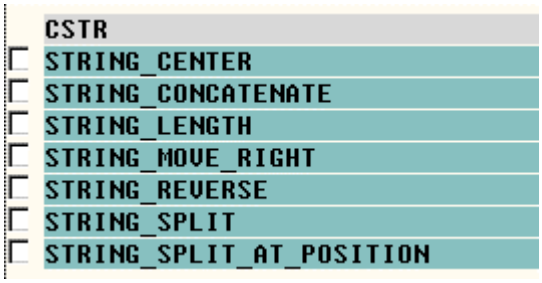
Calling Function Modules

This section describes calling function modules from the Function Builder.

Finding Function Modules

Before programming a new function or creating a new function module, you should look in the Function Builder to see whether there is an existing function module that already performs the same task.

For more information about this, refer to Finding Function Modules [Ext.] in the ABAP Workbench documentation. For example, you might look for function modules that process strings by entering *STRING* as a search criterion in the Repository Information System. This is an extract from the list of function modules found:



The title CSTR is the function group. There is a main program SAPLCSTR that contains these function modules. If you select a function module, you can display its attributes in the Function Builder.

Important attributes:

- Documentation

  The documentation describes the purpose of the function module, lists the parameters for passing data to and from the module, and the exceptions. It tells you how you can pass data to and from the function module, and which errors it handles.

- Interface parameters and exceptions

  This section provides further information about the interface parameters and exceptions, and how to use the function module. For further information, refer to Displaying Information about Interface Parameters [Ext.] in the ABAP Workbench documentation. Function modules can have the following interface parameters:

- Import parameters. These must be supplied with data when you call the function module, unless they are flagged as optional. You cannot change them in the function module.
- Export parameters. These pass data from the function module back to the calling program. Export parameters are always optional. You do not have to receive them in your program.
- Changing parameters. These must be supplied with data when you call the function module, unless they are flagged as optional. They can be changed in the function module. The changed values are then returned to the calling program.
- Tables parameters. You use these to pass internal tables. They are treated like CHANGING parameters. However, you can also pass internal tables with other parameters if you specify the parameter type appropriately.

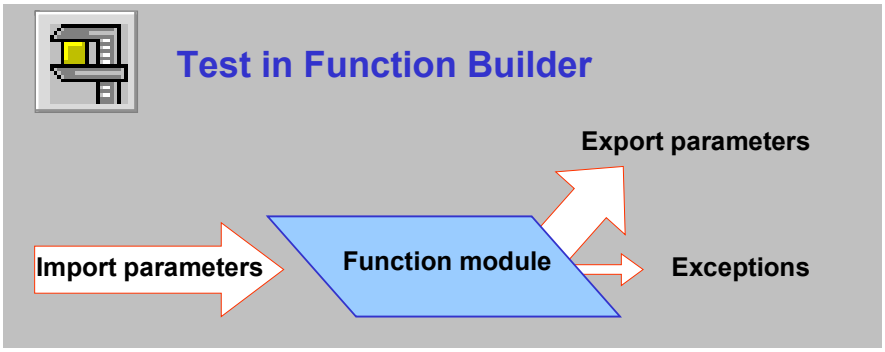
You can specify the types of the interface parameters, either by referring to ABAP Dictionary types or elementary ABAP types. When you call a function module, you must ensure that the actual parameter and the interface parameters are compatible.

Interface parameters are, by default, passed by value. However, they can also be passed by reference. Tables parameters can only be passed by reference. You can assign default values to optional importing and changing parameters. If an optional parameter is not passed in a function module call, it either has an initial value, or is set to the default value.

Exceptions are used to handle errors that occur in function modules. The calling program checks whether any errors have occurred and then takes action accordingly.

Testing Function Modules

Before you include a function module in a program, you can test it in the Function Builder.



For more information about this, refer to Testing Function Modules [Ext.] in the ABAP Workbench documentation.

Calling Function Modules in ABAP

To call a function module, use the CALL FUNCTION statement:

CALL FUNCTION <module>
[EXPORTING $f_1 = a_1 .... f_n = a_n$]
[IMPORTING $f_1 = a_1 .... f_n = a_n$]
[CHANGING $f_1 = a_1 .... f_n = a_n$]
[TABLES $f_1 = a_1 .... f_n = a_n$]
[EXCEPTIONS $e_1 = r_1 .... e_n = r_n$ [ERROR_MESSAGE = r_E]
[OTHERS = r_o]].

You can specify the name of the function module <module> either as a literal or a variable. Each interface parameter <f_i> is explicitly assigned to an actual parameter <a_i>. You can assign a return value <r_i> to each exception <e_i>. The assignment always takes the form <interface parameter> = <actual parameter>. The equals sign is not an assignment operator in this context.

- After EXPORTING, you must supply all non-optional import parameters with values appropriate to their type. You can supply values to optional import parameters if you wish.
- After IMPORTING, you can receive the export parameters from the function module by assigning them to variables of the appropriate type.

- After CHANGING or TABLES, you must supply values to all of the non-optional changing or tables parameters. When the function module has finished running, the changed values are passed back to the actual parameters. You can supply values to optional changing or tables parameters if you wish.

You can use the EXCEPTIONS option to handle the exceptions of the function module. If an exception <e_i> is raised while the function module is running, the system terminates the function module and does not pass any values from the function module to the program, except those that were passed by reference. If <e_i> is specified in the EXCEPTION option, the calling program handles the exception by assigning <r_i> to SY-SUBRC. <r_i> must be a numeric literal.

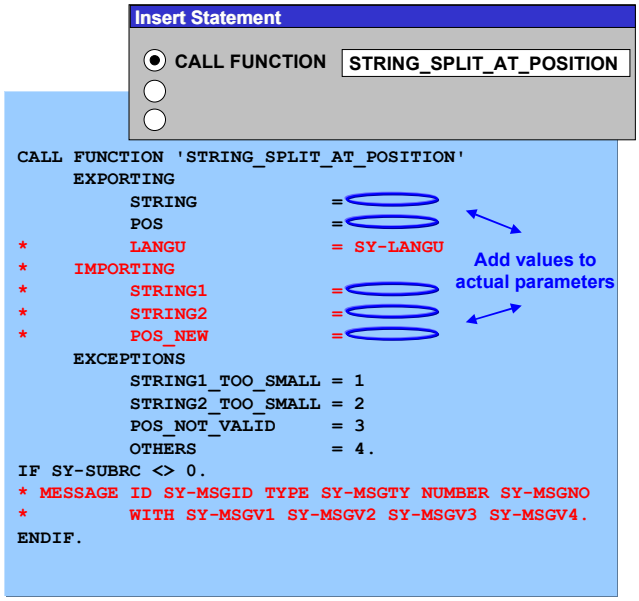
If you specify of ERROR_MESSAGE in the exception list you can influence the message handling of function modules. Normally, you should only call messages in function modules using the MESSAGE ... RAISING statement. With ERROR_MESSAGE you can force the system to treat messages that are called without the RAISING option in a function module as follows:

- Messages of classes S, I, and W are ignored (but written to the log in a background job).
- Messages of classes E and A stop the function module as if the exception ERROR_MESSAGE had occurred (SY-SUBRC is set to <r_E>).

If you specify OTHERS after EXCEPTIONS, the system assigns a single return code to all other exceptions that you have not specified explicitly in the list.

You can use the same number <r_i> for several exceptions.

The recommended and easiest way to call a function module is to use the *Insert statement* function in the ABAP Editor. If you select Call Function and specify the name of the function module (F4 help is available), the system inserts a CALL FUNCTION statements with all of the options of that function module in the source code.




```
CALL FUNCTION 'STRING_SPLIT_AT_POSITION'
     EXPORTING
          STRING             =
          POS                =
*         LANGU              = SY-LANGU
*    IMPORTING
*         STRING1            =
*         STRING2            =
*         POS_NEW            =
     EXCEPTIONS
          STRING1_TOO_SMALL = 1
          STRING2_TOO_SMALL = 2
          POS_NOT_VALID     = 3
          OTHERS            = 4.
IF SY-SUBRC <> 0.
* MESSAGE ID SY-MSGID TYPE SY-MSGTY NUMBER SY-MSGNO
*         WITH SY-MSGV1 SY-MSGV2 SY-MSGV3 SY-MSGV4.
ENDIF.
```

Optional parts of the function call are inserted as comments. In the above example, STRING and POS are obligatory parameters. LANGU, on the other

hand, is optional. It has the default value SY-LANGU (the system field for the logon language). Handling export parameters is optional. Handling exceptions is also theoretically optional. However, you should always do so. That is why the EXCEPTIONS lines are not commented out.

You can trigger exceptions in the function module using either the RAISE or the MESSAGE ... RAISING statement. If the calling program handles the exception, both statements return control to the program. The MESSAGE ..... RAISING statement does not display a message in this case. Instead, it sets the following system fields:

- Message class → SY-MSGID
- Message type → SY-MSGTY
- Message number → SY-MSGNO
- SY-MSGV1 to SY-MSGV4 (contents of fields <f1> to <f4>, included in a message).

You can use the system fields to trigger the message from the calling program.

To ensure that you use the right data types for the actual parameters, you must refer to the function module interface. If you double-click the name of a function module in the source code of your program, the system navigates to its source code in the Function Builder. You can then display the interface by choosing *Goto → Interface*.

For example, in the above case

- STRING, STRING1, and STRING2 have the generic type C. The actual parameter must also have type C, but the length does not matter.
- POS and POS_NEW have the fully-specified type I. The actual parameter must also have type I.
- LANGU also has a fully-defined type, since it is defined with reference to the ABAP Dictionary field SY-LANGU. The actual parameter must have the same type.

A complete call for the function module STRING_SPLIT_AT_POSITION might look like this:

```
PROGRAM CALL_FUNCTION.

DATA: TEXT(10) TYPE C VALUE '0123456789',
      TEXT1(6) TYPE C,
      TEXT2(6) TYPE C.

PARAMETERS POSITION TYPE I.

CALL FUNCTION 'STRING_SPLIT_AT_POSITION'
     EXPORTING
          STRING            =  TEXT
          POS               =  POSITION
     IMPORTING
          STRING1           =  TEXT1
          STRING2           =  TEXT2
     EXCEPTIONS
          STRING1_TOO_SMALL = 1
          STRING2_TOO_SMALL = 2
```

```
          POS_NOT_VALID     = 3
          OTHERS            = 4.

CASE SY-SUBRC.
  WHEN 0.
    WRITE: / TEXT, / TEXT1, / TEXT2.
  WHEN 1.
    WRITE 'Target field 1 too short!'.
  WHEN 2.
    WRITE 'Target field 2 too short!'.
  WHEN 3.
    WRITE 'Invalid split position!'.
  WHEN 4.
    WRITE 'Other errors!'.
ENDCASE.
```

The function module splits an input field at a particular position into two output fields. If the contents of POSITION are in the interval [4,6], the function module is executed without an exception being triggered. For the intervals [1,3] and [7,9], the system triggers the exceptions STRING2_TOO_SMALL and STRING2_TOO_SMALL respectively. For all other values of POSITION, the exception POS_NOT_VALID is triggered.

Creating Function Modules

You can only create function modules and function groups using the Function Builder in the ABAP Workbench. For further information, refer to Creating New Function Modules [Ext.]. This section uses an example to illustrate how a function module is created from the point of view of ABAP programming.

We are going to create a function module READ_SPFLI_INTO_TABLE to read data for a specified airline from table SPFLI into an internal table, which it then passes back to the calling program.

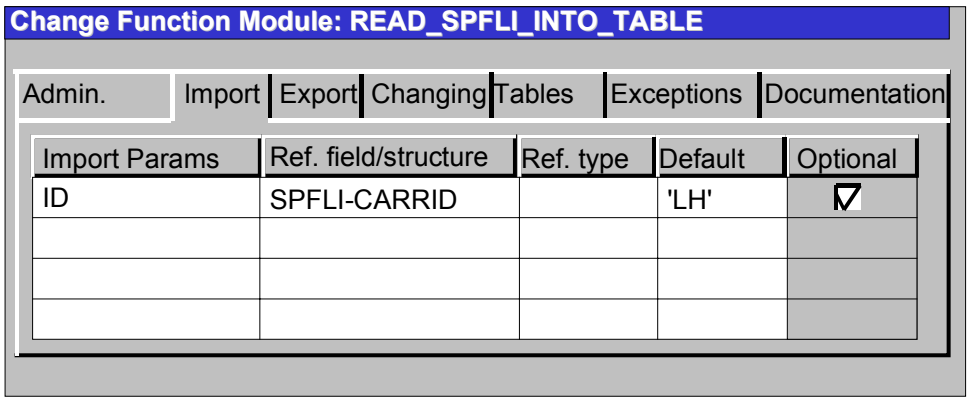
Function Groups and Function Modules

Firstly, we create a new function group DEMO_SPFLI to hold the function module (see Creating a Function Group [Ext.]). Then, we can create the new function module (see Creating a Function Module [Ext.]).

Parameter Interface

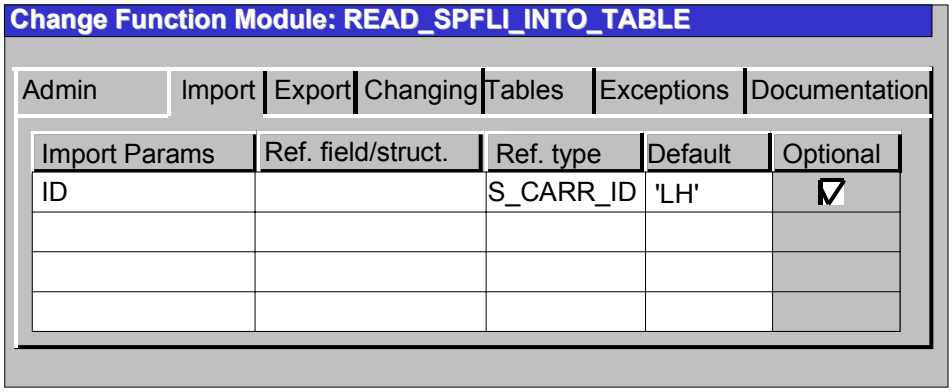
You can specify the types of interface parameters in function modules in the same way as the parameter interfaces [Page 459] of subroutines. Since function modules can be used anywhere in the system, their interfaces can only contain references to data types that are declared systemwide. These are the elementary ABAP data types, the systemwide generic types, such as ANY TABLE, and types defined in the ABAP Dictionary. You cannot use LIKE to refer to data types declared in the main program.

The function module READ_SPFLI_INTO_TABLE requires an import parameter to restrict the selection to a single airline. To specify the type, we can refer to the key field CARRID of the database SPFLI:



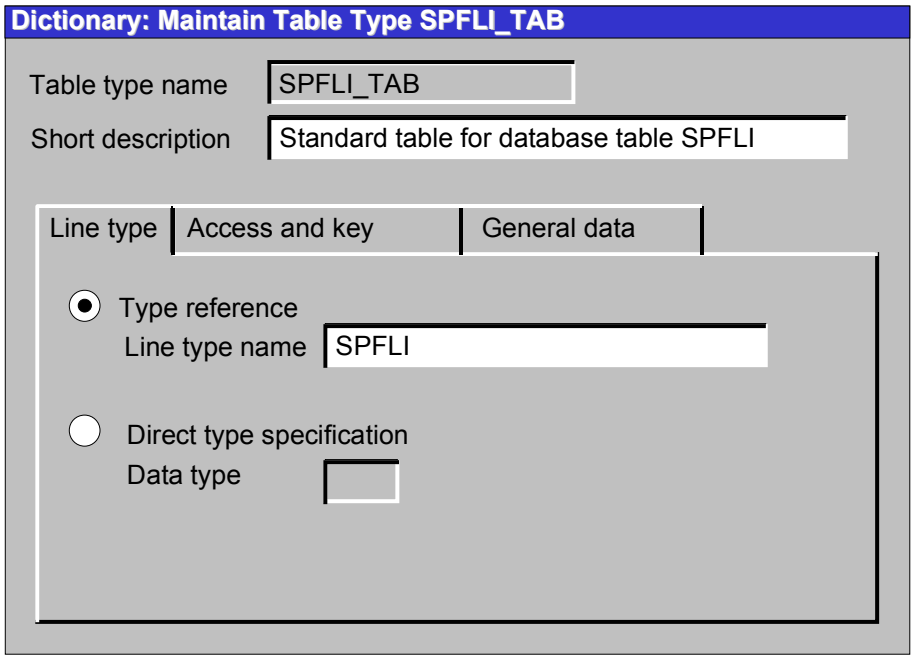
Under *Ref. field/structure*, you can enter a column of a database table, a component of a ABAP Dictionary structure, or a whole ABAP Dictionary structure. The import parameter ID is optional, and has a default value.

The following type specification would have the same effect:

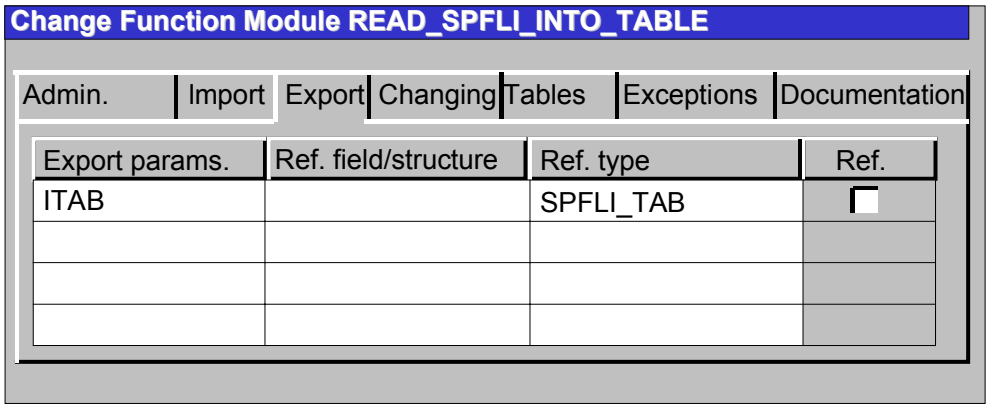


Ref. type can contain any generic or full data type that is recognized systemwide. Here, the parameter is defined with reference to the elementary ABAP Dictionary type (or data element) S_CARR_ID. This is the type used to define the field SPFLI-CARRID.

To pass data back to the calling program, the function module needs an export parameter with the type of an internal table. For this, we define a systemwide table type SPFLI_TAB with the line type SPFLI in the ABAP Dictionary.



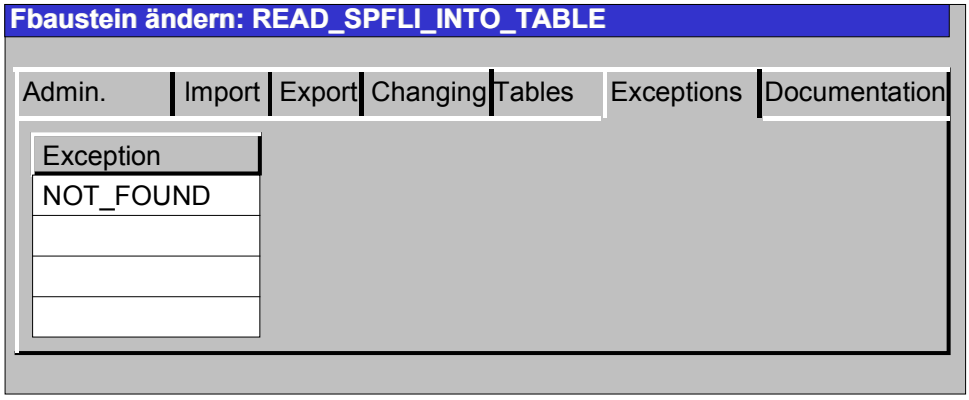
We can now use this data type to specify the type of the export parameter ITAB:



The internal table is passed by value. Only internal tables that are passed using tables parameters can be passed exclusively by reference.

Exceptions

Our function module needs an exception that it can trigger if there are no entries in table SPFLI that meet the selection criterion. The exception NOT_FOUND serves this function:



Source Code

Having defined the parameter interface and exceptions, we can now write the source code of our function module. To do this, choose *Source code* in the Function Builder. This opens the ABAP Editor for the include program L<fgrp>U<xx> (see Function Groups [Page 481]). This is the include that will hold the program code for the function module;

Change F Module: READ_SPFLI_INTO_TABLE/LDEMO_SPFLI_U01

```
FUNCTION READ_SPFLI_INTO_TABLE.
*"----------------------------------------------------------
*"*"Local interface:
*"       IMPORTING
*"           VALUE(ID) LIKE  SPFLI-CARRID DEFAULT 'LH '
*"       EXPORTING
*"           VALUE(ITAB) TYPE  SPFLI_TAB
*"       EXCEPTIONS
*"           NOT_FOUND
*"----------------------------------------------------------


ENDFUNCTION.
```

The source code of the function module occurs between the FUNCTION and ENDFUNCTION statements. The definitions of the parameter interface and the exceptions is displayed here in comment lines. Its real coding is generated invisibly by the Function Builder.

Data in Function Modules

You can use the TYPES and DATA statements to create local data types and objects. The interface parameters also behave like local data objects. In addition, you can access all of the global data of the main program. This data is defined in the include program L<fgrp>TOP. To open this include, choose *Goto → Global data*. The global data behaves like the instance attributes of a class. The first time you call a function module in a particular function group, the data is loaded into memory. It can then be accessed and changed by all of the function modules in the group. The system retains the values until the next time a function module is called.

Calling Subroutines

You use subroutines [Page 451] for local modularization. Function modules can also use this technique. The function module that they call are defined in the corresponding main program.

If you only want to call a subroutine from a single function module, it is best to define them in the same include program as the function module itself, directly after the ENDFUNCTION statement. These subroutines can be called from all function modules in the function group, but for clarity, they should only be called from the function module that precedes them.

If you want to define a subroutine that will be called from several different function modules, you can define a special include program for it with the name L<fgrp>F<xx>.

Raising Exceptions

There are two ABAP statements for raising exceptions. They can only be used in function modules:

RAISE <except>.

and

MESSAGE..... RAISING <except>.

The effect of these statements depends on whether the calling program handles the exception or not. If the name <except> of the exception or OTHERS occurs in the EXCEPTIONS addition of the CALL FUNCTION statement, the exception is handled by the calling program.

If the calling program does not handle the exception

- The RAISE statement terminates the program and switches to debugging mode.
- The MESSAGE ..... RAISING statement display the specified message. How the processing continues depends on the message type.

If the calling program handles the exception, both statements return control to the program. No values are transferred. The MESSAGE ..... RAISING statement does not display a message. Instead, it fills the system fields SY-MSGID, SY-MSGTY, SY-MSGNO, and SY-MSGV1 to SY-MSGV4.

Source Code of READ_SPFLI_INTO_TABLE

The entire source code of READ_SPFLI_INTO_TABLE looks like this:

```
FUNCTION READ_SPFLI_INTO_TABLE.
*"----------------------------------------------------------------
--
*"*"Local interface:
*"       IMPORTING
*"             VALUE(ID) LIKE  SPFLI-CARRID DEFAULT 'LH '
*"       EXPORTING
*"             VALUE(ITAB) TYPE  SPFLI_TAB
*"       EXCEPTIONS
*"              NOT_FOUND
*"----------------------------------------------------------------
--

  SELECT * FROM SPFLI INTO TABLE ITAB WHERE CARRID = ID.

  IF SY-SUBRC NE 0.
    MESSAGE E007(AT) RAISING NOT_FOUND.
  ENDIF.

ENDFUNCTION.
```

The function module reads all of the data from the database table SPFLI where the key field CARRID is equal to the import parameter ID and places the entries that it finds into the internal table SPFLI_TAB. If it cannot find any entries, the exception NOT_FOUND is triggered using MESSAGE...RAISING. Otherwise, the table is passed to the caller as an exporting parameter.

Calling READ_SPFLI_INTO_TABLE

The following program calls the function module READ_SPFLI_INTO_TABLE:

```
REPORT DEMO_FUNCTION_MODULE.

PARAMETERS CARRIER TYPE S_CARR_ID.

DATA: JTAB TYPE SPFLI_TAB,
      WA   LIKE LINE OF JTAB.

CALL FUNCTION 'READ_SPFLI_INTO_TABLE'
     EXPORTING
          ID        = CARRIER
     IMPORTING
          ITAB      = JTAB
     EXCEPTIONS
          NOT_FOUND = 1
          OTHERS    = 2.

CASE SY-SUBRC.
  WHEN 1.
    MESSAGE ID SY-MSGID TYPE SY-MSGTY NUMBER SY-MSGNO.
  WHEN 2.
    MESSAGE E702(AT).
ENDCASE.

LOOP AT JTAB INTO WA.
  WRITE: /  WA-CARRID, WA-CONNID, WA-CITYFROM, WA-CITYTO.
ENDLOOP.
```

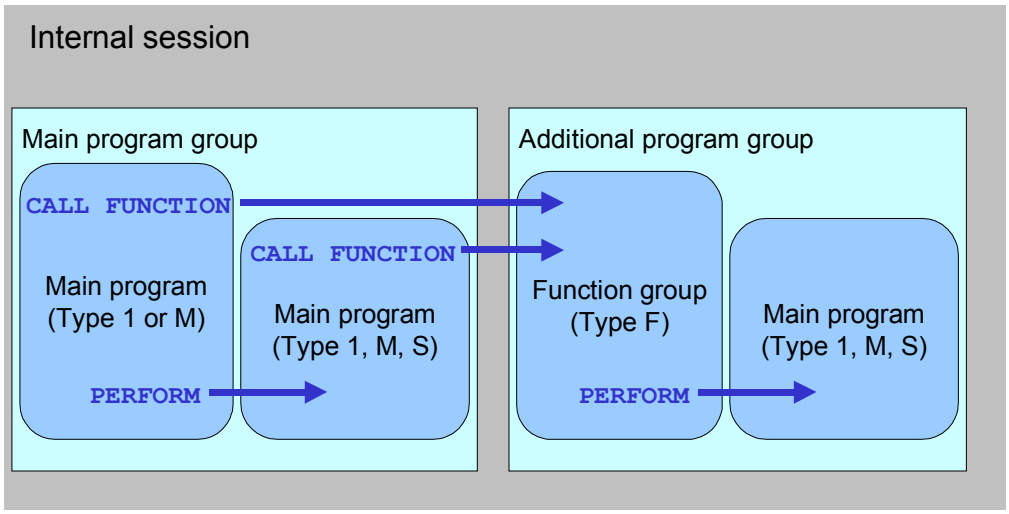
The actual parameters CARRIER and JTAB have the same data types as their corresponding interface parameters in the function module. The exception NOT_FOUND is handled in the program. It displays the same message that the function module would have displayed had it handled the error.

Organization of External Procedure Calls

Each time you start or call a program with type 1 or M, a new internal session [Page 66] is opened in the current R/3 terminal session. When the program calls external procedures, their main program and working data are also loaded into the memory area of the internal session. Here, there is a difference between calling external subroutines, function modules, and working with classes in ABAP Objects.

Program Groups in an Internal Session

This illustration shows the memory organization for external subroutines and function modules.



The main program (type 1 or M) belongs to a main program group. When you call a function module from a function group that has not already been loaded, an additional program group is created, into which the function group to which the function module belongs is loaded. The additional program group, along with its data, exists for the rest of the lifetime of the internal session.

If you call an external subroutine using PERFORM, the main program of the subroutine is loaded into the existing main program or function group in the corresponding main or additional program group (if it has not already been loaded).

An additional program group is only opened in an external PERFORM if the subroutine belongs to a function group that has not yet been loaded. This will happen if the main program of the subroutine begins with the FUNCTION-POOL statement. The program type does not necessarily have to be set to F.

Classes in ABAP Objects behave like function groups. The first time you address a class (for example, using the CREATE OBJECT statement) or address a static class component, the system opens an additional program group for the class.

Screen Handling

The system only displays and processes the screens, selection screens, and lists of the main program and main program group or function group of an additional program group, along with their associated GUI statuses. The CALL SCREEN and CALL SELECTION-SCREEN statements or list display statements in external subroutines apply to the screens of the main program or function group. The screens are processed in the main program or function group. So, for example, even if the main program of an external subroutine has a screen 100, the CALL SCREEN 100 statement will call the screen number 100 in the **calling main program**.

Interface Work Areas

Interface work areas [Page 130] that you declare using the TABLES, NODES, or DATA BEGIN OF COMMON PART statements exist once only in each program group, and are shared by all of its programs. Each main program that begins with the PROGRAM or REPORT statement, and each function group that begins with FUNCTION-POOL shares the interface work areas with the main programs of all of its external subroutines.

Dynamic Assignment

The assignment of screens and interface work areas to subroutines is defined by the **sequence** of the subroutine calls. The first program to call an external subroutine shares its interface work areas with the subroutine, and the subroutine uses its screens. The sequence of the subroutine calls is not necessarily statically defined. Instead, it can change dynamically depending on user actions or the contents of fields. For example, a subroutine may belong to a main program group on occasion when it is run, but the next time, it may belong to an additional program group. For this reason, you should avoid using external subroutines if they use interface work areas or call screens.

Special Techniques

Catchable Runtime Errors

Runtime errors occur when the ABAP runtime environment discovers an error at runtime that could not be detected while the program was being created. A runtime error may be catchable or non-catchable. This section contains an overview of the different ABAP program checks and a description of how to handle catchable runtime errors.

Program Checks [Page 498]

Catching Runtime Errors [Page 500]

Program Checks

The system checks ABAP programs statically when you create them and dynamically when they are running.

Static Checks

In the static checks, the syntax check tests the syntax of the program, and the extended program check performs a semantic check (including, for example, checking whether the number and type of the interface parameters in an external procedure call is correct).

Syntax Check

The syntax check checks the syntax and internal semantics of a program. When an error occurs, the check stops, and the system generates an appropriate error message. A dialog box often appears, in which a proposed solution is displayed. Choose "Correct" to adopt this proposed solution in the program. The system then repeats the syntax check.

When you run a syntax check on an include program, the system combines it with the TOP include of the program before checking it. The system only checks all of the includes belonging to a program for consistency and completeness (for example, ensuring that all of the subroutines called from the program have actually been defined) when you run the syntax check for the main program.

The syntax check may generate warnings. In this case, the syntax check does not end immediately, and you can, technically, run the program. However, you should always correct syntax warnings, since they could cause unforeseen errors when you run the program. Furthermore, if SAP tightens aspects of ABAP syntax in a future release, any warning could become a syntax error.

Extended Program Check

Many checks are excluded from the standard syntax check for performance reasons. The extended program check performs a complete check that includes the interfaces of external procedures called from your program.

Errors in the extended program check cause runtime errors when you run the program. You **must** correct them. The exception to this is coding that cannot be reached. However, you should delete this to minimize the size of your program and make the source code easier to understand.

Warnings in the extended program check should also be corrected. If your program contains statements that are definitely correct but still produce warnings in the extended program check, you can exclude them from the check using pseudocomments (`"#EC`…).

You should always run the extended program check on a new program. You have not finished developing a new program until you can run the extended program check on it without any errors or warnings. Messages are permissible, since they are generally not critical.

The extended program check is also only a static check. It cannot eliminate all of the circumstances that could lead to runtime errors. For example, any statements in which you specify arguments dynamically as the contents of fields, or in which you call procedures dynamically, cannot be checked statically.

Dynamic Check

Errors that cannot be detected statically and that occur at runtime are detected by the ABAP runtime environment. These are called **runtime errors**. If you do not catch [Page 500] a runtime error, the runtime environment terminates the program, generates a short dump, and switches to a special screen from which you can analyze it.

Short dumps are divided into blocks describing different aspects of the error. In the overview, you can see what information is contained in the short dump, for example, the contents of data objects, active calls, control structures, and so on. The short dump overview also allows you to display the point in the program at which the termination occurred in the ABAP Debugger.

The following kinds of errors can occur:

- Internal error

  The kernel encountered an error. In this case, inform SAP.

- Installation and environment or resource error

  An error that occurred due to incorrect system installation or missing resources (for example, database closed).

- Error in application program

  Typical causes:

  Contents of a numeric field have the wrong format.

  Arithmetic overflow.

  External procedure does not exist.

  Type conflict when passing parameters to an external procedure.

Short dumps are retained in the system for 14 days. Use Transaction ST22 to administer short dumps. If you want to retain a short dump beyond the default 14 days, use the *Keep* function. If you want to delete short dumps by a date criterion other than the default 14 days, choose *Reorganize*. If you display a short dump using Transaction ST22, you cannot branch to the Debugger.

If problems occur in an ABAP program that you cannot solve yourself, you can send an extract of the short dump to SAP. The contents of the short dump are essential to the SAP Hotline and Remote Consulting to enable them to solve your problem.

Catching Runtime Errors

Some runtime errors are caused by errors that make it impossible to permit a program to continue. However, some are not so serious, and these can be handled within the program. Runtime errors are therefore categorized as either catchable or non-catchable. The keyword documentation for each ABAP statement indicates the runtime errors that can occur in connection with the statement, and whether or not they are catchable. Catchable runtime errors are divided into ERROR classes that allow you to handle similar errors together.

You can handle catchable runtime errors in an ABAP program using the following control statements:

CATCH SYSTEM-EXCEPTIONS <exc_1> = <rc_1> ... <exc_n> = <rc_n>.
  ...
ENDCATCH.

The expressions <exc_i> indicate either a single catchable runtime error or the name of an ERROR class. The expressions <rc_i> are numeric literals. If one of the specified runtime errors occurs between CATCH and ENDCATCH, the program does not terminate. Instead, the program jumps straight to the ENDCATCH statement. After ENDCATCH, the numeric literal <rc_i> that you assigned to the runtime error is contained in the return code field SY-SUBRC. The contents of any fields that occur in the statement in which the error occurred cannot be guaranteed.

CATCH control structures are like IF structures, that is, you can nest them to any depth, but they must begin and end within the same processing block. Furthermore, CATCH control structures only catch runtime errors at the current call level, and not in any procedures that you call from within the CATCH … ENDCATCH block.

The keyword documentation for the CATCH statement contains a list of all catchable runtime errors and the way in which they are organized into ERROR classes.

```
REPORT demo_catch_endcatch.

DATA: result TYPE p DECIMALS 3,
      number TYPE i VALUE 11.

CATCH SYSTEM-EXCEPTIONS arithmetic_errors = 5.
  DO.
    number = number - 1.
    result = 1 / number.
    WRITE: / number, result.
  ENDDO.
ENDCATCH.

SKIP.

IF sy-subrc = 5.
  WRITE / 'Division by zero!'.
ENDIF.
```

This program calculates the quotient of 1 and NUMBER ten times until the catchable runtime error BCD_ZERODIVIDE occurs. This belongs to the ERROR class ARITHMETIC_ERRORS, and is caught in the example using this class.

Checking Authorizations

SQL-Anweisungen [Page 1038]

Berechtigungskonzept [Page 504]

Berechtigungsprüfungen [Page 505]

In an ABAP program, there are no automatic authorization checks associated with Open SQL statements. This can cause problems, since Open SQL and Native SQL statements allow unrestricted access to all database tables.

Not all users should have authorization to access all data that is available by using the SQL statements that they are allowed to use. However, once you have released a program, any user with authorization for it can run it. This means that the programmer is responsible for checking that the user is authorized to access the data that the program processes.

To check the authorization of the user of an ABAP program, use the AUTHORITY-CHECK statement:

```
AUTHORITY-CHECK OBJECT '<object>'
                ID '<name1>' FIELD <f1>
                ID '<name2>' FIELD <f2>
                  .............
                ID '<name10>' FIELD <f10>.
```

<object> is the name of the object that you want to check. You must list the names (<name1>, <name2> ...) of **all** authorization fields that occur in <object>. You can enter the values $<f_1>$, $<f_2>$.... for which the authorization is to be checked either as variables or as literals. The AUTHORITY-CHECK statement checks the user's profile for the listed object, to see whether the user has authorization for **all** values of <f>. Then, and only then, is SY-SUBRC set to 0. You can avoid checking a field by replacing FIELD <f> with DUMMY. You can only evaluate the result of the authorization check by checking the contents of SY-SUBRC. For a list of the possible return values and further information, see the keyword documentation for the AUTHORITY-CHECK statement. For further general information about the SAP authorization concept, refer to Users and Authorizations.

There is an authorization object called F_SPFLI. It contains the fields ACTVT, NAME, and CITY.

```
SELECT * FROM SPFLI.
  AUTHORITY-CHECK OBJECT 'F_SPFLI'
                  ID 'ACTVT'  FIELD '02'
                  ID 'NAME' FIELD SPFLI-CARRID
                  ID 'CITY'   DUMMY.
  IF SY-SUBRC NE 0. EXIT. ENDIF.
ENDSELECT.
```

If the user has the following authorizations for F_SPFLI:

ACTVT 01-03, NAME AA-LH, CITY none,

and the value of SPFLI-CARRID is not between “AA” and “LH”, the authorization check terminates the SELECT loop.

Checking User Authorizations

BC - Benutzer und Rollen

BC - Benutzer und Berechtigungen [Ext.]

Much of the data in an R/3 system has to be protected so that unauthorized users cannot access it. Therefore the appropriate authorization is required before a user can carry out certain actions in the system. When you log on to the R/3 system, the system checks in the user master record to see which transactions you are authorized to use. An authorization check is implemented for every sensitive transaction.

If you wish to protect a transaction that you have programmed yourself, then you must implement an authorization check.

This means you have to:

- allocate an authorization object in the definition of the transaction;
- program an AUTHORITY-CHECK.

  AUTHORITY-CHECK OBJECT <authorization object>
  ID <authority field 1> FIELD <field value 1>.
  ID <authority field 2> FIELD <field value 2>.
  ...
  ID <authority-field n> FIELD <field value n>.

  The **OBJECT** parameter specifies the authorization object.

  The **ID** parameter specifies an authorization field (in the authorization object).

  The **FIELD** parameter specifies a value for the authorization field.

The authorization object and its fields have to be suitable for the transaction. In most cases you will be able to use the existing authorization objects to protect your data. But new developments may require that you define new authorization objects and fields.

The following topics provide more information:

Defining Authorization Checks

Defining Authorization Objects

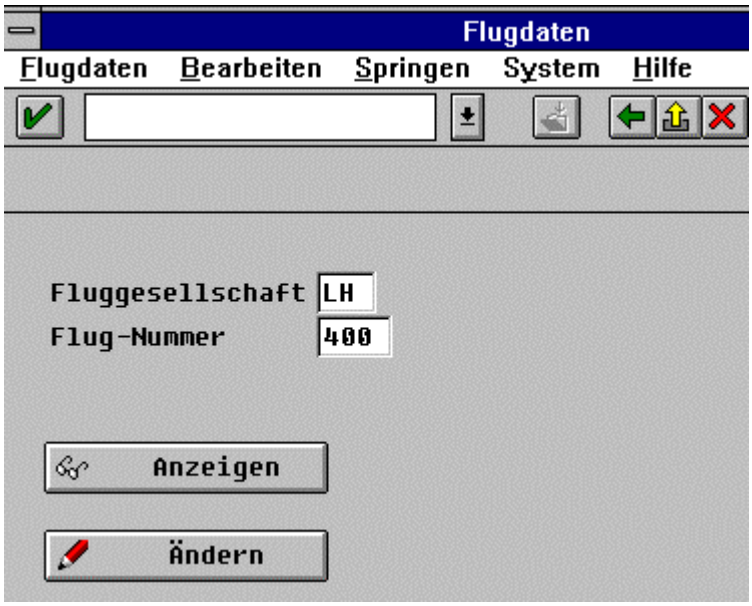
Defining Authorization Fields

Defining an Authorization Check

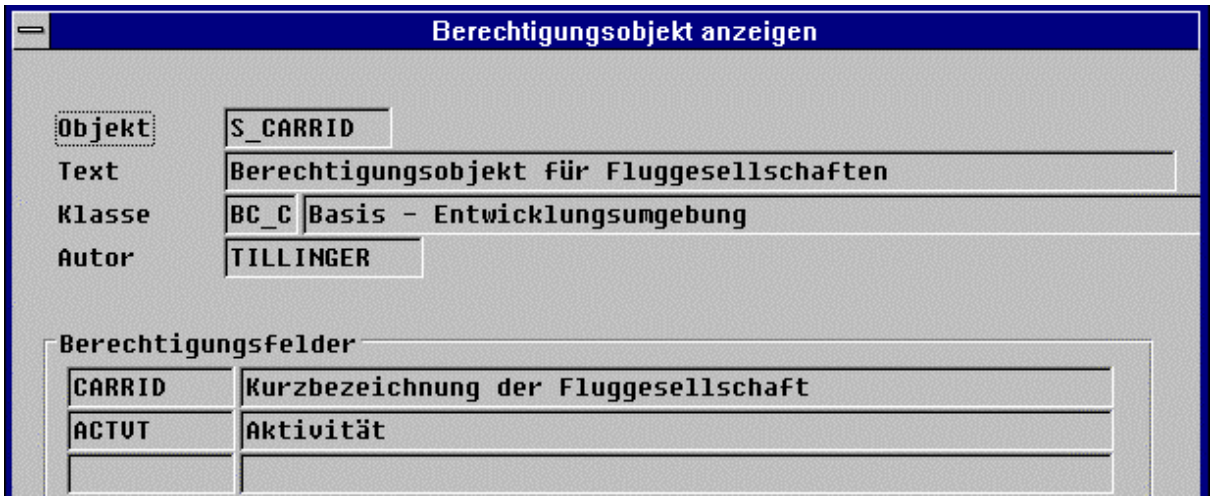
Selektionsbildverarbeitung [Page 739]

Transaction TZ80 contains an example of how to include an authorization check in a program.

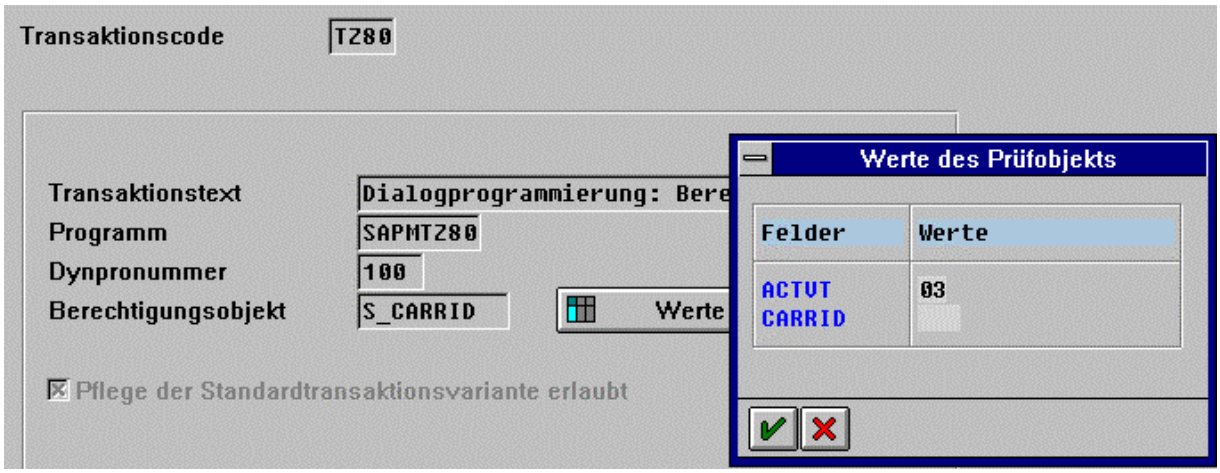
This example from the flight reservation system consists of two screens. On the first screen, the user can enter data and request details of the flight by choosing *Display*. Alternatively, the user can change the data by choosing *Change*.



The authorization object S_CARRID is allocated to Transaction TZ80. This authorization object contains two fields. Generic values can be entered in the first field *Airline carrier.* In the second field *Activity* you can choose between create (01), change (02) and display (03).



When you have programmed a new transaction, you can specify an authorization object in the definition of the transaction code. Next to the entry field for the authorization object is a pushbutton labeled *Values*.



In order to be able to start Transaction TZ80, you need an authorization in your user master record for the object *S_CARRID* containing the value Display (03) for the field Activity, and any value for the field *Airline carrier* (such as '*').

Maintenance of standard transaction variant allowed means that it is possible to create a default variant for the transaction. For further information, refer to the online manual for Transaction SHD0.

A more sophisticated authorization check is possible using the AUTHORITY-CHECK statement.

Within Transaction TZ80 you can only display and change flight data if you have the appropriate authorization in your user master record for the authorization object S_CARRID.

```
*&---------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
*&---------------------------------------------------------------*
MODULE USER_COMMAND_0100 INPUT.
  CASE OK_CODE.
    WHEN 'SHOW'.
          AUTHORITY-CHECK OBJECT 'S_CARRID'
                        ID 'CARRID' FIELD '*'
                        ID 'ACTVT'  FIELD '03'.

          IF SY-SUBRC NE 0. MESSAGE E009. ENDIF.
                MODE = CON_SHOW.
                SELECT SINGLE * FROM  SPFLI
                        WHERE  CARRID     = SPFLI-CARRID
                        AND    CONNID     = SPFLI-CONNID.
                IF SY-SUBRC NE 0.
                      MESSAGE E005 WITH SPFLI-CARRID SPFLI-CONNID.
                ENDIF.
                CLEAR OK_CODE.
                SET SCREEN 200.
    WHEN 'CHNG'.
          AUTHORITY-CHECK OBJECT 'S_CARRID'
                        ID 'CARRID' FIELD '*'
```

```
                    ID 'ACTVT'  FIELD '02'.
        IF SY-SUBRC NE 0. MESSAGE E010. ENDIF.
        MODE = CON_CHANGE.
        SELECT SINGLE * FROM  SPFLI
                      WHERE  CARRID      = SPFLI-CARRID
                      AND    CONNID      = SPFLI-CONNID.
        IF SY-SUBRC NE 0.
          MESSAGE E005 WITH SPFLI-CARRID SPFLI-CONNID.
        ENDIF.
        OLD_SPFLI = SPFLI.
        CLEAR OK_CODE.
        SET SCREEN 200.
  ENDCASE.
ENDMODULE.              " USER_COMMAND_0100  INPUT
```

If you choose the *Display* function in the transaction, the AUTHORITY-CHECK statement checks for the value ‘03’ in the ACTVT field in the S_CARRID authorization object. If you choose the *Change* function, then the value ‘02’ has to be present in the ACTVT field.

For further information about setting up authorizations in user master records, refer to Users and Authorizations.

Checking the Runtime of Program Segments

The ABAP Workbench provides several tools for the runtime measurement of complete application programs: SQL Trace and Runtime Analysis. For more information about these tools, see BC - ABAP Workbench: Tools [Ext.].

This section shows how you can measure the runtime of specific program segments when developing ABAP programs.

For this purpose, you use the GET RUN TIME FIELD statement.

GET RUN TIME FIELD [Page 509]

The following topic shows an example for measuring the runtime of database accesses.

Measuring the Runtime of Database Accesses [Page 511]

GET RUN TIME FIELD

You can use the GET RUN TIME FIELD statement to measure the relative runtime of program segments in microseconds. The syntax is as follows:

Syntax

GET RUN TIME FIELD <f>.

When this statement is called initially, the value of field <f>, which should be of type I, is set to zero. Each subsequent call of the statement sets the value of field <f> to the runtime of the program that has elapsed since the statement was called first.

In the client/server environment of an R/3 System, runtimes are no fixed values but can vary depending on the system load. SAP therefore recommends that you measure runtimes several times and calculate the minimum or the average from the results.

```
DATA T TYPE I.

GET RUN TIME FIELD T.
WRITE: / 'Runtime', T.

DO 10 TIMES.
  GET RUN TIME FIELD T.
  WRITE: / 'Runtime', T.
ENDDO.
```

The output list of this program segment might look as follows:

```
Runtime        0

Runtime    4.926
Runtime    5.228
Runtime    5.649
Runtime    6.100
Runtime    6.512
Runtime    6.906
Runtime    7.324
Runtime    7.724
Runtime    8.231
Runtime    8.623
```

After initialization, the runtime of the DO loop is added up in field T.

The best way to measure the runtime of a specific program segment is to calculate the difference between the relative runtimes before and after the segment.

```
DATA: T1 TYPE I,
      T2 TYPE I,
```

```
      T  TYPE P DECIMALS 2,
      N  TYPE I VALUE 1000.

T = 0.
DO N TIMES.
  GET RUN TIME FIELD T1.

*****************************
* Code to be tested         *
*****************************

  GET RUN TIME FIELD T2.
  T2 = T2 - T1.
  T = T + T2 / N.
ENDDO.

WRITE: / 'Mean Runtime: ', T, 'microseconds'.
```

This example shows a DO loop that is constructed around a program segment to be tested. In this example, the program segment is just a comment. The difference of the relative runtimes after (T2) and before (T1) the program segment is measured N times. The mean value (T) is calculated from the results. The output might look as follows:

Mean Runtime: 23,40 microseconds

The output shows the offset time that results from the runtime measurement itself. This offset must be subtracted from the runtime of a real program segment.

If you replace the comment block in the program with the simple assignment

T = T.

the result might be:

Mean Runtime: 28,84 microseconds

This shows that the runtime of a simple assignment (without type conversion) is about 5 microseconds. If you assign T to a field of type C, the runtime is increased by a factor of about four due to the type conversion.

Runtime Measurement of Database Accesses

The following example shows how you can use the GET RUN TIME FIELD statement to measure the runtime of database accesses.

```
DATA: T1 TYPE I,
      T2 TYPE I,
      T  TYPE P DECIMALS 2.

PARAMETERS N TYPE I DEFAULT 10.

DATA: TOFF  TYPE P DECIMALS 2,
      TSEL1 TYPE P DECIMALS 2,
      TSEL2 TYPE P DECIMALS 2.

TABLES SBOOK.

T = 0.
DO N TIMES.
  GET RUN TIME FIELD T1.

* Offset

  GET RUN TIME FIELD T2.
  T2 = T2 - T1.
  T = T + T2 / N.
ENDDO.
TOFF = T.
WRITE: / 'Mean Offset Runtime     :', TOFF, 'microseconds'.

SKIP.
T = 0.
DO N TIMES.
  GET RUN TIME FIELD T1.

  SELECT * FROM SBOOK.
  ENDSELECT.

  GET RUN TIME FIELD T2.
  T2 = T2 - T1.
  T = T + T2 / N.
ENDDO.
TSEL1 = T - TOFF.
WRITE: / 'Mean Runtime SELECT *   :',
         TSEL1, 'microseconds'.

SKIP.
T = 0.
DO N TIMES.
  GET RUN TIME FIELD T1.

  SELECT CARRID CONNID FROM SBOOK
                  INTO (SBOOK-CARRID, SBOOK-CONNID).
  ENDSELECT.
```

```
  GET RUN TIME FIELD T2.
  T2 = T2 - T1.
  T = T + T2 / N.
ENDDO.
TSEL2 = T - TOFF.
WRITE: / 'Mean Runtime SELECT List:',
         TSEL2, 'microseconds'.
```

The output might look as follows:

Mean Offset Runtime : 25,22 microseconds

Mean Runtime SELECT * : 257.496,85 microseconds

Mean Runtime SELECT List: 167,904.63 microseconds

This example analyses the runtime of three program segments:

- an ‘empty’ program segment to determine the offset of the runtime measurement
- a program segment that reads all data from database table SBOOK
- a program segment that reads two columns of database table SBOOK.

The result shows that the offset of the runtime measurement can be ignored in this case and that reading specific columns of a table is faster than reading complete rows.

Generating and Running Programs Dynamically

This section describes how to create, generate, and start ABAP modules during the runtime of your program. Simple examples in the following topics illustrate the procedure. However, you should be aware that this method affects system response times, since programs created dynamically must be regenerated for each run. SAP therefore recommends that you use the standard way of creating and generating program code with the ABAP Editor as frequently as possible.

The following topics provide information on:

Creating a New Program Dynamically [Page 514]

Changing Existing Programs Dynamically [Page 516]

Running Programs Created Dynamically [Page 517]

Creating and Starting Temporary Subroutines [Page 520]

Creating a New Program Dynamically

To create new dynamic programs during the runtime of an ABAP program, you must use an internal table. For this purpose, you should create this internal table with one character type column and a row width of 72. You can use any method you like from Filling Internal Tables [Page 278] to write the code of your new program into the internal table. Especially, you can use internal fields in which the contents depend on the flow of the program that you use to create the new program. The following example shows how to proceed generally:

```
DATA CODE(72) OCCURS 10.

APPEND 'REPORT ZDYN1.'
       TO CODE.

APPEND 'WRITE / "Hello, I am dynamically created!".'
       TO CODE.
```

Two lines of a simple program are written into internal table CODE.

In the next step you must transfer the new module, which in the above example is an executable program (report), into the program library. To do this, you can use the following statement:

Syntax

INSERT REPORT <prog> FROM <itab>.

Program <prog> is added to your current development class in the R/3 Repository. If a program with this name does not yet exist, a new program is created with the following attributes:

- Title: none
- Type: 1 (report)
- Application: S (Basis)

Either write the name of program <prog> using single quotation marks, or specify the name of a character field that contains the program name. It makes sense to use the name specified in the code as the program name, although you are not required to do so. <itab> is the internal table that contains the source code. For the above example, you could write:

```
INSERT REPORT 'ZDYN1' FROM CODE.
```

or

```
DATA REP(8).

REP = 'ZDYN1'

INSERT REPORT REP FROM CODE.
```

From the initial screen of the ABAP Editor, you can access and change all components of the new program.

You can call the ABAP Editor for the above example:

```
1  report zdyn1.
2  write / 'Hello, I am dynamically created!'.
3
4
```

If a program with name <prog> already exists, the source code of that program is replaced without warning with a new one from internal table <itab>.

Therefore, you have to be careful when assigning names to programs created dynamically. This is especially true if the program that creates the new program is simultaneously used by multiple users. In this case, employ a user-specific naming convention to ensure that one user does not replace the program of another user.

The GENERATE SUBROUTINE POOL statement that creates temporary external subroutines in the main memory is an alternative option of generating dynamic programs (see Creating and Starting Temporary Subroutines [Page 520]).

Changing Existing Programs Dynamically

Since you cannot check the syntax when a program is generated dynamically, you may want to have a set of syntactically correct structures in the system that you can load into an internal table, modify, and write back to the system. Such structures can consist of reports, subroutines, include programs, or modules from the module pool. Use the following syntax:

Syntax

READ REPORT <prog> INTO <itab>.

This statement reads program <prog> that is stored in the library into internal table <itab>.

Now, you can modify the internal table and create a new program as described in Creating a New Program Dynamically [Page 514].

Assume the following simple report:

REPORT ZSTRUC1.

WRITE / 'Hello, I am a little structure!'.

and the following lines of another program:

DATA CODE(72) OCCURS 10.

READ REPORT 'ZSTRUC1' INTO CODE.

APPEND 'SKIP.' TO CODE.
APPEND 'WRITE / "and I am a dynamic extension!".' TO CODE.

INSERT REPORT 'ZDYN2' FROM CODE.

In this example, the existing executable program (report) ZSTRUC1 is read into internal table CODE, it is extended by two additional lines and written to program ZDYN2. After the ABAP Editor for ZDYN2 has been called, the screen looks as follows:

```
1  report zstruc1 .
2  write / 'Hello, I am a little structure!'.
3  skip.
4  write / 'and I am a dynamic extension!'.
5
```

Running Programs Created Dynamically

After creating programs dynamically, you can start them in the normal way from the R/3 System. However, you may want not only to create, but also to start programs at runtime. In this case, you can

- use the SUBMIT statement,
- work with include programs or subroutines.

To start the created or modified program directly from your running program, use:

Syntax

SUBMIT <prog>.

For more information about the SUBMIT statement, see Calling Executable Programs (Reports) [Page 1018]

Assume the following simple report:

```
REPORT ZDYN3.
WRITE / 'Dynamic Program!'.
```

The following executable program (report) starts, modifies, and restarts ZDYN3:

```
REPORT ZMASTER1.
DATA CODE(72) OCCURS 10.
DATA LIN TYPE I.
READ REPORT 'ZDYN3' INTO CODE.
SUBMIT ZDYN3 AND RETURN.
DESCRIBE TABLE CODE LINES LIN.
MODIFY CODE INDEX LIN FROM
    'WRITE / "Dynamic Program Changed!".'.
INSERT REPORT 'ZDYN3' FROM CODE.
SUBMIT ZDYN3.
```

The output of this program is displayed on two subsequent output screens. The first screen displays:

Dynamic Program!

The second screen displays:

Dynamic Program Changed!

When you use the SUBMIT statement, all modifications made to a program during runtime take immediate effect before they are submitted. In the above example, ZDYN3 is submitted from ZMASTER1 first in its original and then in its modified form, generating different results.

This is not the case if you change the codes of include programs or subroutines dynamically.

Assume the following include program:

```
*** INCLUDE ZINCLUD1.
WRITE / 'Original INCLUDE program!'.
```

and an executable program (report) for modifying and including it:

```
REPORT ZMASTER2.
DATA CODE(72) OCCURS 10.
DATA LIN TYPE I.
READ REPORT 'ZINCLUD1' INTO CODE.
DESCRIBE TABLE CODE LINES LIN.
MODIFY CODE INDEX LIN FROM
        'WRITE / "Changed INCLUDE program!".'.
INSERT REPORT 'ZINCLUD1' FROM CODE.
INCLUDE ZINCLUD1.
```

If you run ZMASTER2, the source code of include program ZINCLUD1 is changed and replaced in the system. However, the last line of ZMASTER2 executes the older version since the runtime object of ZMASTER2 is generated before ZINCLUD1 is modified. Only when ZMASTER2 is run a second time, does the system determine that ZINCLUD1 has been changed. Exactly the same is true if you dynamically modify the source code of a subroutine and call it from within the same program.

One way to solve this problem is to use the INCLUDE statement within an external subroutine that is called by the program. This allows you to create or modify include programs or subroutines and use the updated versions directly in the same program.

Assume the following include program:

```
*** INCLUDE ZINCLUD1.
WRITE / 'Original INCLUDE program!'.
```

and an external subroutine:

```
PROGRAM ZFORM1.
FORM SUB1.
   INCLUDE ZINCLUD1.
ENDFORM.
```

The following program reads the include program, modifies it, enters it back into the system, and calls the subroutine.

```
REPORT ZMASTER3.
DATA CODE(72) OCCURS 10.
READ REPORT 'ZINCLUD1' INTO CODE.
```

```
APPEND 'WRITE / "Extension of INCLUDE program!".' TO CODE.
INSERT REPORT 'ZINCLUD1' FROM CODE.
PERFORM SUB1(ZFORM1).
```

This produces the following output:

Original INCLUDE program!

Extension of INCLUDE program!

In this case, the updated version of the include program is used in the subroutine because its time stamp is checked when the subroutine is called, and not when the calling program is generated.

Creating and Starting Temporary Subroutines

The dynamic programs discussed in the other topics of this section are created in the R/3 Repository. This topic describes how you can create and call dynamic subroutines in the main memory.

You write the source code of the subroutines as explained in Creating a New Program Dynamically [Page 514] into an internal table. To generate the program, you use the following statement:

Syntax

GENERATE SUBROUTINE POOL <itab> NAME <prog> [<options>].

This statement creates a subroutine pool in the main memory area of the running program. You pass the source code of the subroutine pool in internal table <itab>. The statement returns the name of the generated subroutine pool in field <prog> that should have type C of length 8. You use the name in <prog> to call the external subroutines defined in table <itab> through dynamic subroutine calls as explained in Specifying the Subroutine Name at Runtime [Page 470].

The subroutine pool only exists during the runtime of the generating program and can only be called from within this program. You can create up to 36 subroutine pools for one program.

If an error occurs during generation, SY-SUBRC is set to 8. Otherwise, it is set to 0.

You can use the following options <options> in the GENERATE SUBROUTINE POOL statement:

- MESSAGE <mess>

  In the case of a syntax error, field <mess> contains the error message.

- INCLUDE <incl>

  In the case of a syntax error, field <incl> contains the name of the include program in which the error possibly occurred.

- LINE <line>

  In the case of a syntax error, field <line> contains the number of the wrong line.

- WORD <word>

  In the case of a syntax error, field <word> contains the wrong word.

- OFFSET <offs>

  In the case of a syntax error, field <offs> contains the offset of the wrong word in the line.

- TRACE-FILE <trac>

  If you use this option, you switch on the trace mode and field <trac> contains the trace output.

  Compared to INSERT REPORT, the GENERATE SUBROUTINE POOL statement has a better performance. Furthermore, you do not have to care about naming conventions or restrictions of the correction and transport system when you use statement GENERATE SUBROUTINE POOL.

```
REPORT SAPMZTST.

DATA: CODE(72) OCCURS 10,
  PROG(8), MSG(120), LIN(3), WRD(10), OFF(3).

APPEND 'PROGRAM SUBPOOL.'
       TO CODE.
APPEND 'FORM DYN1.'
       TO CODE.
APPEND
 'WRITE / ''Hello, I am the temporary subroutine DYN1!''.'
       TO CODE.
APPEND 'ENDFORM.'
       TO CODE.
APPEND 'FORM DYN2.'
       TO CODE.
APPEND
 'WRIT / ''Hello, I am the temporary subroutine DYN2!''.'
       TO CODE.
APPEND 'ENDFORM.'
       TO CODE.

GENERATE SUBROUTINE POOL CODE NAME PROG
          MESSAGE MSG
          LINE  LIN
          WORD  WRD
          OFFSET OFF.

IF SY-SUBRC <> 0.
 WRITE: / 'Error during generation in line', LIN,
    / MSG,
    / 'Word:', WRD, 'at offset', OFF.
ELSE.
  WRITE: / 'The name of the subroutine pool is', PROG.
  SKIP 2.
 PERFORM DYN1 IN PROGRAM (PROG).
  SKIP 2.
 PERFORM DYN2 IN PROGRAM (PROG).
ENDIF.
```

In this program, a subroutine pool containing two subroutines is placed into table CODE. Note that the temporary program must contain a REPORT or PROGRAM statement. Statement GENERATE SUBROUTINE POOL generates the temporary program. The output is as follows:

```
Error during generation in line   6
The statement "WRIT" is not expected. A correct similar statement is "WRITE".
Word: WRIT       at offset   0
```

A generation error occurred since the WRITE statement has been misspelled in the second subroutine, DYN2. After that line has been revised:

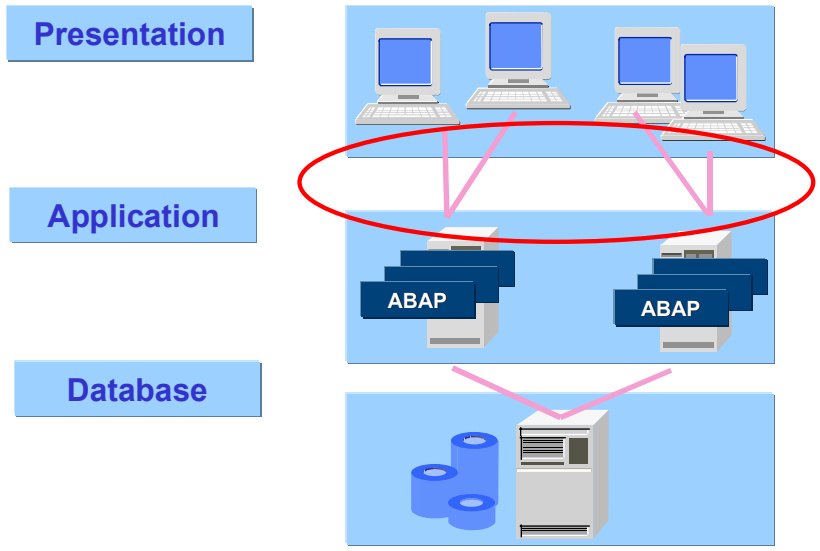
```
APPEND
'WRITE / "Hello, I am the temporary subroutine DYN2!".'
      TO CODE.
```

the output looks as follows:

```
The name of the subroutine pool is %_T01E00

Hello, I am the temporary subroutine DYN1!

Hello, I am the temporary subroutine DYN2!
```

Generation was successful. The internal program name is displayed. Then, the subroutines of the subroutine pool are called. Note that you do not need to know the program name to call the subroutines.

ABAP User Dialogs



This section describes the dialogs that you can use in ABAP to communicate with program users. It explains how to pass data between user dialogs and ABAP programs, how to define user dialogs within your programs, and how to process user input.

Currently, ABAP programs with type 1 (executable programs), M (module pools), and F (function groups) can contain their own user dialogs. In the future, it is planned to allow classes in ABAP Objects also to have screens.

Screens [Page 524]

Selection Screens [Page 681]

Lists [Page 771]

Switching Between Dialog Screens and List Display [Page 895]

Messages [Page 927]

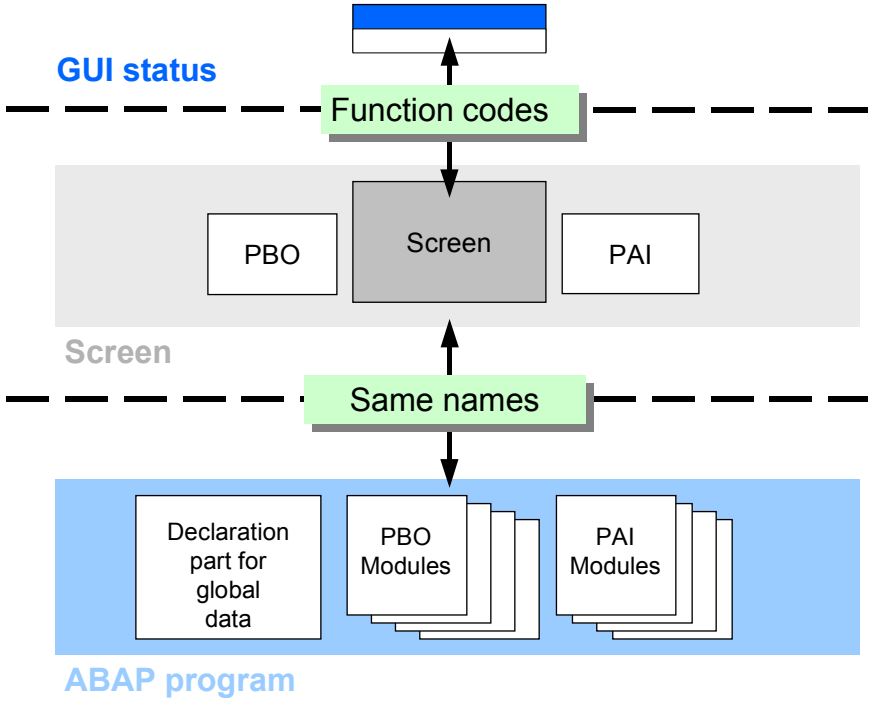
Screens

Screens are the most general type of user dialog that you can use in ABAP programs. You do not define them in ABAP programs, but instead in the Screen Painter. A screen consists of the input/output mask and the flow logic. You can define screens for any program with type 1, M, or F.

The screens in a single ABAP program can be combined to form screen sequences [Page 1000]. You can call single screens or screen sequences either using a transaction code from outside the ABAP program, or by using the CALL SCREEN statement in the corresponding ABAP program. When you call a screen or screen sequence, the screen flow logic takes control of the ABAP program execution [Page 937]. You can define screen sequences dynamically by setting the next screen attribute for a screen dynamically in the ABAP program.

A screen consists of the input/output mask and the flow logic. The screen flow logic is divided into the Process Before Output (PBO) event, which is processed before the screen is displayed, and the Process After Input (PAI) event, which is processed after a user action on the screen.

The following diagram shows the place of screens between the GUI status and the ABAP program:



The screen flow logic calls dialog modules in the ABAP program, either to prepare the screen for display (PBO event) or to process the user's entries (PAI event). Screens are dynamic programs, and have their own data objects, called screen fields. These are linked to the input/output fields that appear on the screen itself. When the screen is displayed, and when it finishes processing, the system passes data between the screen fields and data objects in the ABAP program. Data is copied between identically-named fields.

Each screen has a GUI status, containing a menu bar, standard toolbar, and an application toolbar. Like screens, GUI statuses are independent components of the ABAP program. You

create them in the ABAP Workbench using the Menu Painter. You assign GUI statuses to screens dynamically in your ABAP programs. Each screen is linked to the current GUI status by a special screen field into which the corresponding function code is placed whenever the user chooses a function. The screen passes the function code to the ABAP program just like any other screen field. You can then read it in the program.

Components of Screens [Page 526]

Processing Screens [Page 534]

Complex Screen Elements [Page 635]

Screen Elements

You create screens using the Screen Painter [Ext.] in the ABAP Workbench [Ext.]. The relevant documentation provides comprehensive information about creating screens and their elements. The following sections contain information about screen elements that is important from the aspect of ABAP programming using screens.

Screen Attributes

Like all objects in the R/3 Repository, screens have attributes that both describe them and determine how they behave at runtime. Important screen attributes for ABAP programming:

- Program

  The name of the ABAP program (type 1, M, or F) to which the screen belongs.

- Screen number

  A four-digit number, unique within the ABAP program, that identifies the screen within the program. If your program contains selection screens [Page 681], remember that selection screens and Screen Painter screens use the same namespace. For example, if you have a program with a standard selection screen, you may not contain any further screens with the number 1000. Lists, on the other hand, have their own namespace.

- Screen type

  A normal screen occupies a whole GUI window. Modal dialog boxes only cover a part of a GUI window. Their interface elements are also arranged differently. Selection screens are generated automatically from the definition in the ABAP program. You may not define them using the Screen Painter. A subscreen is a screen that you can display in a subscreen area on a different screen in the same ABAP program.

- Next screen

  Statically-defined screen number, specifying the next screen in the sequence. If you enter zero or leave the field blank, you define the current screen as the last in the chain. If the next screen is the same as the current screen, the screen will keep on calling itself. You can override the statically-defined next screen in the ABAP program.

- Cursor position

  Static definition of the screen element on which the cursor is positioned when the screen is displayed. By default, the cursor appears on the first input field. You can overwrite the static cursor position dynamically in your ABAP program.

- Screen group

  Four-character ID, placed in the system field SY-DYNGR while the screen is being processed. This allows you to assign several screens to a common screen group. You can use this, for example, to modify all of the screens in the group in a uniform way. Screen groups are stored in table TFAWT.

- Hold data

  If the user calls the screen more than once during a terminal session, he or she can retain changed data as default values by choosing *System* → *User profile* → *Hold data*.

Screen Elements

A screen can contain a wide variety of elements, either for displaying field contents, or for allowing the user to interact with the program (for example, filling out input fields or choosing pushbutton functions). You use the Screen Painter to arrange elements on the screen.

You can use the following elements:

- Text fields

  Display elements, which cannot be changed either by the user or by the ABAP program.

- Input/output fields

  Used to display data from the ABAP program or for entering data on the screen. Linked to screen fields.

- Dropdown list boxes

  Special input/output fields that allow users to choose one entry from a fixed list of possible entries.

- Checkboxes

  Special input/output fields which the user can select (value ‘X’) or deselect (value SPACE).

- Radio buttons

  Special input/output fields that are combined into groups. Within a radio button group, only a single button can be selected at any one time. When the user selects one button, all of the others are automatically deselected.

- Pushbuttons

  Elements on the screen that trigger the PAI event of the screen flow logic when chosen by the user. There is a function code attached to each pushbutton, which is passed to the ABAP program when it is chosen.

- Box

  Display element, used for visual grouping of related elements on the screen, such as radio button groups.

- Subscreens

  Area on the screen in which you can place another screen.

- Table controls

  Tabular input/output fields.

- Tabstrip controls

  Areas on the screen in which you can switch between various pages.

- Custom control

  Container on the screen in which you can display another control.

- Status icons

  Display elements, indicating the status of the application program.

- OK_CODE field

  Every screen has a twenty-character OK_CODE field (also known as the function code field), which is not displayed on the screen. User actions that trigger the PAI event also place the corresponding function code into this field, from where it is passed to the ABAP program. You can also use the command field in the standard toolbar to enter the function code. You must assign a name to the OK_CODE field to be able to use it for a particular screen.

All screen elements have a set of attributes, some of which are set automatically, others of which have to be specified in the Screen Painter. They determine things such as the layout of the screen elements on the screen. You can set the attributes of screen elements in the Screen Painter - either for a single element, or using the element list, which lists all of the elements belonging to the current screen. Some of the attributes that you set statically in the Screen Painter can be overwritten dynamically in the ABAP program. For a detailed list of all screen element attributes, refer to Working with Element Attributes [Ext.] in the Screen Painter documentation.

Screen Fields

Screen fields are fields in the working memory of a screen. Their contents are passed to identically-named fields in the ABAP program in the PAI event, and filled from the same identically-named fields in the program in the PBO event. The screen fields are linked with the input/output fields on the screen and with the OK_CODE field. Input and output fields must have a unique name (element attribute **Name**). This assigns them to a screen field with the same name, which allows you to work with their values in an ABAP program.

The technical attributes length and data type of a screen field are defined by the element attributes **DefLg** and **Format**. For the data type, you can select [Ext.] one of the data types in the ABAP Dictionary [Page 105]. These are converted appropriately when data is passed between the screen and the ABAP program. When you use fields [Ext.] from the ABAP Dictionary or the ABAP program, the name, length, and data type of the screen fields are defined automatically. Otherwise, you must set these attributes in the Screen Painter. Most importantly, you must specify the name of the OK_CODE field for each screen, so that a corresponding screen field can be defined. This allows you to use the OK_CODE field in your ABAP programs.

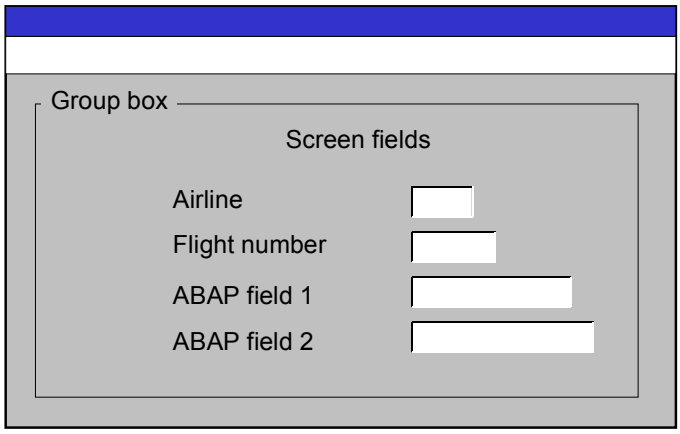
Display elements such as text fields or boxes are not linked to screen fields, and do not need a unique field name. When you create input and output fields by using [Ext.] ABAP Dictionary fields, the system usually also creates field labels using texts from the ABAP Dictionary. The default name for these text fields is the same name as the screen fields of the input and output fields.

As well as the screen fields defined in the Screen Painter, screens also recognize the same predefined system fields [Page 132] as the ABAP program. However, in screens, system fields are administered in the structure SYST, not SY. Therefore, you must address them using the form SYST-<name>.

Suppose we have the following program extract:

```
DATA: TEXT(10),
      NUMBER TYPE I.

CALL SCREEN 100.
```

Screen 100, created in the Screen Painter, contains two fields SPFLI-CARRID and SPFLI-CONNID from the ABAP Dictionary, and two fields, TEXT and NUMBER, from the ABAP program. It also contains three text fields and a box. Overall, the screen looks like this:



The columns in the element list that are relevant for the screen fields are the following:

Name	Type	defLg	Format	Text or I/O template
Box	Frame	62		Group box
%#AUTOTEXT001	Text	14		Screen fields
SPFLI-CARRID	Text	16		Airline
SPFLI-CARRID	I/O	3	CHAR	___
SPFLI-CONNID	Text	15		Flight number
SPFLI-CONNID	I/O	4	NUMC	____
%#AUTOTEXT002	Text	10		ABAP field 1
TEXT	I/O	10	CHAR	__________
%#AUTOTEXT003	Text	10		ABAP field 2
NUMBER	I/O	11	INT4	__________
	OK	20	OK	____________________

The screen has four input/output fields. Each of these fields is linked to an identically-named screen field. The data types and lengths of the screen fields are copied from the ABAP Dictionary or the ABAP program. In our example, the OK_CODE field still needs a name - for example, OK_CODE - to assign it to a screen field.

You can either assign a name to display fields with the type frame and text, or the system can automatically create field names for them. The texts that come from the ABAP Dictionary have the same names as the input/output fields. The texts created in the Screen Painter have generated names.

The last column shows the contents of the screen elements on the screen. Input/output fields only contain the template for the field, since the contents will not be known until runtime. The contents of the output fields, on the other hand, are statically-defined in the Screen Painter and cannot be affected by screen fields.

Screen Flow Logic

Screen flow logic contains the procedural part of a screen. You create it in the flow logic editor [Ext.], which is very similar to the ABAP Editor. The language used to program screen flow logic has a similar syntax to ABAP, but is not part of ABAP itself. It is sometimes referred to as screen language.

Unlike ABAP programs, screen flow logic contains **no** explicit data declarations. You define screen fields by placing elements on the screen mask.

The screen flow logic is like an ABAP program in that it serves as a container for processing blocks. There are four event blocks, each of which is introduced with the screen keyword PROCESS:

```
PROCESS BEFORE OUTPUT.
 ...
PROCESS AFTER INPUT.
 ...
PROCESS ON HELP-REQUEST.
 ...
PROCESS ON VALUE-REQUEST.
 ...
```

As in ABAP, the event block is introduced by the corresponding keyword statement, and it concludes either when the next block is introduced, or at the end of the program. The first two statements are created automatically by the Screen Painter when you create a new screen. The corresponding events are triggered by the runtime environment:

- PROCESS BEFORE OUTPUT (PBO) is automatically triggered after the PAI processing of the previous screen and before the current screen is displayed. You can program the PBO processing of the screen in this block. At the end of the PBO processing, the screen is displayed.
- PROCESS AFTER INPUT (PAI) is triggered when the user chooses a function on the screen. You can program the PAI processing of the screen in this block. At the end of the PAI processing, the system either calls the next screen or carries on processing at the point from which the screen was called.
- PROCESS ON HELP-REQUEST (POH) and PROCESS ON VALUE-REQUEST (POV) are triggered when the user requests field help (F1) or possible values help (F4) respectively. You can program the appropriate coding in the corresponding event blocks. At the end of processing, the system carries on processing the current screen.

As is normal with events, you must only program event blocks for the events to which you want the flow logic to react. However, the screen flow logic must contain at least the two statements PROCESS BEFORE OUTPUT and PROCESS AFTER INPUT in the correct order.

Within the event blocks, you can use the following keywords:

Keyword	Function
MODULE	Calls a dialog module in an ABAP program
FIELD	Specifies the point at which the contents of a screen field should be transported
ON	Used in conjunction with FIELD
VALUES	Used in conjunction with FIELD

CHAIN	Starts a processing chain
ENDCHAIN	Ends a processing chain
CALL	Calls a subscreen
LOOP	Starts processing a screen table
ENDLOOP	Stops processing a screen table

The functions of the individual statements are described in the following sections.

Processing Screens

There are two ways of calling a screen. You can either use a transaction code, or the CALL SCREEN statement in an ABAP program. When you call a screen, the PROCESS BEFORE OUTPUT event (PBO) is called, and the corresponding event block in the screen flow logic is processed. The screen itself is then displayed until the user triggers the PROCESS AFTER INPUT (PAI) event by choosing a function. In between, there may be field or input help processing. The corresponding event blocks in the flow logic are processed, if they exist. The main purpose of event blocks in the screen flow logic is to call dialog modules in the ABAP program and to provide the ABAP program with data.

User Actions on Screens [Page 535]

Calling ABAP Dialog Modules [Page 560]

Input Checks [Page 577]

Field Help, Input Help, and Dropdown Boxes [Page 589]

Modifying Screens Dynamically [Page 611]

User Actions on Screens

There are various ways in which users can interact with screens.

Filling Input Fields

Users can enter values in any input field on the screen, or change the value using the mouse, in the case of radio buttons and checkboxes. The contents are placed in the corresponding screen field. Filling an input field does **not** normally trigger the PAI event. Exceptions to this are Checkboxes and Radio Buttons with Function Codes [Page 545] and input fields with Drop Down Boxes [Page 607].

Triggering the PAI Event

There is a series of user actions that conclude the user's interaction with the screen in the SAPgui and pass control back to the runtime environment on the application server, where the PAI event is triggered.

These are:

- Choosing a pushbutton on the screen
- Selecting a checkbox or radio button to which a function code is assigned.
- Choosing a function in the menu, standard toolbar, or application toolbar.
- Choosing a function key on the keyboard.
- Selecting an entry from a drop down box.

All of these actions have in common that they are linked to a **function code**.

- The function code of a pushbutton, checkbox, radio button, or dropdown box on the screen is set in the corresponding element attributes.
- The function codes in menus, the standard toolbar, and the application toolbar are set in the GUI status.
- The function codes of function keys are also assigned in the GUI status.

If the OK_CODE field in the element list has a name (and is therefore assigned to the corresponding screen field), it is filled with the corresponding function code when the user chooses the function. If there is a field in the ABAP program with the same name, you can find out the function that the user chose by querying its contents in a PAI module. If the OK_CODE field does not have a name, the PAI event is still triggered, but there is no screen field into which the function code can be passed. Note also that the OK_CODE field is filled with the contents of the identically-named field in the ABAP program in the PBO event.

The PAI event is always triggered when the user resizes a screen containing elements for which the *Resizing* attribute is active. This applies to table controls, subscreens, and custom controls.

Processing Input/Output Fields [Page 537]

Pushbuttons on the Screen [Page 542]

Checkboxes and Radio Buttons with Function Codes [Page 545]

Using GUI Statuses [Page 548]

User Actions on Screens

Processing Input/Output Fields

Anbindung von ABAP-Feldern dynamischer Länge an Dynprofelder [Ext.]

Input/output fields can be either conventional fields in which the user can enter values using the keyboard or by selecting from a value list, or checkboxes or radio buttons, for which the mouse is required. All input/output fields have a name linking them to a screen field. The data type of the screen field determines the input format. For example, you cannot enter letters in a numeric field. The screen recognizes when you try to enter invalid values. Radio buttons and checkboxes always have the data type CHAR and length 1. A selected radio button or checkbox has the value 'X', when empty, both have the value SPACE.

The ABAP program must contain identically-named data objects that correspond to the screen fields, otherwise data may be lost. The ABAP data types that correspond to the screen data types are listed in a table in the Data Types in the ABAP Dictionary [Page 105] section. The ABAP fields for checkboxes and radio buttons must have type C and length 1.

After the PBO event has been processed, the screen fields are filled with the values from the ABAP fields. Before or during the PAI event, the values from the screen fields are written to the ABAP fields. Note that only one radio button within a group can be selected. If more than one of the fields contains the value 'X', the program terminates.

To ensure that screen fields and ABAP fields always correspond exactly, use the following procedure.

Local Program Fields

If you want to use input/output fields in one program only, you should create them in the ABAP program, activate the program, and then copy [Ext.] the fields from the ABAP program to the screen. Afterwards, however, you must avoid changing the fields in the ABAP program.

Screen Fields with Dictionary Reference

If your input/output fields are required in more than one program, and you want to use information from the ABAP Dictionary, such as field labels, field help, and input help, you should copy [Ext.] fields from the ABAP Dictionary. You can refer to both structures and database tables.

You must then declare identically-named fields as an interface work area in the ABAP program using the TABLES statement. Declaring an identically-named field using a TYPES reference to the data type in the ABAP Dictionary is insufficient for data to be transferred between the screen and the ABAP program.

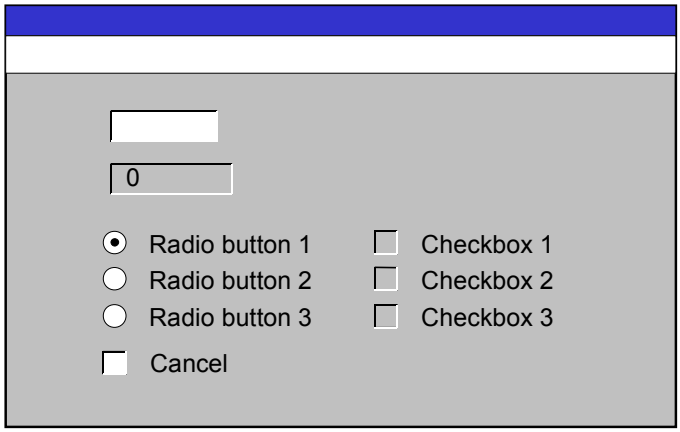
To avoid naming conflicts between screen fields, work areas in programs, and database tables, it is often worth defining your own ABAP Dictionary structure for screens, containing all of the input/output fields you want to use for one or more screens of a program, or even for a whole development class. The work areas that you declare with TABLES in the ABAP program then function exclusively as an interface between the program and the screen. If necessary, you can assign values between program fields and the interface work area.

The advantage of referring to ABAP Dictionary data types is that both the screen fields and the fields in the ABAP program are updated automatically if the data type changes.

Local program fields

```
PROGRAM DEMO_DYNPRO_INPUT_OUTPUT.

DATA: INPUT  TYPE I,
      OUTPUT TYPE I,
      RADIO1, RADIO2, RADIO3, BOX1, BOX2, BOX3, EXIT.

CALL SCREEN 100.

MODULE INIT_SCREEN_100 OUTPUT.
  CLEAR INPUT.
  RADIO1 = 'X'.
  CLEAR: RADIO2, RADIO3.
ENDMODULE.

MODULE USER_COMMAND_0100 INPUT.
  OUTPUT = INPUT.
  BOX1 = RADIO1.
  BOX2 = RADIO2.
  BOX3 = RADIO3.
  IF EXIT NE SPACE.
    LEAVE PROGRAM.
  ENDIF.
ENDMODULE.
```

The next screen (statically defined) for screen 100 is itself. It has the following layout:



The part of the element list relevant for the screen fields is as follows:

Name	Type	defLg	Format
INPUT	I/O	9	INT4
OUTPUT	I/O	11	INT4
RADIO1	Radio	1	CHAR
BOX1	Check	1	CHAR
RADIO2	Radio	1	CHAR
BOX2	Check	1	CHAR

RADIO3	Radio	1	CHAR
BOX3	Check	1	CHAR
EXIT	Check	1	CHAR
	OK	20	OK

The screen fields OUTPUT, BOX1, BOX2, and BOX3 cannot accept user input.

The length of INPUT is such that the user can enter a nine-digit integer without thousand separators. However, the display in the OUTPUT field contains up to two thousand separators. Had we left the length of INPUT at 11 digits, a runtime error could have occurred.

The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE INIT_SCREEN_100.

PROCESS AFTER INPUT.
  MODULE USER_COMMAND_0100.
```

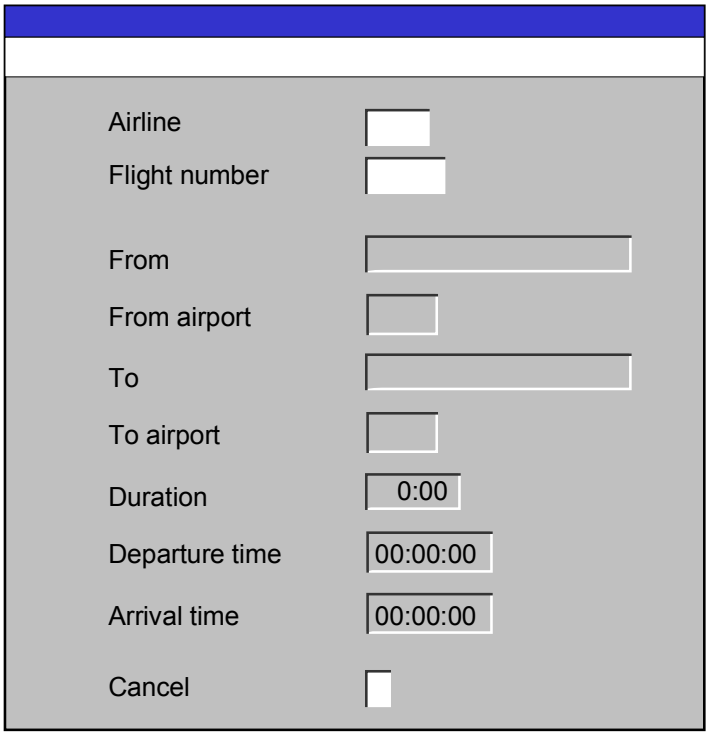
The entries in the input fields are passed to the ABAP program in the PAI event, and assigned to the output fields in the dialog module USER_COMMAND_100. The next time the screen appears, the screen fields have been filled appropriately. The input fields are set in the PBO dialog module INT_SCREEN_100. If the user selects the screen field EXIT (value ‘X’), the program ends.

Screen fields with Dictionary reference

```
PROGRAM DEMO_DYNPRO_DICTIONARY.

TABLES SDYN_CONN.

DATA WA_SPFLI TYPE SPFLI.

CALL SCREEN 100.

MODULE INIT_SCREEN_100 OUTPUT.
  CLEAR SDYN_CONN-MARK.
  MOVE-CORRESPONDING WA_SPFLI TO SDYN_CONN.
  CLEAR WA_SPFLI.
ENDMODULE.

MODULE USER_COMMAND_0100 INPUT.

  IF SDYN_CONN-MARK = 'X'.
    LEAVE PROGRAM.
  ENDIF.

  MOVE-CORRESPONDING SDYN_CONN TO WA_SPFLI.

  SELECT  SINGLE
        CITYFROM AIRPFROM CITYTO AIRPTO FLTIME DEPTIME ARRTIME
    INTO  CORRESPONDING FIELDS OF WA_SPFLI
    FROM  SPFLI
    WHERE CARRID = WA_SPFLI-CARRID AND CONNID = WA_SPFLI-CONNID.
```

ENDMODULE.

The statically-defined next screen for screen 100 is 100. It uses components of the structure SDYN_CONN, copied [Ext.] from the ABAP Dictionary, and looks like this:



The structure SDYN_CONN exists in the ABAP Dictionary specially for screens that use the flight data model. It contains the components of the database table SPFLI, but also a component MARK. MARK uses the domain S_FLAG, which may only have the values SPACE and X. On the above screen, the ABAP Dictionary text for MARK has been overwritten with 'Cancel'. For all other fields, the ABAP Dictionary texts have been retained. The input attribute of some of the fields has been switched off in the Screen Painter.

Users can enter values for the airline and flight number. Automatic field checks against check tables in the ABAP Dictionary, field help, and input help are automatically available. The field checks are performed automatically before any of the dialog modules in the ABAP program are called. Users cannot enter values for the airline that do not exist in the check table SCARR, values for the flight number that are not in SPFLI, or values for MARK other than SPACE and X. There is no need to program any of these checks in the ABAP program.

The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE INIT_SCREEN_100.

PROCESS AFTER INPUT.
  MODULE USER_COMMAND_0100.
```

The module USER_COMMAND_100 in the ABAP program reads data from the database table with the key fields specified (and checked) on the screen, and sends

them back to the screen in the PBO module INIT_SCREEN_100. The work area SDYN_CONN, defined using the TABLES statement, serves as an interface between the program and the screen. Meanwhile the data from the database is processed in the work area WA_SPFLI. If the user selects the screen field ‘Cancel’ (value ‘X’), the program ends.

Pushbuttons on the Screen

Pushbuttons are the only elements of screens that trigger the PAI event when the user chooses them. In the attributes of a pushbutton, you can specify a function code up to 20 characters long.

Pushbuttons have a label - the text that you specify statically in the attributes - and can also have icons. If you set the *Output field* attribute for a pushbutton, you can also set its text dynamically in the ABAP program. To do this, you must create a field in the ABAP program with the same name as the pushbutton. You must then assign the required text to the field before the screen is displayed. You can also assign icons to dynamic texts on pushbuttons by including the icon code in the text. The icon codes are all contained in the include program <ICON>. For example, the ICON_CANCEL () icon has the code @0W@.

In each PAI event, the function code, as long as it is not empty, is placed in the system field SYST-UCOMM (SY-UCOMM) and assigned to the OK_CODE field. Empty function codes are placed in neither the SY-UCOMM field nor the OK_CODE field. The function code of a pushbutton is empty if you have not entered it in the corresponding attribute in the Screen Painter. Before you can work with the OK_CODE field, you must assign a name to it in the Screen Painter. For further information, refer to Reading Function Codes [Page 555].

The main way of allowing users to choose functions that trigger the PAI event and send function codes to the program is through the GUI status. The main reason is one of space - the GUI status can contain many more functions than you would be able to place on the screen as pushbuttons.

However, you should use pushbuttons in the following cases:

- Very important and frequently-used function codes that should appear on the screen as well as in the GUI status, for example, the *Execute*, *Display*, and *Change* functions on the initial screen of the ABAP Editor.
- Applications that only use pushbuttons, where function codes in the GUI status make no sense - for example, in a calculator application. The function codes of the corresponding pushbuttons have no corresponding function in the GUI status elements, and only exist as attributes of screen elements.
- When you have a GUI status that is used for a large number of screens, but use it on screens that have a particular function that is not required anywhere else. In this case, you should place it on the screen as a pushbutton.

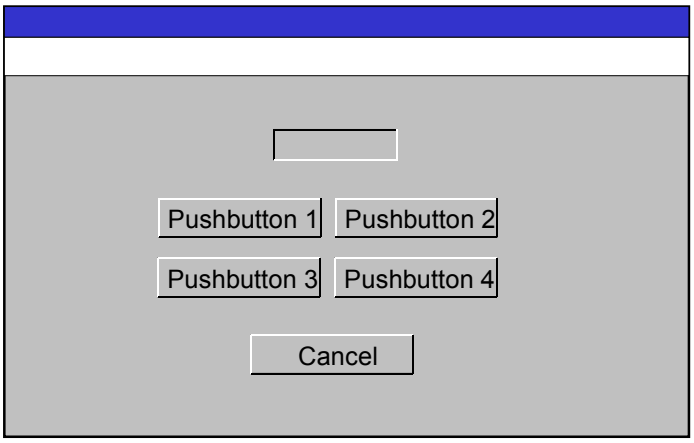
When you define pushbuttons, you must ensure that your function code is either the same as a corresponding function in the GUI status, or does not accidentally coincide with the function code of an entirely different function. Although the consistency of function codes in GUI statuses is checked in the Menu Painter, this is not possible for pushbuttons, since the GUI status of a screen is not known until runtime.

Pushbuttons on screens.

```
PROGRAM DEMO_DYNPRO_PUSH_BUTTON.

DATA: OK_CODE LIKE SY-UCOMM,
      SAVE_OK LIKE OK_CODE,
      OUTPUT(8) TYPE C.

CALL SCREEN 100.
```

```
MODULE USER_COMMAND_0100 INPUT.
  SAVE_OK = OK_CODE.
  CLEAR OK_CODE.
  CASE SAVE_OK.
    WHEN 'BUTTON_EXIT'.
      LEAVE PROGRAM.
    WHEN 'BUTTON_1'.
      OUTPUT = 'Button 1'(001).
    WHEN 'BUTTON_2'.
      OUTPUT = 'Button 2'(002).
    WHEN 'BUTTON_3'.
      OUTPUT = 'Button 3'(003).
    WHEN 'BUTTON_4'.
      OUTPUT = 'Button 4'(004).
    WHEN OTHERS.
      OUTPUT = SAVE_OK.
  ENDCASE.
ENDMODULE.
```

The next screen (statically defined) for screen 100 is itself. It has the following layout:



The part of the element list relevant for the screen fields is as follows:

Name	Type	Format	Text	Function code
OUTPUT	I/O	CHAR		
BUTTON1	Push		Pushbutton 1	BUTTON_1
BUTTON2	Push		Pushbutton 2	BUTTON_2
BUTTON3	Push		Pushbutton 3	BUTTON_3
BUTTON4	Push		Pushbutton 4	BUTTON_4
EXIT_BUTTTON	Push		Cancel	BUTTON_EXIT
OK_CODE	OK	OK		

Pushbuttons on the Screen

The input option for the screen field OUTPUT has been switched off in the Screen Painter.

The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.

PROCESS AFTER INPUT.
  MODULE USER_COMMAND_0100.
```

When the user chooses a pushbutton, the PAI event is triggered. The function code of the pushbutton is assigned to the screen field OK_CODE, which is then passed onto the ABAP field with the same name. The module USER_COMMAND_0100 is then processed.

Firstly, the contents of the OK_CODE field are copied to the auxiliary variable SAVE_OK code, and OK_CODE is initialized. You should always do this, since it ensures that the screen field OK_CODE has always been initialized before the PBO event and can therefore not unintentionally contain an incorrect value.

Next, in the CASE structure, a text symbol is assigned to the OUTPUT field according to the button that the user chose. This is displayed in the output field on the screen. If the user chooses *Cancel*, the program ends.

Checkboxes and Radio Buttons with Function Codes

In the Screen Painter, you can assign a function code (up to 20 characters long) to checkboxes and radio buttons.

Checkboxes and radio buttons **without a function code** behave like normal input/output fields. Clicking the object changes the contents of the field, but does not trigger the PAI event. (Clicking a pushbutton, on the other hand, always triggers the PAI event, even if it has an empty function code.)

When a **function code is assigned** to a checkbox or radio button, clicking it not only changes the field contents, but also triggers the PAI event and places the function code in the OK CODE field. For further information, refer to Reading Function Codes [Page 555].

While it is possible to assign an individual function code to each checkbox, you can only assign one function code to all of the radio buttons in a group. When you assign a function code to a radio button in the Screen Painter, the system automatically applies the same function code to all of the other radio buttons in the group.

You can use checkboxes and radio buttons with function codes as follows:

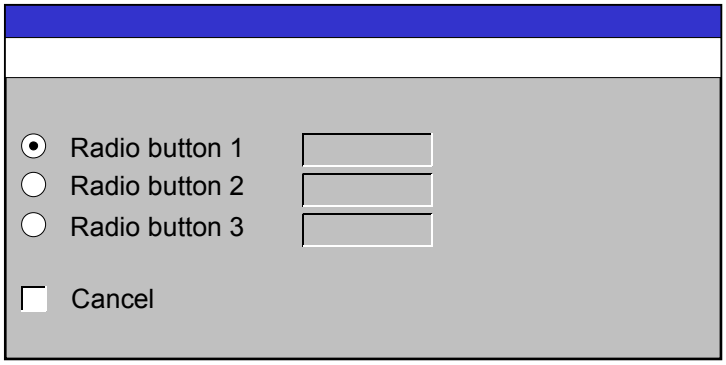
- For processing parts of screens (context-sensitive processing). For example, only when a radio button or checkbox is selected is particular data read and placed in the corresponding input/output fields.
- You can fill fields with patterns depending on checkboxes or radio buttons. A typical example would be formatting settings for letters. The input fields can all be processed separately, but it is possible to fill all input fields simultaneously and consistently by choosing a pattern.
- You can control dynamic screen modifications [Page 611] directly using checkboxes or radio buttons. For example, you can make sure that an input/output field cannot accept input until the user selects a radio button.

As when you create pushbuttons [Page 542], you should ensure when you assign function codes to checkboxes and radio buttons that they do not coincide with function codes from the GUI status.

```
PROGRAM demo_dynpro_check_radio.

DATA: radio1(1) TYPE c, radio2(1) TYPE c, radio3(1) TYPE c,
      field1(10) TYPE c, field2(10) TYPE c, field3(10) TYPE c,
      box TYPE c.

DATA: ok_code TYPE sy-ucomm,
      save_ok TYPE sy-ucomm.

CALL SCREEN 100.

MODULE user_command_0100 INPUT.
  save_ok = ok_code.
  CLEAR ok_code.
  CASE save_ok.
    WHEN 'RADIO'.
      IF radio1 = 'X'.
        field1 = 'Selected!'.
        CLEAR: field2, field3.
```

```
      ELSEIF radio2 = 'X'.
        field2 = 'Selected!'.
        CLEAR: field1, field3.
      ELSEIF radio3 = 'X'.
        field3 = 'Selected!'.
        CLEAR: field1, field2.
      ENDIF.
    WHEN 'CANCEL'.
      LEAVE PROGRAM.
  ENDCASE.
ENDMODULE.
```

The next screen (statically defined) for screen 100 is itself. It has the following layout:



The part of the element list relevant for the screen fields is as follows:

Name	Type	DefLg	Format	Function code
RADIO1	Radio	1	CHAR	RADIO
FIELD1	I/O	10	CHAR	
RADIO2	Radio	1	CHAR	RADIO
FIELD2	I/O	10	CHAR	
RADIO3	Radio	1	CHAR	RADIO
FIELD3	I/O	10	CHAR	
BOX	Check	1	CHAR	CANCEL
OK_CODE	OK	20	OK	

The input option for the screen fields FIELD1 to FIELD3 has been switched off in the Screen Painter.

The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.

PROCESS AFTER INPUT.
  MODULE user_command_0100.
```

Each time you select one of the three radio buttons, the PAI event is triggered and the function code RADIO and the contents of the screen fields are passed to the

ABAP program. The dialog module USER_COMMAND_100 fills the fields FIELD1 to FIELD3 according to the radio button that was selected. These field contents appear the next time the screen is sent.

The PAI event is also triggered if you select the checkbox. In this case, the function code CANCEL is passed to the ABAP program, and the dialog module USER_COMMAND_100 terminates the program immediately.

Using GUI Statuses

Kontextmenüs [Page 639]

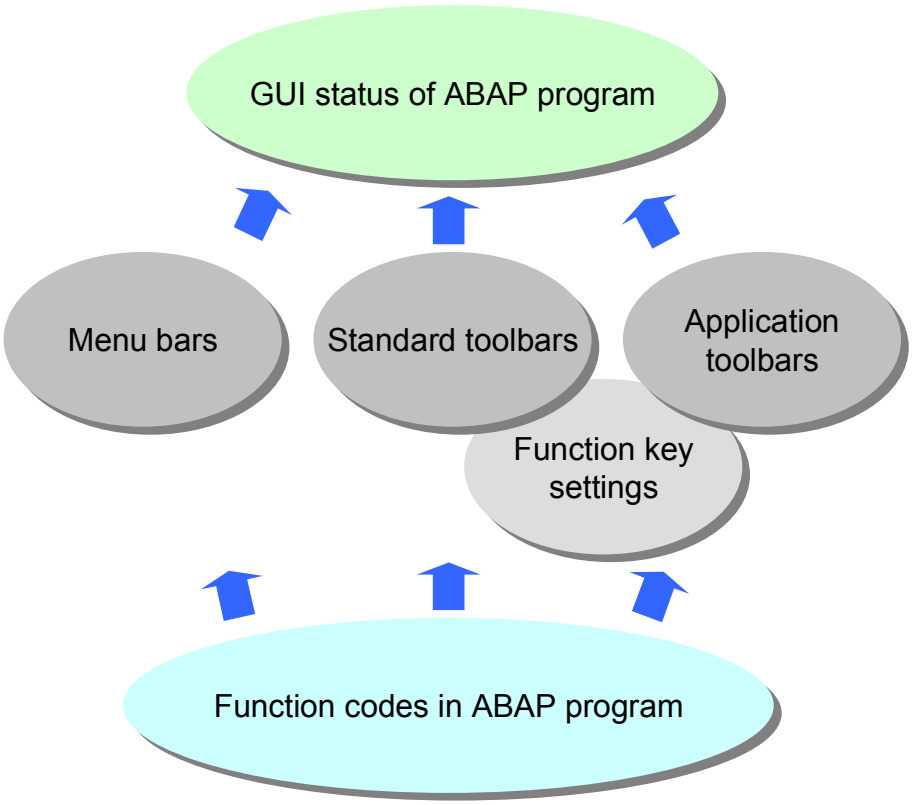
Kontextmenüs [Page 639]

A GUI status is an independent component of an ABAP program. You create them in the ABAP Workbench [Ext.] using the Menu Painter [Ext.]. The relevant documentation provides comprehensive information about creating GUI statuses and their elements. The most important details are contained below.

GUI Status

The function of a GUI status is to provide the user with a range of functions on a screen. Each function has an associated function code, and when the user chooses a function, the PAI event is triggered. In each PAI event, the function code, as long as it is not empty, is placed in the system field SYST-UCOMM (SY-UCOMM) and assigned to the OK_CODE field. Empty function codes are placed in neither the SY-UCOMM field nor the OK_CODE field. Before you can work with the OK_CODE field, you must assign a name to it in the Screen Painter. For further information, refer to Reading Function Codes [Page 555].

All function codes in an ABAP program, apart from those only assigned to pushbuttons on screens, are defined and adminstered in the Menu Painter.



When you define a function code in the Menu Painter, you assign it to a menu entry in a menu bar, and possibly also to a freely-assigned function key on the keyboard. Function codes that are assigned to a function key can also be assigned to an icon in the standard toolbar or a pushbutton in the application toolbar. A GUI status consists of a menu bar, a standard toolbar, an application toolbar, and a function key setting. For each function code, there is a static or dynamic function text [Ext.]. Dynamic function texts allow you to use context-sensitive texts in your ABAP program.

The user interface is a visualization of all possible functions, and is meant to make programs easier to use. From a technical point of view, you could just enter the function code in the command field and trigger the PAI event by pressing ENTER. However, a GUI status should contain **all** possible function codes as menu entries. The most important functions should also be assigned to function keys, and the most important of these should also be assigned to icons in the standard toolbar or the application toolbar.

In the Menu Painter, the functions that you assign to the icons in the standard toolbar must also be assigned to particular function keys. You cannot assign them freely. Instead, the system automatically assigns them to the corresponding function key when you assign the function code to an icon. You should always activate at least one of the Back (F3), Exit (shift+F3), and Cancel (F12) functions, so that the user can always leave the screen in the normal fashion. The assignment of function keys to pushbuttons in the application toolbar is not fixed.

Whenever you assign function codes to interface elements, you should refer to the ergonomic guidelines in the SAP Style Guide [Ext.]. The Menu Painter provides you with help in this respect, since you can use the *Display standards* function and corresponding checks.

As well as the function codes for Normal application functions, which trigger the PAI event, you can also create function codes in the Menu Painter with other functions, for example, to call another transaction or trigger a system functions. You can also create special function codes for pushbuttons on screens, but you must assign another function type to the function code in either the Menu Painter or the Screen Painter.

You can also use the Menu Painter to create area menus. An area menu is a user interface that does not belong to an ABAP program and may therefore not trigger any PAI events. It may only contain function codes that call transactions or system functions. Examples of area menus are the initial screen of the R/3 System (S000) and the initial screen of the ABAP Workbench (S001).

Special Function Keys and Function Codes

Some keys and function codes have special functions. This section deals with the special functions on screens. There are also special functions that apply to lists.

Reserved Function Keys

The following function keys do not trigger the PAI event, but are reserved for other functions:

- F1 calls the field help
- F4 calls the input help
- F10 places the cursor in the menu bar

Function Keys for the Standard Toolbar

The following function keys are firmly assigned to icons in the standard toolbar. The function code that you assign and the corresponding ABAP coding should reflect the sense of the icon.

Using GUI Statuses

Function key	Icon	Purpose
Ctrl+S or F11		Save
F3		Back
Shift+F3		Exit
Esc or F12		Cancel
Ctrl+P		Print
Ctrl+F		Find
Ctrl+G		Find next
Ctrl+PgUp		First page
PgUp		Previous page
PgDn		Next page
Ctrl+PgDn		Last page

The ENTER Key

The ENTER key belongs to the  icon in the standard toolbar, and is always active, even if no function is assigned to it in the GUI status, or there is no GUI status set at all. The PAI event is always triggered when the user chooses ENTER. The following function codes can be passed to SY-UCOMM or the OK_CODE field:

- Any entry in the command field when the user presses ENTER
- If there is no entry in the command field, any function code assigned to the ENTER key in the Menu Painter.
- If the command field does not contain an entry and no function code is assigned to the ENTER key in the Menu Painter, the function code is empty and therefore not passed to SY-UCOMM or the OK_CODE field.

The F2 Key

The F2 key is **always** linked to a mouse double-click. If a function code in the GUI status is assigned to the F2 key, the PAI event can be triggered either when the user chooses F2 or when he or she double-clicks a screen element. In both cases, the corresponding function code is passed to SY-UCOMM and the OK_CODE field. If you need to use the cursor position to control what happens next, you must find it out yourself using the GET CURSOR statement.

Setting the GUI Status

To assign a GUI status to a screen, use the ABAP statement

```
SET PF-STATUS <stat> [OF PROGRAM <prog>]
                [EXCLUDING <f>|<itab>].
```

This statement defines the user interface for all subsequent screens of a screen sequence until another is set using a new SET PF-STATUS statement. The GUI status <stat> must be a component of the current ABAP program, unless you use the OF PROGRAM addition in the SET PF-STATUS statement to set a GUI status of another program <prog>.

The EXCLUDING function allows you to change the appearance and function of a GUI status dynamically. This is useful if the individual user interfaces for a range of screens are very similar. You can define a single global status, and then just deactivate the functions you do not need using EXCLUDING. Specify <f> to deactivate the function code stored in field <f>. Specify <itab> to deactivate all function codes stored in the internal table <itab>. Field <f> and the lines of table <itab> should be of type C, and have length 20.

You should set the GUI status for a screen in the PBO event. If you do not specify a GUI status for a screen, it is displayed with the interface of the previous screen. If you do not specify a GUI status for the first screen of a program, it has no user interface, and the user may not be able to leave the screen.

Setting a GUI Title

As well as the GUI status, a user interface also contains a GUI title, which you also create using the Menu Painter. To assign a GUI status to a screen, use the ABAP statement

```
SET TITLEBAR <title> [OF PROGRAM <prog>]
                [WITH <g1 >... <g9>].
```

This statement defines the title of the user interface for all subsequent screens of a screen sequence until another is set using a new SET TITLEBAR statement. The GUI title <title> must be a component of the current ABAP program, unless you use the OF PROGRAM addition in the SET TITLEBAR statement to set a GUI status of another program <prog>.

A GUI title can contain up to nine placeholders &1 ... &9 that can be replaced with the contents of the corresponding fields <g1> ... <g9> in the WITH addition. The system also replaces '&' placeholders in succession by the contents of the corresponding $<g_i>$ parameters. To display an ampersand character '&', repeat it in the title '&&'.

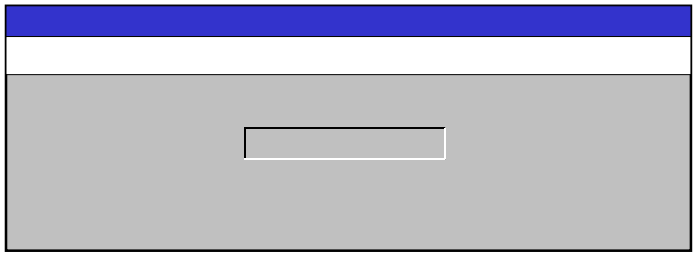
You should set the GUI title for a screen in the PBO event.

GUI status and screen

```
PROGRAM DEMO_DYNPRO_GUI_STATUS.

DATA: OK_CODE LIKE SY-UCOMM,
      SAVE_OK LIKE OK_CODE,
      OUTPUT  LIKE OK_CODE.

CALL SCREEN 100.

MODULE INIT_SCREEN_0100 OUTPUT.
  SET PF-STATUS 'STATUS_100'.
  SET TITLEBAR '100'.
ENDMODULE.

MODULE USER_COMMAND_0100 INPUT.
  SAVE_OK = OK_CODE.
  CLEAR OK_CODE.
  CASE SAVE_OK.
    WHEN 'BACK' OR 'EXIT' OR 'CANCEL'.
      LEAVE PROGRAM.
    WHEN OTHERS.
      OUTPUT = SAVE_OK.
```

```
   ENDCASE.
ENDMODULE.
```

The next screen (statically defined) for screen 100 is itself. It has the following layout:



The part of the element list relevant for the screen fields is as follows:

Name	Type	Format
OUTPUT	I/O	CHAR
OK_CODE	OK	OK

The input option for the screen field OUTPUT has been switched off in the Screen Painter.

The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE INIT_SCREEN_0100.

PROCESS AFTER INPUT.
  MODULE USER_COMMAND_0100.
```

The PBO module INIT_SCREEN_100 sets the GUI status STATUS_100 and a title 100. The GUI status was created using the Menu Painter as follows:

Menu bar

Menu *Demo*:

Function code	Text
SAVE	Save
PRINT	Print
DELETE	Delete
EXIT	Exit

Menu *Edit*:

Function code	Text
MARK	Select
SELE	Choose
SEARCH	Find
SEARCH+	Find next

TOP	First page
PAGE_UP	Previous page
PAGE_DOWN	Next page
BOTTOM	Last page
CANCEL	Cancel

Menu *Goto*:

Function code	Text
BACK	Back

Menu *Extras*:

Function code	Text
FUNCT_F5	Function 1
FUNCT_F6	Function 2
FUNCT_F7	Function 3
FUNCT_F8	Function 4

Menu *Environment*:

Function code	Text
MENU_1	Menu 1
MENU_2	Menu 2
MENU_3	Menu 3

Standard toolbar

Icon	Function code	Text
		
	SAVE	Save
	BACK	Back
	EXIT	Exit
	CANCEL	Cancel
	PRINT	Print
	SEARCH	Find
	SEARCH+	Find next
	PAGE_UP	Previous page
	PAGE_DOWN	Next page
	BOTTOM	Last page
	CANCEL	Cancel

Application toolbar

Position	Function code	Text
1	SELE	
2	MARK	
3	DELETE	
4	FUNCT_F5	Function 1
5	FUNCT_F6	Function 2

function keys

Key	Function code	Text	Icon
F2	SELE	Choose	
F9	MARK	Select	
Shift+F2	DELETE	Delete	
F5	FUNCT_F5	Function 1	
F6	FUNCT_F6	Function 2	
F7	FUNCT_F7	Function 3	
F8	FUNCT_F8	Function 4	

All function codes exist as menu entries, but not all of them are assigned to function keys. The function codes in the standard toolbar activate the icons and are automatically assigned to their function keys. Some of the additional function keys are assigned to the application toolbar.

In the module USER_COMMAND_100, the contents of the OK_CODE field are copied to the auxiliary variable SAVE_OK code, and OK_CODE is initialized. You should always do this, since it ensures that the screen field OK_CODE has always been initialized before the PBO event and can therefore not unintentionally contain an incorrect value. The function code of the function chosen by the user is then assigned to the OUTPUT field and displayed in the corresponding screen field (but not if the user chose BACK, EXIT, or CANCEL, in which cases, the program ends).

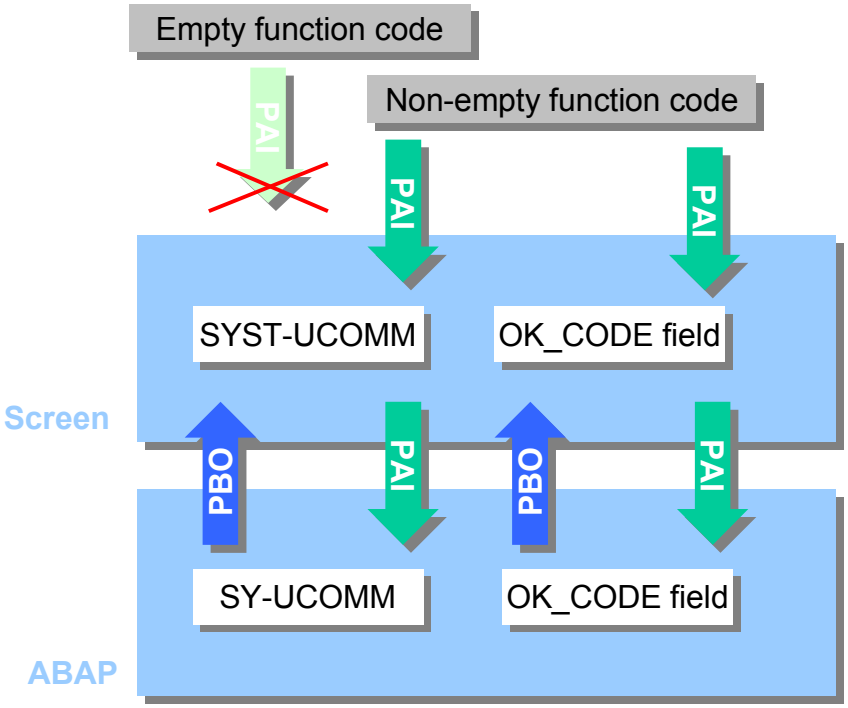
The function code SELE can be passed to the ABAP program in the following ways:

- When you choose *Choose* in the *Edit* menu
- When you choose the  pushbutton in the application toolbar
- When you choose the F2 key on the keyboard
- When you click the right-hand mouse button and then choose *Choose*.
- When you double-click the screen field OUTPUT
- When you enter SELE in the command field and press ENTER.

The other function codes are passed according to their definition, but without the double-click function.

Reading Function Codes

In each PAI event that a user triggers by choosing either a pushbutton on the screen or an element in a GUI status, the corresponding function code is placed into the system field SYST-UCOMM or SY-UCOMM and placed in the OK_CODE field (as long as the function code is not empty). Empty function codes are placed in neither the SY-UCOMM field nor the OK_CODE field.



In your ABAP programs, you should work with the OK_CODE field instead of SY-UCOMM. There are two reasons for this: Firstly, the ABAP program has full control over fields declared within it, and secondly, you should never change the value of an ABAP system field. However, you should also always initialize the OK_CODE field in an ABAP program for the following reason:

In the same way that the OK_CODE field in the ABAP program and the system field SY-UCOMM receive the contents of the corresponding screen fields in the PAI event, their contents are also assigned to the OK_CODE screen field and system field SYST-UCOMM in the PBO event. Therefore, you must clear the OK_CODE field in the ABAP program to ensure that the function code of a screen is not already filled in the PBO event with an unwanted value. This is particularly important when the next PAI event can be triggered with an empty function code (for example, using ENTER). Empty function codes do not affect SY-UCOMM or the OK_CODE field, and consequently, the old field contents are transported.

In your application programs, the first step in PAI processing should be to save the function code in an auxiliary variable and then initialize the OK_CODE field. You can then read the function code from the auxiliary variable (for example, using a CASE structure), and control the program flow from there.

The OK_CODE field can have a different name on each screen. However, common practice is to use the same name for the field on each screen of an ABAP program. You then only need one

Reading Function Codes

field in the ABAP program, with the same name, into which the function code is placed, and from which you can read it.

Global data declarations:

```
DATA: OK_CODE LIKE SY-UCOMM,
      SAVE_OK LIKE SY-UCOMM.
```

Your ABAP program must contain a field with the same name as the OK_CODE field on the screen. To specify the type, you should refer to the system field SY-UCOMM, since this always corresponds to the type of the OK_CODE field on the screen. At the same time, you should declare an appropriate auxiliary variable.

PAI module:

```
MODULE USER_COMMAND_100 INPUT.

  SAVE_OK = OK_CODE.
  CLEAR OK_CODE.

  CASE SAVE_OK.
    WHEN...
   ...
  ENDCASE.

ENDMODULE.
```

In the first PAI module, you should assign the contents of the OK_FIELD to the auxiliary variable and then clear the OK_CODE field and carry on working with the auxiliary variable.

Finding Out the Cursor Position

After user interaction with the screen, you may need to know the position of the cursor when the action occurred. This is particularly important if the user chooses the Choose function (F2 or mouse double-click).

To find out the cursor position, use the following statement:

```
GET CURSOR FIELD <f> [OFFSET <off>]
                     [LINE <lin>]
                     [VALUE <val>]
                     [LENGTH <len>].
```

This statement transfers the name of the screen element on which the cursor is positioned during a user action into the variable <f>. If the cursor is on a field, the system sets SY-SUBRC to 0, otherwise to 4.

The additions to the GET CURSOR statement have the following functions:

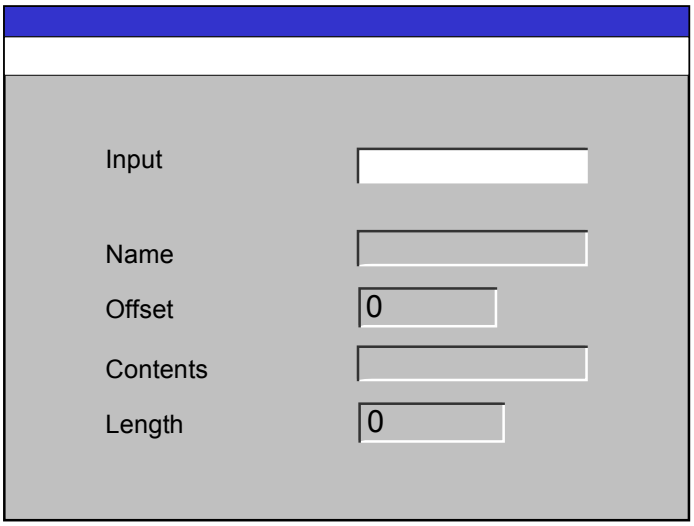
- OFFSET writes the cursor position within the screen element to the variable <off>.
- LINE writes the line number of the table to the variable <lin> if the cursor is positioned in a table control [Page 669]. If the cursor is not in a table control, <lin> is set to zero.
- VALUE writes the contents of the screen field in display format, that is, with all of its formatting characters, as a string to the variable <val>.
- LENGTH writes the display length of the screen field to the variable <len>.

Cursor position on the screen.

```
PROGRAM DEMO_DYNPRO_GET_CURSOR.

DATA: OK_CODE LIKE SY-UCOMM,
      SAVE_OK LIKE OK_CODE.

DATA: INPUT_OUTPUT(20) TYPE C,
      FLD(20) TYPE C,
      OFF     TYPE I,
      VAL(20) TYPE C,
      LEN     TYPE I.

CALL SCREEN 100.

MODULE INIT_SCREEN_0100 OUTPUT.
  SET PF-STATUS 'STATUS_100'.
ENDMODULE.

MODULE USER_COMMAND_0100 INPUT.
  SAVE_OK = OK_CODE.
  CLEAR OK_CODE.
  CASE SAVE_OK.
    WHEN 'CANCEL'.
      LEAVE PROGRAM.
    WHEN 'SELE'.
      GET CURSOR FIELD FLD OFFSET OFF VALUE VAL LENGTH LEN.
```

```
   ENDCASE.
ENDMODULE.
```

The next screen (statically defined) for screen 100 is itself. It has the following layout:



The relevant extract of the element list is as follows:

Name	Type	Format	Text	Function code
TEXT1	Text		Input	
INPUT_OUTPUT	I/O	CHAR		
TEXT2	Text		Name	
FLD	I/O	CHAR		
TEXT3	Text		Offset	
OFF	I/O	INT4		
TEXT4	Text		Contents	
VAL	I/O	CHAR		
TEXT5	Text		Length	
LEN	I/O	INT4		
EXIT_BUTTTON	Push		Cancel	CANCEL
OK_CODE	OK	OK		

The input attribute of all of the input/output fields except INPUT_OUTPUT is inactive.

The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE INIT_SCREEN_0100.

PROCESS AFTER INPUT.
  MODULE USER_COMMAND_0100.
```

The module INIT_SCREEN_100 sets the GUI status STATUS_100 during the PBO event. In the status, the cancel icon  (F12) is active with the function code CANCEL, and the function key F2 is active with the function code SELE.

When you run the program, the user can select any screen element by double-clicking it, or use any screen element connected to the function code SELE. The output fields on the screen return the cursor position.

Calling ABAP Dialog Modules

The main task of the screen flow logic [Page 532] is to call dialog modules in an ABAP program. You can do this using the MODULE statement, which can be programmed in the four possible event blocks of screen flow logic. In the PBO event, you can use this statement to call any dialog module in the ABAP program that has been defined using

MODULE <mod> OUTPUT.
...
ENDMODULE.

In the PAI, POH, and POV events, you can use the statement to call any dialog module in the ABAP program that has been defined using

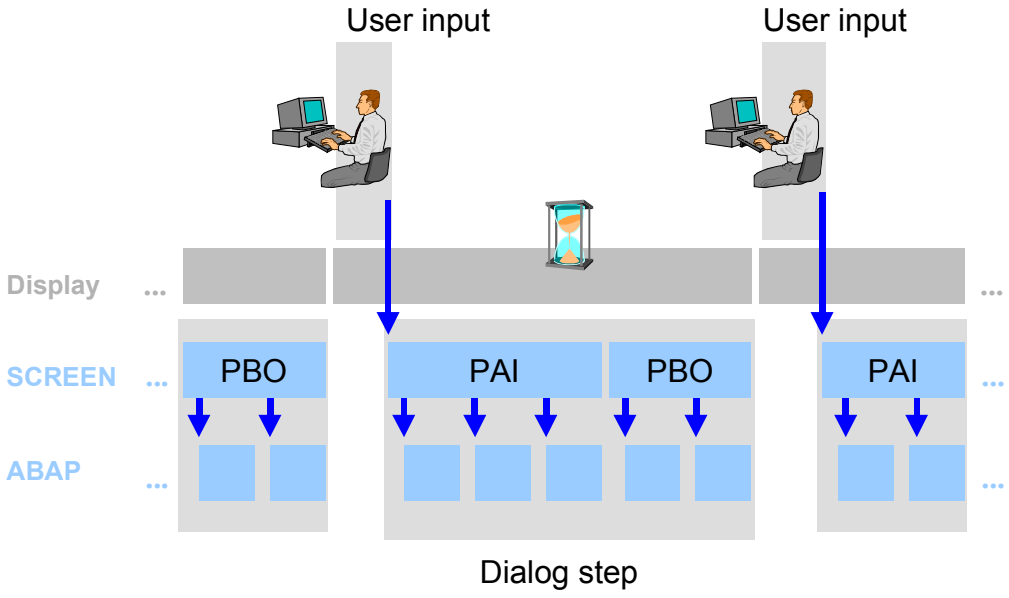
MODULE <mod> [INPUT].
...
ENDMODULE.

It is technically possible to have two dialog modules with the same name in the same program, one defined using OUTPUT, the other with INPUT. However, this is not recommended.

Since dialog modules in an ABAP program can be called from more than one screen, you can program functions that you need in several screens centrally in a single module. For example, it is usual to copy the contents of the OK-code field into an auxiliary variable and then reset the field in all PAI events. You could program this task in a single module that you would then call from the PAI event of all screens.

If you need to distinguish between screen numbers in a dialog module, you can use the system field SY-DYNNR, which always contains the number of the current screen. This would allow you, for example, to analyze the function code in a single PAI module and then control the further program flow according to the screen number and function code.

The actual screen sequence is as follows:



The screen display is prepared in the PBO event. The flow logic calls the corresponding ABAP dialog modules. Then, the screen is displayed. When this happens, control passes from the application server to the SAPgui. The screen accepts user input until the user triggers the PAI event. The control then returns to the application server. The program processes user input by calling the corresponding ABAP dialog modules. After the PAI processing is complete, the PBO processing of the next screen starts. The PAI processing and the PBO processing of the next screen form a single dialog step on the application server. The current screen remains visible until the PBO of the next screen is complete, but is not ready for input.

There are various ways of calling the dialog modules in the flow logic. The syntax of the flow logic allows you to call dialog modules conditionally, and to control the transfer of data between the screen and the ABAP program.

Simple Module Calls [Page 562]

Controlling the Data Transfer [Page 565]

Calling Modules Unconditionally [Page 568]

Calling Modules Conditionally [Page 572]

Simple Module Calls

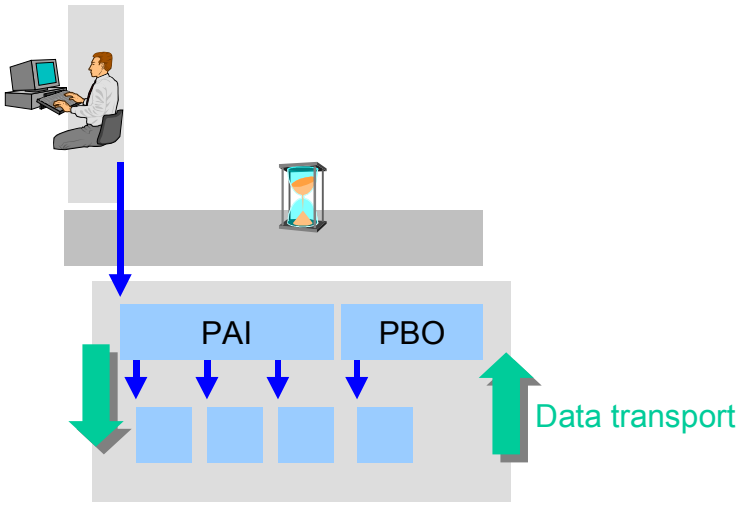
To call a module, use the flow logic statement

MODULE <mod>.

The system starts the module <mod>, which must have been defined for the same event block in which the call occurs.

If you only use simple modules in the screen flow logic, the data transport between the ABAP program and the screen is as follows:

- In the PAI event, all of the data from the screen is transported to the ABAP program (as long as there are program fields with the same names as the screen fields) after the automatic input checks [Page 577] and before the first PAI module is called. This includes the contents of the system fields (for example, SY-UCOMM, which contains the current function code).
- At the end of the last PBO module, and before the screen is displayed, all of the data is transported from the ABAP program to any identically-named fields in the screen.



Dialog step

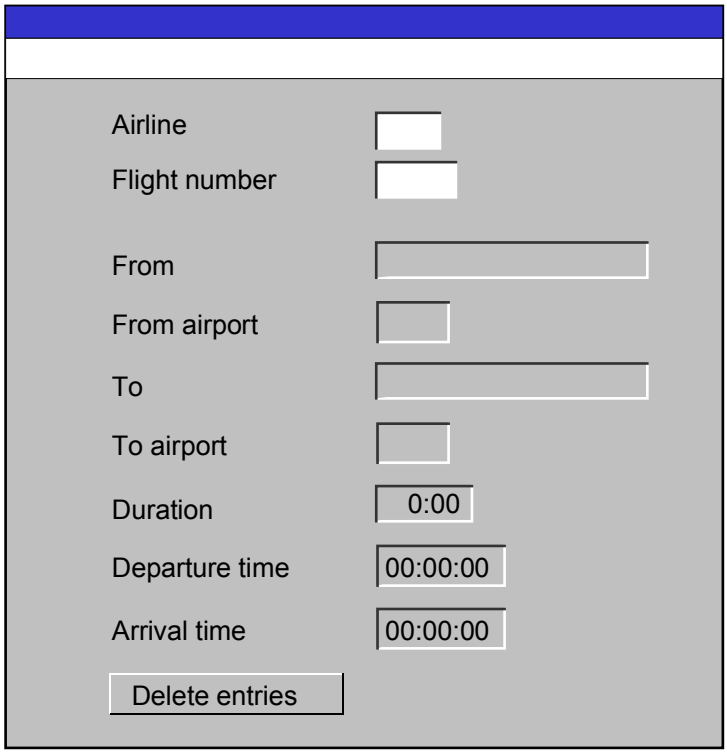
Data is transported between the screen and the ABAP program at the beginning and end of each dialog step on the application server. Do not confuse this with the data transport between a screen on the application server and the SAPgui on the presentation server.

Simple module call

```
PROGRAM DEMO_DYNPRO_MODULE.

TABLES SDYN_CONN.
```

```
DATA: OK_CODE LIKE SY-UCOMM,
      SAVE_OK LIKE OK_CODE,
      WA_SPFLI TYPE SPFLI.

CALL SCREEN 100.

MODULE INIT_SCREEN_100 OUTPUT.
  MOVE-CORRESPONDING WA_SPFLI TO SDYN_CONN.
ENDMODULE.

MODULE STATUS_0100 OUTPUT.
  SET PF-STATUS 'STATUS_100'.
  SET TITLEBAR '100'.
ENDMODULE.

MODULE CLEAR_OK_CODE INPUT.
  SAVE_OK = OK_CODE.
  CLEAR OK_CODE.
ENDMODULE.

MODULE GET_DATA INPUT.
  MOVE-CORRESPONDING SDYN_CONN TO WA_SPFLI.
  CLEAR SDYN_CONN.
ENDMODULE.

MODULE USER_COMMAND_0100 INPUT.
  CASE SY-DYNNR.
    WHEN 0100.
      CASE SAVE_OK.
        WHEN 'CANCEL'.
          LEAVE PROGRAM.
        WHEN 'DISPLAY'.
          PERFORM READ_DATA.
        WHEN 'CLEAR'.
          CLEAR WA_SPFLI.
      ENDCASE.
...
  ENDCASE.
ENDMODULE.

FORM READ_DATA.
  SELECT  SINGLE
          CITYFROM AIRPFROM CITYTO AIRPTO FLTIME DEPTIME ARRTIME
    INTO  CORRESPONDING FIELDS OF WA_SPFLI
    FROM  SPFLI
    WHERE CARRID = WA_SPFLI-CARRID AND CONNID = WA_SPFLI-CONNID.
ENDFORM.
```

The statically-defined next screen for screen 100 is 100. It uses components of the structure SDYN_CONN, copied [Ext.] from the ABAP Dictionary, and looks like this:



The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE INIT_SCREEN_100.
  MODULE STATUS_0100.

PROCESS AFTER INPUT.
  MODULE CLEAR_OK_CODE.
  MODULE GET_DATA.
  MODULE USER_COMMAND_0100.
```

In the GUI status STATUS_100, the symbol (F12) is active with the function code CANCEL, and the functions DISPLAY and CLEAR are assigned to the function keys F5 and shift+F2 respectively.

The program has a similar function to the example program in the section Processing Input/Output Fields [Page 537].

Controlling the Data Transfer

Data is passed from screen fields to ABAP fields with the same names once in each dialog step. If you only use simple module calls, all of the data is transferred in the PAI event before PAI processing starts.

The FIELD statement in the screen flow logic allows you to control the moment at which data is passed from screen fields to their corresponding ABAP fields.

To specify this point, use the following statement in the PAI flow logic:

FIELD <f>.

Data is not transported from the screen field <f> into the ABAP field <f> until the FIELD statement is processed. If a field occurs in more than one FIELD statement, its value is passed to the program when the first of the statements is reached.

Only screen fields that do not appear in a FIELDS statement are transferred at the beginning of the PAI event. Do not use fields in PAI modules until they have been passed to the program from the screen, otherwise the ABAP field will contain the same value as at the end of the previous dialog step.

The exception to this are fields that were initial in the PBO event and are not changed by the user. These are **not** transported by the FIELD statement. If a field of this type is filled with a value in a PAI module before its corresponding FIELD statement is executed, any value that you assign to it is not overwritten.

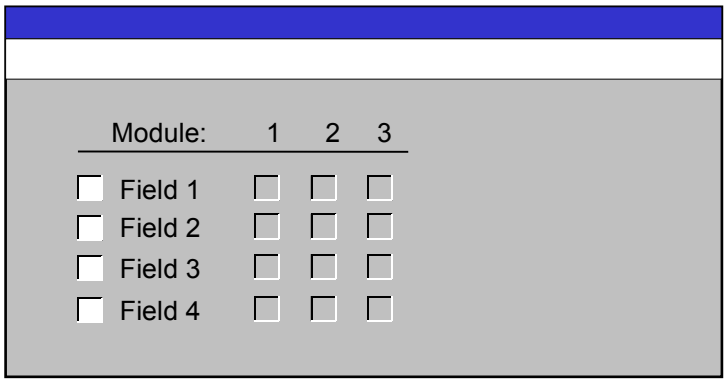
The FIELD statement has further functions in connection with conditional module calls [Page 572] and validity checks [Page 577] .

Controlling the data transfer

```
PROGRAM DEMO_DYNPRO_FIELD_CHAIN.

DATA: OK_CODE LIKE SY-UCOMM,
      SAVE_OK LIKE OK_CODE,
      BOX1, BOX2, BOX3, BOX4,
      MOD1_RESULT1, MOD1_RESULT2, MOD1_RESULT3, MOD1_RESULT4,
      MOD2_RESULT1, MOD2_RESULT2, MOD2_RESULT3, MOD2_RESULT4,
      MOD3_RESULT1, MOD3_RESULT2, MOD3_RESULT3, MOD3_RESULT4.

CALL SCREEN 100.

MODULE INIT_SCREEN_100 OUTPUT.
  SET PF-STATUS 'STATUS_100'.
  CLEAR: BOX1, BOX2, BOX3, BOX4.
ENDMODULE.

MODULE USER_COMMAND_0100 INPUT.
  SAVE_OK = OK_CODE.
  CLEAR OK_CODE.
  IF SAVE_OK = 'CANCEL'.
    LEAVE PROGRAM.
  ENDIF.
ENDMODULE.
```

```
MODULE MODULE_1 INPUT.
  MOD1_RESULT1 = BOX1.
  MOD1_RESULT2 = BOX2.
  MOD1_RESULT3 = BOX3.
  MOD1_RESULT4 = BOX4.
ENDMODULE.

MODULE MODULE_2 INPUT.
  MOD2_RESULT1 = BOX1.
  MOD2_RESULT2 = BOX2.
  MOD2_RESULT3 = BOX3.
  MOD2_RESULT4 = BOX4.
ENDMODULE.

MODULE MODULE_3 INPUT.
  MOD3_RESULT1 = BOX1.
  MOD3_RESULT2 = BOX2.
  MOD3_RESULT3 = BOX3.
  MOD3_RESULT4 = BOX4.
ENDMODULE.
```

The next screen (statically defined) for screen 100 is itself. It has the following layout:



The screen fields BOX1, BOX2, BOX3, and BOX4 are assigned to the checkboxes on the left that are ready for input.

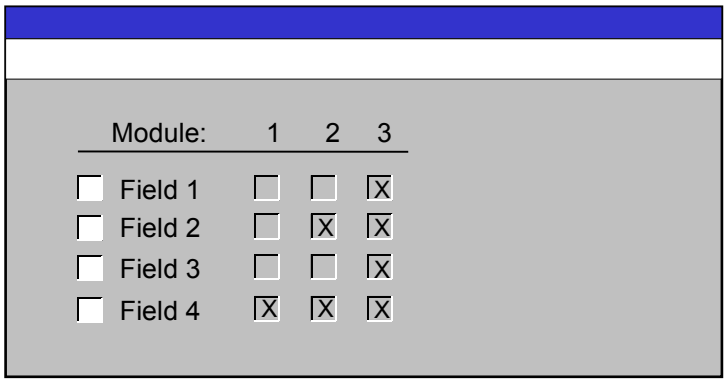
The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE INIT_SCREEN_100.

PROCESS AFTER INPUT.
  MODULE USER_COMMAND_0100.
  MODULE MODULE_1.
  FIELD BOX2.
  MODULE MODULE_2.
  FIELD: BOX1, BOX3.
  MODULE MODULE_3.
```

In the GUI status STATUS_100, the icon  (F12) is active with function code CANCEL.

When the user selects the checkboxes and chooses ENTER to trigger the PAI event, the output fields show the dialog modules in which each screen field is available.

If all of the checkboxes are selected, the result is:



- The screen field BOX4 is transported in the PAI event, since it does not occur in any FIELD statements.
- BOX2 is not transported until before the dialog module MODULE_2, and is therefore not available in USER_COMMAND_0100 or MODULE_1.
- BOX1 and BOX3 are transported before dialog module MODULE_3, and are therefore only available in that module.

Calling Modules Unconditionally

In the PAI event, the PAI modules are called in the sequence in which they occur in the screen flow logic, after the automatic field checks [Page 577]. This means that the input on the screen must satisfy the automatic checks before the first module can be called. In particular, all required fields must be filled, and any checks against value lists or check tables defined for the field in the ABAP Dictionary must be successful.

In some cases, the user may have to enter a considerable amount of data merely in order to be able to leave the screen. To avoid this, you can use special function codes with a special module call, which calls the module regardless of what the user enters on the screen.

Type E Function Codes

You can assign the function type E to the function codes of both pushbuttons on the screen and of elements in the GUI status. To do this for a pushbutton, set the *Function type* attribute in the Screen Painter to E. To do it in the GUI status, choose *Goto* → *Object lists* → *Function list* in the Menu Painter, select the required function codes, and enter E for the *function type*.

If the user chooses a pushbutton or a function in the status, the system bypasses the automatic field checks and calls a special module in the screen flow logic. If the special module call does not exist, the system resumes normal PAI processing, that is, the automatic field checks take place after all.

As a rule, type E functions should allow the user to leave the screen. Consequently, the function codes for Back (F3), Exit (Shift + F3), and Cancel (F12) usually have type E.

Calling a PAI Module for Type E Functions

When the user chooses a function with type E, the screen flow logic jumps directly to the following statement:

MODULE <mod> AT EXIT-COMMAND.

Regardless of where it occurs in the screen flow logic, this statement is executed immediately, and before the automatic checks for the field contents on the screen. Before the module <mod> is executed, the contents of the OK-CODE field are transported to the ABAP field with the same name. However, **no other screen fields** are transported to the program at this stage. If you have more than one MODULE statement with the AT EXIT-COMMAND addition, **only the first** is executed. If there are no MODULE statements with the AT EXIT-COMMAND statement, normal PAI processing resumes.

If the user chooses a function whose function code does not have type E, the MODULE <mod> AT EXIT-COMMAND statement is not executed.

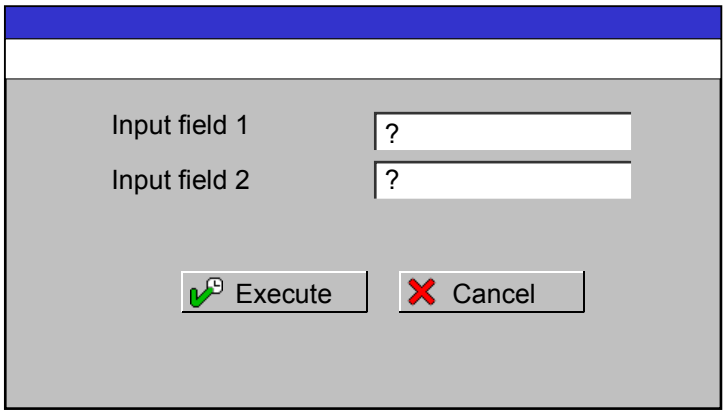
PAI Modules for Type E Functions

The MODULE ... AT EXIT-COMMAND statement is normally used to leave the current screen without the automatic input checks taking place. You should therefore program it to contain an appropriate variant of the LEAVE statement, to leave the current screen, the call chain, or the entire program, as appropriate. If the module does not leave the screen, normal PAI processing resumes after it has finished, that is, the automatic field checks take place, and the normal PAI modules are called, with data being transported from the screen back to the program according to the sequence defined in the FIELDS statements.

Unconditional module call

```
PROGRAM DEMO_DYNPRO_AT_EXIT_COMMAND.

DATA: OK_CODE LIKE SY-UCOMM,
      SAVE_OK LIKE OK_CODE,
      INPUT1(20), INPUT2(20).

CALL SCREEN 100.

MODULE INIT_SCREEN_0100 OUTPUT.
  SET PF-STATUS 'STATUS_100'.
ENDMODULE.

MODULE CANCEL INPUT.
  MESSAGE I888(BCTRAIN) WITH TEXT-001 OK_CODE INPUT1 INPUT2.
  IF OK_CODE = 'CANCEL'.
    CLEAR OK_CODE.
    LEAVE PROGRAM.
  ENDIF.
ENDMODULE.

MODULE BACK INPUT.
  MESSAGE I888(BCTRAIN) WITH TEXT-002 OK_CODE INPUT1 INPUT2.
  IF OK_CODE = 'BACK'.
    CLEAR: OK_CODE, INPUT1, INPUT2.
    LEAVE TO SCREEN 100.
  ENDIF.
ENDMODULE.

MODULE EXECUTE1 INPUT.
  MESSAGE I888(BCTRAIN) WITH TEXT-003 OK_CODE INPUT1 INPUT2.
  SAVE_OK = OK_CODE.
  CLEAR OK_CODE.
ENDMODULE.

MODULE EXECUTE2 INPUT.
  MESSAGE I888(BCTRAIN) WITH TEXT-004 OK_CODE INPUT1 INPUT2.
  IF SAVE_OK = 'EXECUTE'.
    MESSAGE S888(BCTRAIN) WITH TEXT-005.
  ENDIF.
ENDMODULE.
```

The next screen (statically defined) for screen 100 is itself. It has the following layout:



The input fields have the names INPUT1 and INPUT2, and are obligatory fields. The function codes of the pushbuttons are EXECUTE and CANCEL. CANCEL has function type E.

In the GUI status STATUS_100, the back (F3) and cancel (F12) icons are activated with the function codes BACK and CANCEL respectively. Both have the function type E. The function code EXECUTE is assigned to the function key F8. It does not have function type E.

The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE INIT_SCREEN_0100.

PROCESS AFTER INPUT.
  MODULE EXECUTE1.
  MODULE CANCEL AT EXIT-COMMAND.
  MODULE BACK AT EXIT-COMMAND.
  MODULE EXECUTE2.
```

The program uses information and status messages to show which modules are called following user interaction and which data is transported.

- If the user chooses *Execute* without filling out the obligatory fields, the automatic field check displays an error message.
- If the user fills out the obligatory fields and then chooses Execute, all of the screen fields are transported to the program, and the modules EXECUTE1 and EXECUTE2 are called.
- If the user chooses *Cancel*, the OK_CODE field is transported and the CANCEL module is called, regardless of whether the user filled out the obligatory fields. The CANCEL module terminates the program.
- If the user chooses *Back*, the OK_CODE field is transported and the CANCEL module is called, regardless of whether the user filled out the obligatory fields. However, the program does not terminate, since the function code is BACK. Instead, the automatic field checks are performed. If the obligatory fields are filled, the modules EXECUTE1 and EXECUTE2 are called.

  The BACK module is never called, since only the first module with the AT EXIT-COMMAND addition is ever called. In the above example, the function code BACK

should be processed in the CANCEL module. Then, since there is only one module statement with the AT EXIT-COMMAND addition, the position of the statement is irrelevant.

Conditional Module Calls

Simple module calls are processed in the sequence in which they appear in the screen flow logic. However, the syntax of the screen language also allows you to make PAI module calls dependent on certain conditions by using the MODULE statement together with the FIELD statement. You can apply conditions to both single fields and groups of fields. Conditional module calls can help you to reduce the runtime of your program, particularly with modules that communicate with database tables.

Conditions for Single Screen Fields

You can ensure that a PAI module is only called when a certain condition applies by using the following statement:

FIELD <f> MODULE <mod> ON INPUT|REQUEST|*-INPUT.

The additions have the following effects:

- ON INPUT

  The ABAP module is called only if the field contains a value other than its initial value. This initial value is determined by the data type of the field: Space for character fields, zero for numeric fields. Even if the user enters the initial value of the screen as the initial value, the module is not called. (ON REQUEST, on the other hand, does trigger the call in this case.)

- ON REQUEST

  The module <mod> is only called if the user has entered something in the field. This includes cases when the user overwrites an existing value with the same value, or explicitly enters the initial value.

  In general, the ON REQUEST condition is triggered through any form of “manual input”. As well as user input, the following additional methods of entering values also call the module:

  – The element attribute PARAMETER-ID (SPA/GPA parameters).

  – The element attribute HOLD DATA

  – CALL TRANSACTION ... USING

  – Automatic settings of particular global fields

- ON *-INPUT

  The ABAP module is called if the user has entered a “*” in the first character of the field, and the field has the attribute **-entry* in the Screen Painter. When the input field is passed to the program, the * is removed. * behaves like an initial field in the ON INPUT condition.

The functions of the FIELD statement for controlling data transport [Page 565] also apply when you use MODULE.

Conditions for Multiple Screen Fields

To ensure that one or more PAI modules are only called when several screen fields meet a particular condition, you must combine the calls in the flow logic to form a processing chain. You define processing chains as follows:

```
CHAIN.
 ...
ENDCHAIN.
```

All flow logic statements between CHAIN and ENDCHAIN belong to a processing chain. The fields in the various FIELD statements are combined, and can be used in shared conditions.

```
CHAIN.
  FIELD: <f1>, <f2>,...
  MODULE <mod1> ON CHAIN-INPUT|CHAIN-REQUEST.
  FIELD: <g1>, <g2>,...
  MODULE <mod2> ON CHAIN-INPUT|CHAIN-REQUEST.
 ...
ENDCHAIN.
```

The additions ON CHAIN-INPUT and ON CHAIN-REQUEST work like the additions ON INPUT and ON REQUEST that you use for individual fields. The exception is that the module is called whenever at least one of the fields listed in a preceding FIELD statement within the chain meets the condition. So <mod1> is called when one of the fields $<f_i>$ meets the condition. <mod2> is called when one of the fields $<f_i>$ or $<g_i>$ meets the condition.

Within a processing chain, you can combine individual FIELD statements with a MODULE statement to set a condition for a single field within the chain:

```
CHAIN.
  FIELD: <f1>, <f2>,...
  FIELD  <f> MODULE <mod1> ON  INPUT|REQUEST|*-INPUT
                           |CHAIN-INPUT|CHAIN-REQUEST.
  MODULE <mod2> ON CHAIN-INPUT|CHAIN-REQUEST.
ENDCHAIN.
```

The module <mod1> is called when screen field <f> meets the specified condition for individual fields. <mod2> is called when one of the fields $<f_i>$ or <f> meets the condition. If you use the addition ON CHAIN-INPUT or ON CHAIN-REQUEST with FIELD <f>, the condition also applies to the entire chain and module <mod1> and <mod2> are both called.

In cases where you apply conditions to various combinations of screen fields, it is worth setting up a separate processing chain for each combination and calling different modules from within it.

The functions of the FIELD statement for controlling data transport [Page 565] also apply when you use processing chains. Within a processing chain, screen fields are not transported until the FIELD statement. Processing chains also have another function for the FIELDS statements that they contain. This is described in the section on validity checks [Page 577].

Calling Modules after Cursor Selection

You can specify that a module should only be called if the cursor is positioned on a particular screen element. To do this, use the statement

```
MODULE <mod> AT CURSOR-SELECTION.
```

The module <mod> is called whenever the function code of the user action is CS with function type S. If you use this statement, it is best to assign the function code CS to function key F2. This also assigns it to the mouse double-click.

The module is called in the sequence in which it occurs in the flow logic. It does not bypass the automatic input checks. Data is transported from screen fields in the order in which it is defined by the FIELD statements. The function code is empty, and neither SY-UCOMM nor the

Conditional Module Calls

OK_CODE field is affected. You can also combine this MODULE statement with the FIELD statement:

FIELD <f> MODULE <mod> AT CURSOR-SELECTION.

or, for more than one field:

```
CHAIN.
  FIELD: <f1>, <f2>,...
  MODULE <mod> AT CURSOR-SELECTION.
ENDCHAIN.
```

The module <mod> is only called if the cursor is positioned on an input/output field <f> or an input/output field $<f_i>$ in the processing chain. You can only apply this statement to input/output fields.

The call hierarchy of the different combinations is as follows:

- If a MODULE... AT CURSOR-SELECTION statement is executed that was combined with FIELD, a statement without FIELD is not executed.
- If a statement using FIELD appears more than once for the same screen field <f>, only the first statement is executed.
- If a statement without FIELD occurs more than once, only the last statement is executed.

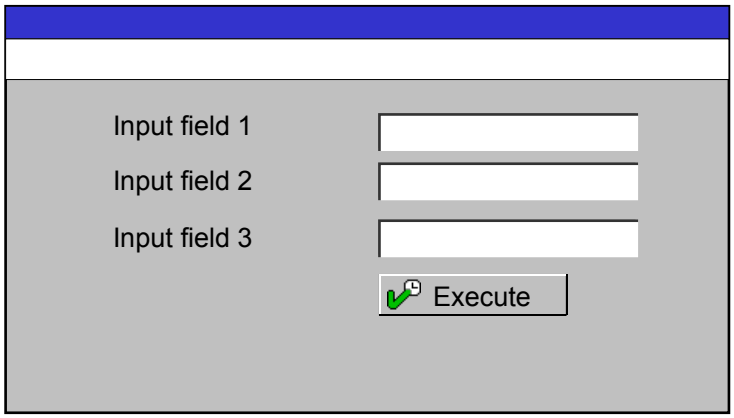
It is irrelevant whether the statements occur within a CHAIN ... ENDCHAIN block or not.

Conditional module calls

```
PROGRAM DEMO_DYNPRO_ON_CONDITION.

DATA: OK_CODE LIKE SY-UCOMM,
      INPUT1(20), INPUT2(20), INPUT3(20),
      FLD(20).

CALL SCREEN 100.

MODULE INIT_SCREEN_100 OUTPUT.
  SET PF-STATUS 'STATUS_100'.
ENDMODULE.

MODULE CANCEL INPUT.
  LEAVE PROGRAM.
ENDMODULE.

MODULE CURSOR INPUT.
  GET CURSOR FIELD FLD.
  MESSAGE I888(BCTRAIN) WITH TEXT-001 FLD.
ENDMODULE.

MODULE MODULE_1 INPUT.
  MESSAGE I888(BCTRAIN) WITH TEXT-002.
ENDMODULE.

MODULE MODULE_2 INPUT.
  MESSAGE I888(BCTRAIN) WITH TEXT-003.
ENDMODULE.
```

```
MODULE MODULE_* INPUT.
  MESSAGE I888(BCTRAIN) WITH TEXT-004 INPUT3.
ENDMODULE.

MODULE C1 INPUT.
  MESSAGE I888(BCTRAIN) WITH TEXT-005 '1'.
ENDMODULE.

MODULE C2 INPUT.
  MESSAGE I888(BCTRAIN) WITH TEXT-005 '2' TEXT-006 '3'.
ENDMODULE.
```

The next screen (statically defined) for screen 100 is itself. It has the following layout:



The screen fields INPUT1, INPUT2, and INPUT3 are assigned to the input fields. The function code of the pushbutton is EXECUTE.

In the GUI status STATUS_100, the icon (F12) is active with function code CANCEL and function type E. The function key F2 is also active with function code CS and function type S. The F8 key is active with the function code EXECUTE and no special function type.

The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE INIT_SCREEN_100.

PROCESS AFTER INPUT.
  MODULE CANCEL AT EXIT-COMMAND.
  CHAIN.
    FIELD: INPUT1, INPUT2.
    MODULE MODULE_1 ON CHAIN-INPUT.
    FIELD  INPUT3 MODULE MODULE_* ON *-INPUT.
    MODULE MODULE_2 ON CHAIN-REQUEST.
  ENDCHAIN.
  FIELD INPUT1 MODULE C1 AT CURSOR-SELECTION.
  CHAIN.
    FIELD: INPUT2, INPUT3.
    MODULE C2 AT CURSOR-SELECTION.
```

```
ENDCHAIN.
MODULE CURSOR AT CURSOR-SELECTION.
```

The program uses information messages to show which modules are called following user interaction and which data is transported.

- Whenever one of the input fields 1 or 2 is not initial, the system calls the module MODULE_1 for any user interaction.
- Whenever one of the three input fields is changed, the system calls the module MODULE_2 for any user interaction.
- Whenever input field 3 contains a * entry, the system calls module MODULE_* for any user interaction.
- If the user chooses F2 or double-clicks a text field on the screen, the system calls the module CURSOR.
- If the user chooses F2 or double-clicks input field 1, the system calls the module C1.
- If the user chooses F2 or double-clicks input field 2 or 3, the system calls the module CURSOR. Module C2 is never executed, since the MODULE ... AT CURSOR SELECTION statement occurs twice, and only the last is processed.

Input Checks

It is normally necessary to check user input for validity and consistency. There are three kinds of input checks on screens:

- Automatic Input Checks [Page 578]

  Automatic input checks are called in the PAI event before data is transported back to the ABAP program and before dialog modules are called.

- Input Checks in the Flow Logic [Page 581]

  Input checks in the flow logic can be performed before you call dialog modules.

- Input Checks in Dialog Modules [Page 584]

  These input checks are programmed in PAI modules. If the user enters an incorrect value, you can make input fields ready for input again without repeating the PBO processing.

Automatic Input Checks

In the PAI event, the screen makes a series of automatic input checks. These are carried out before any data is transferred to the ABAP program, and before the screen flow logic is processed. Before the automatic input checks, you can call a single dialog module unconditionally [Page 568] using a special function type. You normally use this dialog module to bypass the checks and leave the screen directly.

If the automatic input checks find an error, a message appears in the status bar of the screen, and the corresponding fields remain ready for input. The user must correct the entries and trigger the PAI again. The actual PAI processing does not start until there are no more errors.

The automatic input checks run in the following order:

Mandatory Fields

If a field is defined as a mandatory field in the Screen Painter, the user must enter a value for it before the PAI processing can start.

Input Format

The values entered in the input fields on the screen must correspond to the data format of the corresponding screen field. For example, the format of a date field (type DATS) is eight characters and has the format YYYYMMDD. All of the characters must be numeric. MM must be 12 or less, and DD must be 31 or less. The system also checks that the specified day is valid for the month.

ABAP Dictionary Checks

If you create an input field in the Screen Painter by copying [Ext.] a ABAP Dictionary field, the screen checks whether:

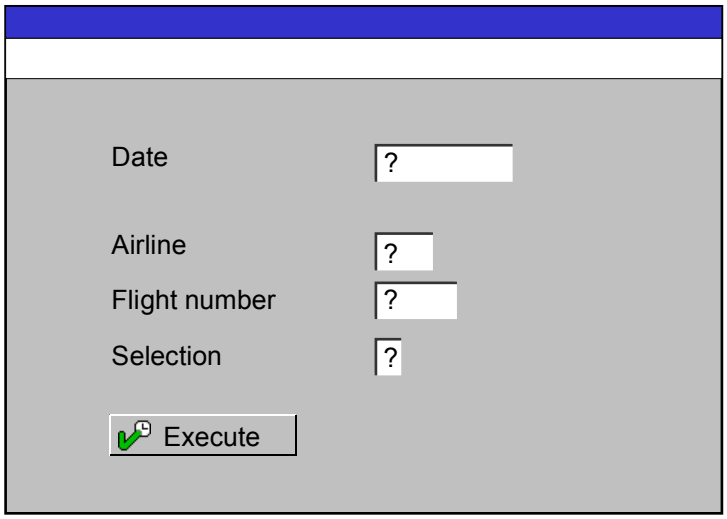
- The value satisfies any **foreign key relationship** to another database table, that is, the system checks whether the value is contained in the check table as a foreign key [Ext.]. This is only checked if the *Foreign key* attribute is set in the Screen Painter for the input field. The input check is not necessarily identical with the input help [Page 595]. The programmer is responsible for ensuring that the input help presents values that will all pass any foreign key check.
- That the value is one of the **fixed values** of the domain [Ext.] of the fields, that is, the system checks the definition of the underlying domain in the ABAP Dictionary. The fixed values of the domain can also be used as input help [Page 595]. However, the value table of a domain is not checked. It is only used as a default value for the check tables of the fields that refer to the domain.

Automatic input checks

```
PROGRAM DEMO_DYNPRO_AUTOMATIC_CHECKS.

DATA: OK_CODE LIKE SY-UCOMM,
      DATE TYPE D.

TABLES SDYN_CONN.
```

```
CALL SCREEN 100.

MODULE INIT_SCREEN_100 OUTPUT.
  SET PF-STATUS 'STATUS_100'.
ENDMODULE.

MODULE CANCEL INPUT.
  LEAVE PROGRAM.
ENDMODULE.

MODULE PAI INPUT.
  MESSAGE I888(BCTRAIN) WITH TEXT-001.
ENDMODULE.
```

The next screen (statically defined) for screen 100 is 100. It has the following layout:



The date field DATE from the program is assigned to the input field *Date*. The other input fields are the CARRID; CONNID, and MARK components of the ABAP Dictionary structure SDYN_CONN. All of the fields are mandatory. The function code of the pushbutton is EXECUTE.

In the GUI status STATUS_100, the icon (F12) is active with function code CANCEL and function type E. Furthermore, the function key F8 is assigned to the function code EXECUTE.

The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE INIT_SCREEN_100.

PROCESS AFTER INPUT.
  MODULE CANCEL AT EXIT-COMMAND.
  MODULE PAI.
```

The user must fill all of the input fields with valid values before the PAI module can be called:

- All of the input fields must contain values

Automatic Input Checks

- The date entry must have the correct format
- The airline must be in the check table SCARR
- The flight number must exist for the corresponding airline in the check table SPFLI
- The selection MARK must be one of the fixed values of the domain S_FLAG.

The user can leave the screen using Cancel ( , F12) without entering all values correctly, since the module call is programmed using AT EXIT-COMMAND.

Checking Fields in the Screen Flow Logic

There are two special variants of the FIELD statement that you can use in PAI processing to check the values of screen fields. However, this method is obsolete and should no longer be used. It is only supported for compatibility reasons.

Checking a Value List

You can check a screen field against a value list as follows:

FIELD <f> VALUES (<v_1>, <v_2>,...).

The individual entries <v_i> in the list can have the following format:

- [NOT] <val>
- [NOT] BETWEEN <val1> AND <val2>

You can check against single fields <val> or intervals between <val1> and <val2>. The comparison fields must have the data type CHAR or NUMC, must be enclosed in inverted commas, and must be written in uppercase. If the check fails, an error message is displayed and the corresponding field is again ready for entry.

The input help [Page 595] for <f> can also use the value list in the FIELD statement. This helps the user to enter only correct values.

The functions of the FIELD statement for controlling data transport [Page 565] also apply when you use the VALUES addition.

Checking Against Database Tables

You can check a screen field against the contents of a database table as follows:

```
FIELD <f> SELECT   *
          FROM  <dbtab>
          WHERE <k1> = <f1> AND <k2> = <f2> AND...
          [INTO  <g>]
          WHENEVER [NOT] FOUND SEND ERRORMESSAGE|WARNING
                           [<num> [WITH <h1>... <h4>]].
```

This combines the FIELD statement with a SELECT statement. The syntax of the SELECT statement must be entered exactly as shown above. In particular, the WHERE condition may not contain relational operators of the type ‘EQ’. In the WHERE condition, the fields of the primary key <k_i> of the database table <dbtab> are checked against the screen fields <f_i>. If a matching entry is found, you can write it into a screen field <g>, but you do not have to. You can also send an error or warning message depending on the outcome of the search, which makes the input field for <f> ready for input again.

<num> allows you to specify a message number from the message class specified in the MESSAGE-ID of the first statement in the program. In this case, the message class can only be **two characters** long. If you do not specify a message class, the system displays a default message. If you use WITH, you can pass literals or screen fields to any placeholderse (&) within the message.

The input help [Page 595] for <f> can also use the value list in the FIELD statement. This helps the user to enter only correct values.

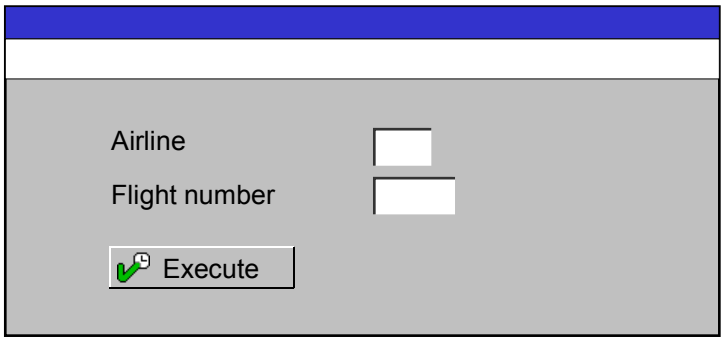
Checking Fields in the Screen Flow Logic

You can also use the SELECT statement in screen flow logic without combining it with the FIELD statement. This allows you to fill screen fields with the contents of database tables without calling ABAP modules. The functions of the FIELD statement for controlling data transport [Page 565] also apply when you use SELECT.

Checking Fields in the Screen Flow Logic

```
PROGRAM DEMO_DYNPRO_VALUE_SELECT MESSAGE-ID AT.

DATA: OK_CODE LIKE SY-UCOMM,
      CARRIER TYPE SPFLI-CARRID,
      CONNECT TYPE SPFLI-CONNID.

CALL SCREEN 100.

MODULE INIT_SCREEN_0100 OUTPUT.
  SET PF-STATUS 'STATUS_100'.
ENDMODULE.

MODULE CANCEL INPUT.
  LEAVE PROGRAM.
ENDMODULE.

MODULE MODULE_1 INPUT.
  MESSAGE I888(BCTRAIN) WITH TEXT-001 CARRIER
                TEXT-002 CONNECT.
ENDMODULE.

MODULE MODULE_2 INPUT.
  MESSAGE I888(BCTRAIN) WITH TEXT-001 CARRIER
                TEXT-002 CONNECT.
ENDMODULE.
```

The next screen (statically defined) for screen 100 is itself. It has the following layout:



The program fields CARRIER and CONNECT are assigned to the input fields. The function code of the pushbutton is EXECUTE.

In the GUI status STATUS_100, the icon  (F12) is active with function code CANCEL and function type E. Furthermore, the function key F8 is assigned to the function code EXECUTE.

The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE INIT_SCREEN_0100.

PROCESS AFTER INPUT.
  MODULE CANCEL AT EXIT-COMMAND.
  FIELD CARRIER VALUES (NOT 'AA', 'LH', BETWEEN 'QF' AND 'UA').
  MODULE MODULE_1.
  FIELD CONNECT SELECT  *
              FROM  SPFLI
              WHERE CARRID = CARRIER AND CONNID = CONNECT
              WHENEVER NOT FOUND SEND ERRORMESSAGE 107
                          WITH CARRIER CONNECT.
  MODULE MODULE_2.
```

The user must enter a value of CARRIER that is in the list following VALUES before MODULE_1 is called. When MODULE_1 is called, CONNECT has not yet been transported.

Next, the user can only enter a value for CONNECT that exists in the database table SPFLI as part of a primary key together with CARRIER. If not, error message 107 from message class AT is displayed in the status bar:

```
E: Unable to find any entries for key & &.
```

CONNECT is not transported, and module MODULE_2 not called until the user has entered a correct value.

Input Checks in Dialog Modules

You cannot perform input checks in PAI modules of programs until you have transported the contents of the input fields to the ABAP program. You can then use logical expressions [Page 225] to check the values that the user entered. You should then allow the user to correct any wrong entries before calling further modules.

You can do this by sending warning (type W) or error (type E) messages [Page 927] from PAI modules that are called in conjunction with the ABAP statements FIELD and CHAIN.

Checking Single Fields

If you send a warning or error message from a module <mod> that you called using a FIELD statement as follows:

FIELD <f> MODULE <mod>.

the corresponding input field on the current screen (and only this field) is made ready for input again, allowing the user to enter a new value. If the field is only checked once, the PAI processing continues directly after the FIELD statement, and the preceding modules are not called again.

Checking a Set of Fields

If you send a warning or error message from a module <mod> that you called using a FIELD statement as follows:

CHAIN.
  FIELD: $<f_1>$, $<f_2>$,...
  MODULE $<mod_1>$.
  FIELD: $<g_1>$, $<g_2>$,...
  MODULE $<mod_2>$.
...
ENDCHAIN.

all of the fields on the screen that belong to the processing chain (all of the fields listed in the field statements) are made ready for input again. Other fields are not ready for input. Whenever the MODULE statement appears within a processing chain, even if there is only one FIELD attached to it, all of the fields in the chain (not only the affected field) are made ready for input again, allowing the user to enter new values. If the fields in the processing chain are only checked once, the PAI processing continues directly after the FIELD statement, and the preceding modules are not called again.

Controlling Input and Data Transport

If you use the FIELD statement outside a chain, only a single field is made ready for input when a warning or error message is displayed. If you use it between the CHAIN and ENDCHAIN statements, it controls a set of fields. All of the fields controlled by a FIELD statement are transported back to the screen, **bypassing** PBO processing. This means that any changes made to the field contents before the message become visible on the screen. This also applies to information

Checking Fields Repeatedly

You may sometimes need to include the same field in more than one FIELD or CHAIN statement. If one of the corresponding modules sends a warning or error message, PAI processing resumes with the value that the user corrected. However, in this case, processing cannot simply resume at the corresponding FIELD or CHAIN statement if the field in question has already been included in an earlier FIELD or CHAIN statement.

Instead, all of the FIELD and CHAIN statements containing a field in which an error occurred are repeated. PAI processing resumes at the first FIELD or CHAIN statement containing a field in which an error occurred and that the user changed the last time the screen was displayed.

Example:

```
PROCESS AFTER INPUT.

  FIELD F1 MODULE M1.
  FIELD F2 MODULE M2.

  CHAIN.
    FIELD: F1, F2, F3.
    FIELD: F4, F5, F1.
    MODULE M3.
    MODULE M4.
  ENDCHAIN.

  CHAIN.
    FIELD: F6.
    MODULE M5.
  ENDCHAIN.

  CHAIN.
    FIELD F4.
    MODULE M6.
  ENDCHAIN.
```

If module M6 contains a warning or error message, the screen is displayed again, after which processing resumes with the first CHAIN statement and module M3, since this is the first occurrence of the field F4.

Remaining Functions in the FIELD Statement

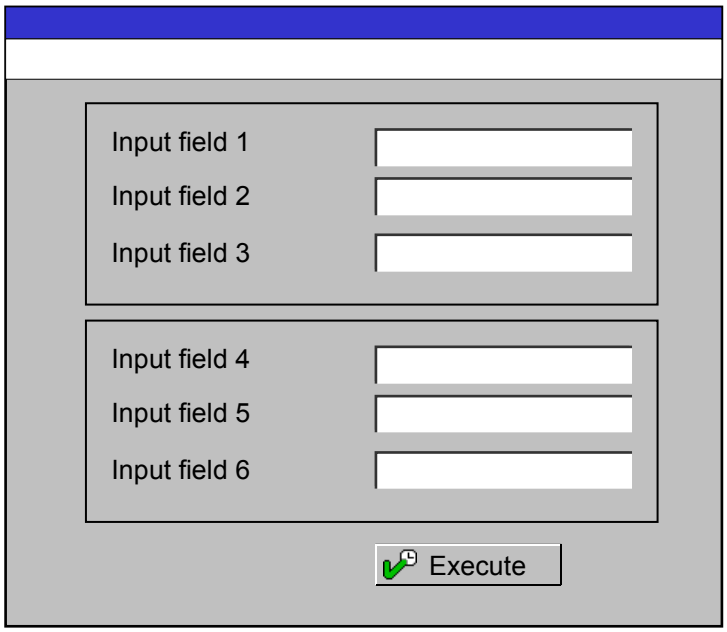
All of the functions of the FIELD and CHAIN statements for controlling data transport [Page 565] and conditional module calls [Page 572] can also be used in combination with warning and error messages. The contents of each field are transported at the FIELD statement in which the field occurs. If a warning or error message occurs in a conditional module of a processing chain, all of the fields in that chain will be ready for input when the screen is redisplayed, although not all of the fields will have been transported.

If a warning or error message occurs in a module that is not linked with a FIELD or CHAIN statement, none of the fields on the screen will be ready for input. In this case, the user can only exit the program, but only as long as you have included an unconditional module call [Page 568] in the program.

Input Checks in Dialog Modules

```
PROGRAM DEMO_DYNPRO_FIELD_CHAIN.

DATA: OK_CODE LIKE SY-UCOMM,
      INPUT1 TYPE I, INPUT2 TYPE I, INPUT3 TYPE I,
      INPUT4 TYPE I, INPUT5 TYPE I, INPUT6 TYPE I,
      SUM TYPE I.

CALL SCREEN 100.

MODULE INIT_SCREEN_100 OUTPUT.
  CLEAR: INPUT1, INPUT2, INPUT3, INPUT4, INPUT5, INPUT6.
  SET PF-STATUS 'STATUS_100'.
ENDMODULE.

MODULE CANCEL INPUT.
  LEAVE PROGRAM.
ENDMODULE.

MODULE MODULE_1 INPUT.
  IF INPUT1 < 50.
    MESSAGE E888(BCTRAIN) WITH TEXT-001 '50' TEXT-002.
  ENDIF.
ENDMODULE.

MODULE MODULE_2 INPUT.
  IF INPUT2 < 100.
    MESSAGE E888(BCTRAIN) WITH TEXT-001 '100' TEXT-002.
  ENDIF.
ENDMODULE.

MODULE MODULE_3 INPUT.
  IF INPUT3 < 150.
    MESSAGE E888(BCTRAIN) WITH TEXT-001 '150' TEXT-002.
  ENDIF.
ENDMODULE.

MODULE CHAIN_MODULE_1 INPUT.
  IF INPUT4 < 10.
    MESSAGE E888(BCTRAIN) WITH TEXT-003 '10' TEXT-002.
  ENDIF.
ENDMODULE.

MODULE CHAIN_MODULE_2 INPUT.
  CLEAR SUM.
  SUM = SUM + : INPUT4, INPUT5, INPUT6.
  IF SUM <= 100.
    MESSAGE E888(BCTRAIN) WITH TEXT-004 '100' TEXT-002.
  ENDIF.
ENDMODULE.

MODULE EXECUTION INPUT.
  MESSAGE I888(BCTRAIN) WITH TEXT-005.
ENDMODULE.
```

The next screen (statically defined) for screen 100 is itself. It has the following layout:



The screen fields INPUT1 to INPUT6 are assigned to the input fields. The function code of the pushbutton is EXECUTE.

In the GUI status STATUS_100, the icon (F12) is active with function code CANCEL and function type E. Furthermore, the function key F8 is assigned to the function code EXECUTE with the function type <blank>.

The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE INIT_SCREEN_100.

PROCESS AFTER INPUT.
  MODULE CANCEL AT EXIT-COMMAND.
  FIELD INPUT1 MODULE MODULE_1.
  FIELD INPUT2 MODULE MODULE_2.
  FIELD INPUT3 MODULE MODULE_3.
  CHAIN.
    FIELD INPUT4.
    MODULE CHAIN_MODULE_1.
    FIELD INPUT5.
    FIELD INPUT6 MODULE CHAIN_MODULE_2.
  ENDCHAIN.
  MODULE EXECUTION.
```

This program demonstrates how you can check input fields in dialog modules.

The fields INPUT1 to INPUT3 are checked independently of each other in the modules MODULE_1 to MODULE_3. As long as the user does not enter a corresponding value, the screen is repeatedly displayed with the appropriate field ready for input.

The fields INPUT4 to INPUT6 are checked together in a processing chain. If INPUT4 does not satisfy the condition in CHAIN_MODULE_1, all three fields are again ready for input. The same applies if the three fields do not satisfy the condition in CHAIN_MODULE_2.

The EXECUTION module, from which an information message is displayed, is not executed until all six fields satisfy the appropriate conditions.

Field Help, Input Help, and Dropdown Boxes

Feldhilfe [Page 590]

Field help (F1) and input help (F4) are standard functions throughout the system. You therefore cannot assign any other function codes to the F1 or F4 keys.

- If the user chooses the F1 key or the corresponding help icon (), a help text appears for the field in which the cursor is currently positioned.
- If the user chooses F4 or the input help button () to the right of a screen field, a list of possible entries appears for the cursor in which the cursor is currently positioned. The user can then choose one or more values, which are then copied into the screen field.

There are various ways of making field and input help available to users. For example, you can either use ABAP Dictionary functions, or program your own help functions.

Another special way to display lists of values is to use a dropdown box.

Field Help [Page 590]

Input Help [Page 595]

Dropdown Boxes [Page 607]

Field Help

There are three ways of displaying field help for screen elements:

Data Element Documentation

If you place a field on the screen in the Screen Painter by copying [Ext.] a ABAP Dictionary field, the corresponding data element documentation from the ABAP Dictionary is automatically displayed when the user chooses field help (as long as the help has not been overridden in the screen flow logic).

For further information about creating data element documentation, refer to data elements [Ext.].

Data Element Supplement Documentation

If the data element documentation is insufficient, you can expand it by writing a data element supplement

Data element supplement documentation contains the heading *Definition*, as well as the following others:

- *Use*
- *Procedure*
- *Examples*
- *Dependencies*

To create data element supplement documentation for a screen, choose *Goto* → *Documentation* → *DE supplement doc.* from the element list of the screen. A dialog box appears in which the system proposes a number as the identified for the data element supplement. You can then enter help texts for the above headings using the SAPscript editor.

Data element supplement documentation created in this way is program- and screen-specific. Any data element supplement documentation created in the ABAP Dictionary with the same number is overridden by the screen-specific documentation. You can link existing data element supplement documentation created in the ABAP Dictionary with a screen field by using the table THLPF. To do this, crate a new row in THLPF containing the following data: Program name, screen name, field name, and number of the data element supplement documentation.

To display data element supplement documentation, you must code the following screen flow logic in the POH event:

```
PROCESS ON HELP-REQUEST.
...
  FIELD <f> [MODULE <mod>] WITH <num>.
...
```

After PROCESS ON HELP-REQUEST, you can only use FIELD statements. If there is no PROCESS ON HELP-REQUEST keyword in the flow logic of the screen, the data element documentation for the current field, or no help at all is displayed when the user chooses F1. Otherwise, the next FIELD statement containing the current field <f> is executed.

If there is screen-specific data element supplement documentation for the field <f>, you can display it by specifying its number <num>. The number <num> can be a literal or a variable. The variable must be declared and filled in the corresponding ABAP program.

You can fill the variables, for example, by calling the module <mod> before the help is displayed. However, the FIELD statement does **not** transport the contents of the screen field <f> to the ABAP program in the PROCESS ON HELP-REQUEST event.

For further information about data element supplement documentation, refer to Data Element Supplements [Ext.].

Calling Help Texts from Dialog Modules

If data element supplement documentation is insufficient for your requirements, or you want to display help for program fields that you have not copied from the ABAP Dictionary, you can call dialog modules in the POH event:

```
PROCESS ON HELP-REQUEST.
...
  FIELD <f> MODULE <mod>.
...
```

After the PROCESS ON HELP-REQUEST statement, you can only use the MODULE statement together with the FIELD statement. When the user chooses F1 for a field <f>, the system calls the module <mod> belonging to the FIELD <f> statement. If there is more than one FIELD statement for the same field <f>, only the first is executed. However, the contents of the screen field <f> are not available in the module <mod>, since it is **not** transported by the FIELD statement during the PROCESS ON HELP-REQUEST event. The field help should not be dependent on the user input.

The module <mod> is defined in the ABAP program like a normal PAI module. The processing logic of the module must ensure that adequate help is displayed for the field in question. Instead of calling an extra screen with text fields, you should use one of the following function modules to display a suitable SAPscript document:

- HELP_OBJECT_SHOW_FOR_FIELD

  This function module displays the data element documentation for components of any structure or database table from the ABAP Dictionary. You pass the name of the component and structure or table to the import parameters FIELD and TABLE.

- HELP_OBJECT_SHOW

  Use this function module to display any SAPscript document. You must pass the document class (for example, TX for general texts, DE for data element documentation) and the name of the document to the import parameters DOKCLASS and DOKNAME. For technical reasons, you must also pass an empty internal table with the line type TLINE to the tables parameter of the function module.

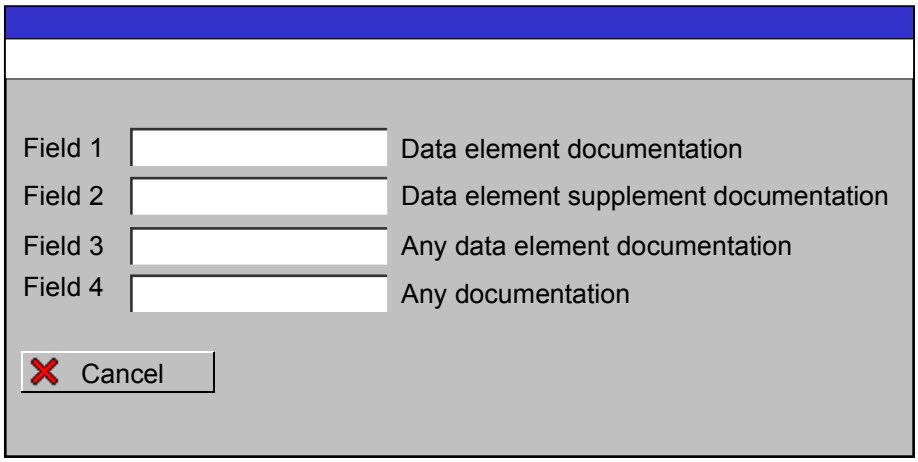
For further information about how to create SAPscript documents, refer to the Documentation of System Objects [Ext.] documentation.

Field help on screens.

```
REPORT DEMO_DYNPRO_F1_HELP.

DATA:  TEXT(30),
       VAR(4),
       INT TYPE I,
```

```
      LINKS TYPE TABLE OF TLINE,
      FIELD3, FIELD4.

TABLES DEMOF1HELP.

TEXT = TEXT-001.

CALL SCREEN 100.

MODULE CANCEL INPUT.
  LEAVE PROGRAM.
ENDMODULE.

MODULE F1_HELP_FIELD2 INPUT.
  INT = INT + 1.
  CASE INT.
    WHEN 1.
    VAR = '0100'.
    WHEN 2.
    VAR = '0200'.
    INT = 0.
  ENDCASE.
ENDMODULE.

MODULE F1_HELP_FIELD3 INPUT.
  CALL FUNCTION 'HELP_OBJECT_SHOW_FOR_FIELD'
      EXPORTING
          DOKLANGU                = SY-LANGU
          DOKTITLE                = TEXT-002
          CALLED_FOR_TAB          = 'DEMOF1HELP'
          CALLED_FOR_FIELD        = 'FIELD1'.
ENDMODULE.

MODULE F1_HELP_FIELD4 INPUT.
  CALL FUNCTION 'HELP_OBJECT_SHOW'
      EXPORTING
          DOKCLASS                = 'TX'
          DOKLANGU                = SY-LANGU
          DOKNAME                 = 'DEMO_FOR_F1_HELP'
          DOKTITLE                = TEXT-003
      TABLES
          LINKS                   = LINKS.
ENDMODULE.
```

The next screen (statically defined) for screen 100 is 100. It has the following layout:



The screen fields DEMOf1HELP-FIELD1 and DEMOF1HELP-FIELD2 from the ABAP Dictionary and the program fields FIELD3 and FIELD4 are assigned to the input fields. The pushbutton has the function code CANCEL with function type E.

The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.

PROCESS AFTER INPUT.
  MODULE CANCEL AT EXIT-COMMAND.

PROCESS ON HELP-REQUEST.
  FIELD DEMOF1HELP-FIELD2 MODULE F1_HELP_FIELD2 WITH VAR.
  FIELD FIELD3 MODULE F1_HELP_FIELD3.
  FIELD FIELD4 MODULE F1_HELP_FIELD4.
```

The components FIELD1 and FIELD2 of structure DEMOF1HELP both refer to the data element DEMOF1TYPE. This data element is documented, and also has two supplements with numbers 0100 and 0200.

The following field help is displayed:

- When the user chooses F1 on the input field for DEMOF1HELP-FIELD1, the data element documentation for DEMOF1TYPE is displayed, since the field does not occur in the PROCESS ON HELP-REQUEST event.
- If the user chooses F1 repeatedly for the input field DEMOF1HELP-FIELD2, the data element documentation is displayed, along with the supplement documentation for either 0100 or 0200 alternately. The variable VAR is filled in the dialog module F1_HELP_FIELD2.
- When the user chooses F1 on the input field for FIELD3, the data element documentation for DEMOF1TYPE is displayed, since this is called in the dialog module F1_HELP_FIELD3 by the function module HELP_OBJECT_SHOW_FOR_FIELD.
- When the user chooses F1 on the input field for FIELD4, the SAPscript documentation DEMO_FOR_F1_HELP is displayed, since this is called in the dialog module F1_HELP_FIELD4 by the function module HELP_OBJECT.

Input Help

One of the important features of screens is that they can provide users with lists of possible entries for a field. There are three techniques that you can use to create and display input help:

1. Definition in the ABAP Dictionary

   In the ABAP Dictionary, you can link search helps to fields. When you create screen fields with reference to ABAP Dictionary fields, the corresponding search helps are then automatically available. If a field has no search help, the ABAP Dictionary still offers the contents of a check table, the fixed values of the underlying domain, or static calendar or clock help.

2. Definition on a screen

   You can use the input checks of the screen flow logic, or link search helps from the ABAP Dictionary to individual screen fields.

3. Definition in dialog modules

   You can call ABAP dialog modules in the POV event of the screen flow logic and program your own input help.

These three techniques are listed in order of ascending priority. If you use more than one technique at the same time, the POV module calls override any definition on the screen, which, in turn, overrides the link from the ABAP Dictionary.

However, the order of preference for these techniques should be the order listed above. You should, wherever possible, use a search help from the ABAP Dictionary, and only use dialog modules when there is really no alternative. In particular, you should consider using a search help exit to enhance a search help before writing your own dialog modules.

Input Help from the ABAP Dictionary [Page 596]

Input Help on the Screen [Page 601]

Input Help in Dialog Modules [Page 603]

Input Help from the ABAP Dictionary

The main input help available from the ABAP Dictionary is in the form of search helps [Ext.]. Search helps are independent Repository objects that you create using the ABAP Dictionary. They are used to present input help for screen fields. You can use link search helps to table fields and data elements. As well as search helps, you can still, in exceptional cases, use check tables, fixed values, or static input help.

Input Help Methods

- Search helps

  There are two kinds of search helps: elementary and collective. An elementary search help [Ext.] represents a search path. It defines the location of the data for the hit list, how values are exchanged between the screen and the selection method, and the user dialog that occurs when the user chooses input help. A collective search help [Ext.] consists of two or more elementary search helps. Collective search helps cover a series of search paths for a field, all of which could be useful. The collective search help is the interface between the screen and the various elementary search helps.

- Check tables

  The ABAP Dictionary allows you to define relationships between tables using foreign keys [Ext.]. A dependent table is called a foreign key table, and the referenced table is called the check table. Each key field of the check table corresponds to a field in the foreign key table. These fields are called foreign key fields. One of the foreign key fields is designated as the check field for checking the validity of values. The key fields of the check table can serve as input help for the check field.

- Fixed values

  You can restrict the values that a domain [Ext.] in the ABAP Dictionary may take by assigning fixed values to it. The fixed values can be used as input help for the fields that are defined using that domain. However, the value table of a domain is not used for input help. It is only used as a default value for the check tables of the fields that refer to the domain.

- Static input help

  Fields with the types DATS and TIMS have their own predefined calendar and clock help that can be used as input help.

Input Help Hierarchy in the ABAP Dictionary

There are various ways of linking search helps with fields of database tables or components of structures. The input help available to a user depends on the type of link. The following list shows, in ascending order of priority, the input help that is used:

1. Calendar and clock help

   If no other input help has been defined for a field with type DATS or TIMS, the calendar or clock help is displayed.

2. Domain fixed values

   If a field has no check table or search help, any fixed values of the underlying domain are used.

3. Search help of the data element

   Search helps from the ABAP Dictionary can be attached to a data element [Ext.]. If a field has no check table or search help of its own, the system uses the search help assigned to the underlying data element.

4. Check table

   If the check table has no text table and no search help of its own, and a field has no search help of its own, the contents of the key fields of the check table are used as input help.

5. Check table with text table

   You can define a text table [Ext.] for a table. If the check table for a field has a text table defined for it, the input help displays both the key fields from the check table and the corresponding text from the text table in the user's logon language.

6. Check table with search help

   Search helps can be attached to the check table [Ext.] of a field. The search help is displayed with the values from the check table. However, it allows you to transfer the values of more than one parameter.

7. Search help for a field

   Search helps can also be directly attached to a field [Ext.] of a structure or a database table. This has the highest priority, and is always displayed for input help. When you attach the search help, you should therefore ensure that it only offers values that are also in the check table, otherwise errors may occur in the automatic input checks [Page 578].

Search Helps and Value Transport

Search helps have an interface (search help parameters), which determines which entries, already made by the user on the screen, should be used when compiling the hit list, and the screen fields that should be filled when the user chooses an entry. The parameters of a search help are divided into import and export parameters. Parameters can also be both import and export parameters simultaneously.

When the user starts the input help, the contents of the fields on the screen are passed to the import parameters of the search help. If a search help is assigned to a data element or directly to a screen field, only one search help parameter is assigned to the field, and values are only transported from the field to this parameter. If the search help is assigned to a table or structure field, or to the check table of the field, there may be more than one parameter of the search help that has to be filled with values. When the input help is started and the search help is assigned to a table or structure field, the system tries to find a field with the same name as each import parameter of the search help. Wherever it finds an identically-named field, the field contents are transferred to the search help parameter.

When the user selects a line from the hit list of the search help, the system transfers values from the export parameters of the search help to the corresponding screen fields. Values from the hit list are only returned to fields on the screen that are linked with an export parameter of the search help and are ready for input.

Changing Search Helps

In exceptional cases, you can modify the standard flow of an input help using a search help exit [Ext.]. A search help exit is a function module with a predefined interface. You can call it at

defined points within the input help process. The search help exit allows you to store your own program logic that either steers the subsequent processing or replaces it altogether.

The function module can change the attributes of the search help, the selection options that are used to preselect the hit list, the hit list itself, and also the subsequent processing steps.

All search help exits must have the same interface as the function module F4IF_SHLP_EXIT_EXAMPLE. However, you can define any number of additional optional parameters, especially exporting parameters. For further information about the interface, refer to the function module documentation.

If you have assigned a search help exit to a search help, it is called by the help processor at the following points. They are the points at which the user can interact with the input help, since these are the points at which you can best change the flow of the search help in the interests of the user:

1. Before the dialog box for selecting the search path is displayed.

   At the SELONE event (collective search helps only). This makes it possible to make the search help dependent on the transaction, on other system variables, or even on the state of the radio buttons on the screen. (This is the only event in which the search help exit is called for collective search helps. All other events call the search help exit for the selected elementary search help.)

2. After an elementary search help has been selected (event PRESEL1).

   Here, you can change the assignment of the search help to the screen by, for example, changing the way in which the search help parameters are assigned to screen fields.

3. Before the dialog box for entering search conditions is displayed.

   (PRESEL event). This enables you to change the contents of the dialog box, or to suppress it altogether.

4. Before data is selected

   (SELECT event). Although the value selection contains no user interaction, it can still be overridden either partially or fully by the search help exit. This may be necessary if it not possible to read the data using a SELECT statement for a table or view.

5. Before the hit list is displayed

   (DISP event). Here, you can affect how the hit list is displayed by, for example, suppressing certain lines or columns of the list depending on the authorizations of the user.

6. Before the line selected by the user is returned to the screen

   (RETURN event). This can make sense if the subsequent flow of the transaction is to depend on the value selected by the user. A typical use of this event would be to set SPA/GPA parameters.

Certain search help functions are requested repeatedly in similar ways. One example of this is the possibility to set the search help that will be used dynamically. Standard function modules have been written for these cases, which you can use either directly as search help exits, or call from within a search help exit. Such function modules all have the prefix F4UT_.

The Role of Domain Value Tables

Prior to Release 4.0, it was possible to use the value table of a domain to provide input help. This is no longer possible, primarily because unexpected results could occur if the value table had

more than one key field. It was not possible to restrict the other key fields, which meant that the environment of the field was not considered, as is normal with check tables.

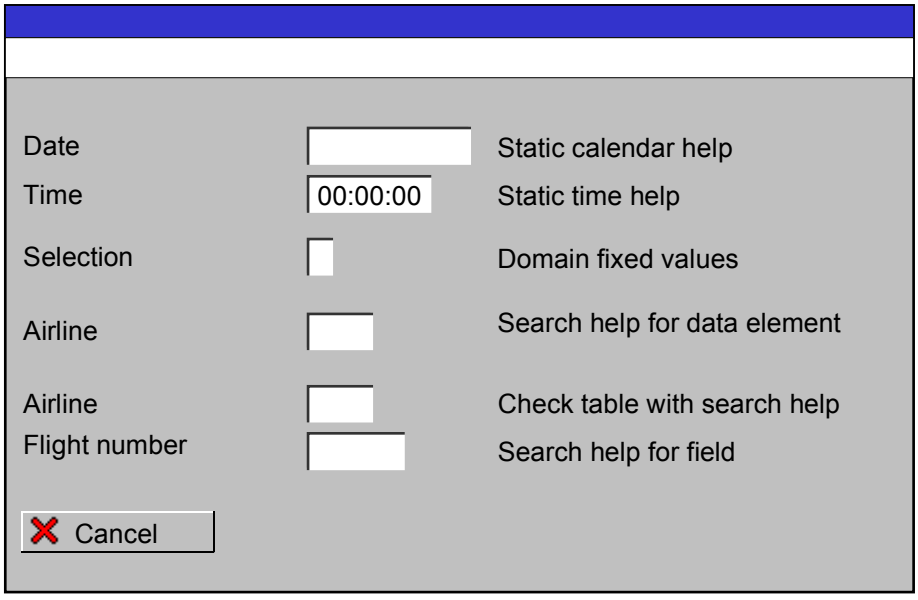
In cases where this kind of value help was appropriate, you can reconstruct it by creating a search help for the data elements that use the domain in question, and using the value table as the selection method.

Example

Input help from the ABAP Dictionary.

```
REPORT DEMO_DYNPRO_F4_HELP_DICTIONARY.

TABLES DEMOF4HELP.

CALL SCREEN 100.

MODULE CANCEL INPUT.
  LEAVE PROGRAM.
ENDMODULE.
```

The next screen (statically defined) for screen 100 is itself. It has the following layout:



The components of the ABAP Dictionary structure DEMOF4HELP are assigned to the input fields. The pushbutton has the function code CANCEL with function type E.

The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.

PROCESS AFTER INPUT.
  MODULE CANCEL AT EXIT-COMMAND.
```

When the user chooses input help for the individual fields, the following is displayed:

Input Help from the ABAP Dictionary

- The *Date* and *Time* fields refer to the components DATE_FIELD and TIME_FIELD respectively of the ABAP Dictionary structure. These have the data type DATS and TIMS, so the input help is a calendar and a clock respectively.
- The *Selection* field refers to the structure component MARK_FIELD. This has the underlying domain S_FLAG, which has two fixed values. These are displayed as the input help.
- The first *Airline* field refers to the structure component CARRIER1. This component has the underlying data element DEMOF4DE, to which the search help parameter CARRID of search help DEMO_F4_DE is assigned. The search help reads the columns CARRID and CARRNAME from the database table SCARR. Only CARRNAME is listed, but CARRID is flagged as an export parameter.
- The second *Airline* field refers to the structure component CARRIER2. This component has the check table SCARR. The check table, in turn, has the search help H_SCARR attached to it. This lists and exports the columns CARRID and CARRNAME.
- The *Flight number* field refers to the structure component CONNID. The search help DEMO_F4_FIELD is assigned to it. The search help has two parameters CARRID and CONNID, which are assigned to the components CARRIER2 and CONNID of the structure. The search help imports CARRIER, reads the **corresponding** data from the database table SPFLI, lists CARRIER and CONNID, and exports CONNID.

Input Help on the Screen

Within the Screen Painter, you can define two types of input help:

1. The FIELD statement with one of the additions VALUES or SELECT.
2. Linking a search help directly to a screen field.

If you link a search help directly to a screen field, it overrides the additions of the FIELD statement. However, the input check [Page 581] functions of the FIELD statement remain unaffected.

Input Help in Flow Logic

The following input help methods are obsolete and should not be used. They are still supported for compatibility reasons.

In the screen flow logic, you can specify a value list for a screen field <f> as follows:

FIELD <f> VALUES (<val_1>, <val_2>,...).

The value list contains a series of single values <val_i>. The NOT and BETWEEN additions for the input check [Page 581] are not appropriate for input help.

You can also create a value list by accessing a database table as follows:

FIELD <f> SELECT *
FROM <dbtab>
WHERE <k_1> = <f_1> AND <k_2> = <f_2> AND...

In the WHERE condition, the fields of the primary key <k_i> of the database table <dbtab> are checked against the screen fields <f_i>. The WHENEVER addition, used with input checks [Page 581], is not necessary for input help.

If you have used a ABAP Dictionary reference for field <f>, the selection and the hit list formatting may be affected by any check table attached to the field.

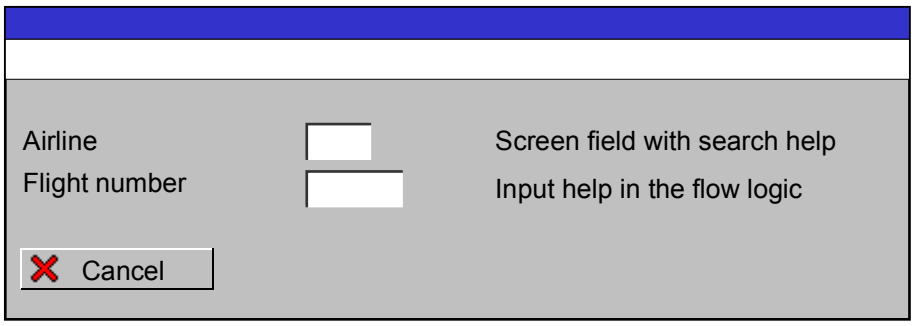
Attaching a Search Help

Search helps from the ABAP Dictionary can be attached to a screen field [Ext.]. To do this, enter the name of the search help in the corresponding field in the attributes of the screen field in the Screen Painter. This assigns the first parameter of the search help to the screen field. It is only possible to place a value from the hit list onto the screen.

Input help on a screen.

```
REPORT DEMO_DYNPRO_F4_HELP_DYNPRO MESSAGE-ID AT.

DATA: CARRIER(3) TYPE C,
      CONNECTION(4) TYPE C.

CALL SCREEN 100.

MODULE CANCEL INPUT.
  LEAVE PROGRAM.
ENDMODULE.
```

The next screen (statically defined) for screen 100 is itself. It has the following layout:



The input fields have been adopted from the program fields CARRIER and CONNECTION. The function code of the pushbutton is CANCEL, with function type E. The search help DEMO_F4_DE with the search help parameter CARRID is assigned to the screen field CARRIER. The search help uses the database table SCARR.

The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.

PROCESS AFTER INPUT.
  MODULE CANCEL AT EXIT-COMMAND.
  FIELD CARRIER VALUES ('AA', 'LH').
  FIELD CONNECTION SELECT *
                   FROM  SPFLI
                   WHERE CARRID = CARRIER
                     AND CONNID = CONNECTION.
```

When the user chooses the input help for the individual fields, the following input help is displayed:

- For the Airline field, the search help displays the names of the airlines and places the airline code in the input field for the chosen line. If the airline code is not one of those listed in the VALUES list of the screen flow logic, the input check triggers an error message in the PAI event. So the search help overrides the VALUES addition for the input help, but not for the input checks. This is therefore not an appropriate place to use the VALUE addition.
- For the Flight number field, the flow logic displays the selected entries from the database table SPFLI and places the selected line in the input field.

Input Help in Dialog Modules

You can call dialog modules in the POV event using the event keyword PROCESS ON VALUE-REQUEST.

```
PROCESS ON VALUE-REQUEST.
...
  FIELD <f> MODULE <mod>.
...
```

After the PROCESS ON VALUE-REQUEST statement, you can only use the MODULE statement together with the FIELD statement. When the user chooses F4 for a field <f>, the system calls the module <mod> belonging to the FIELD <f> statement. If there is more than one FIELD statement for the same field <f>, only the first is executed. The module <mod> is defined in the ABAP program like a normal PAI module. However, the contents of the screen field <f> are not available, since it is **not** transported by the FIELD statement during the PROCESS ON HELP-REQUEST event. You can now program your own value lists in the module. However, this procedure is only recommended if it really is not possible to use a search help. Defining search helps is much easier than PROCESS ON VALUE-REQUEST, since the system takes over some of the standard operations, such as getting field contents from the screen. It also ensures that the F4 help has a uniform look and feel throughout the system. Furthermore, it means that you do not have to reassign input help to fields on each screen.

Despite the introduction of search helps (and search help exits), there are still cases in which you need to use parts of the standard F4 functions directly. In this case, there are some standard function modules that you can use in the POV event. They support search helps, as well as all other kinds of input help, and are responsible for data transport between the screen and the input help. These all have the prefix F4IF_. The most important are:

- F4IF_FIELD_VALUE_REQUEST

  Calls the input help of the ABAP Dictionary dynamically. You can pass the component names of a structure or database table of the ABAP Dictionary to the function module in the import parameters TABNAME and FIELDNAME. The function module starts the ABAP Dictionary input help [Page 596] for this component. All of the relevant screen fields are read. If you specify the import parameters DYNPPROG, DYNPNR, and DYNPROFIELD, the user's selection is returned to the corresponding field on the screen. If you specify the table parameter RETURN_TAB, the selection is returned into the table instead.

- F4IF_INT_TABLE_VALUE_REQUEST

  This function module displays a value list that you created in an ABAP program. The value list is passed to the function module as the table parameter VALUE_TAB. If you specify the import parameters DYNPPROG, DYNPNR, and DYNPROFIELD, the user's selection is returned to the corresponding field on the screen. If you specify the table parameter RETURN_TAB, the selection is returned into the table instead.

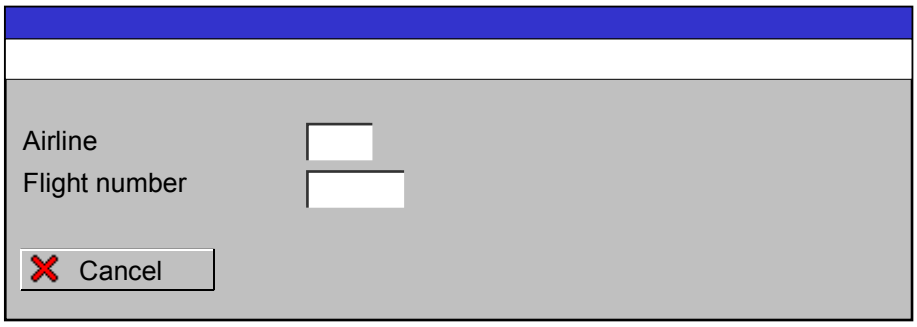
There are also two function modules - DYNP_VALUES_READ and DYNP_VALUES_UPDATE - that can read the values of screen fields and return values to them during the POV event. For further information, refer to the relevant function module documentation.

Input help in dialog modules

```
REPORT DEMO_DYNPRO_F4_HELP_MODULE.

TYPES: BEGIN OF VALUES,
         CARRID TYPE SPFLI-CARRID,
         CONNID TYPE SPFLI-CONNID,
       END OF VALUES.

DATA: CARRIER(3) TYPE C,
      CONNECTION(4) TYPE C.

DATA: PROGNAME LIKE SY-REPID,
      DYNNUM   LIKE SY-DYNNR,
      DYNPRO_VALUES TYPE TABLE OF DYNPREAD,
      FIELD_VALUE LIKE LINE OF DYNPRO_VALUES,
      VALUES_TAB TYPE TABLE OF VALUES.

CALL SCREEN 100.

MODULE INIT OUTPUT.
  PROGNAME = SY-REPID.
  DYNNUM   = SY-DYNNR.
  CLEAR: FIELD_VALUE, DYNPRO_VALUES.
  FIELD_VALUE-FIELDNAME = 'CARRIER'.
  APPEND FIELD_VALUE TO DYNPRO_VALUES.
ENDMODULE.

MODULE CANCEL INPUT.
  LEAVE PROGRAM.
ENDMODULE.

MODULE VALUE_CARRIER INPUT.

  CALL FUNCTION 'F4IF_FIELD_VALUE_REQUEST'
      EXPORTING
          TABNAME          = 'DEMOF4HELP'
          FIELDNAME        = 'CARRIER1'
          DYNPPROG         =  PROGNAME
          DYNPNR           =  DYNNUM
          DYNPROFIELD      = 'CARRIER'.

ENDMODULE.

MODULE VALUE_CONNECTION INPUT.

  CALL FUNCTION 'DYNP_VALUES_READ'
      EXPORTING
          DYNAME               = PROGNAME
          DYNUMB               = DYNNUM
          TRANSLATE_TO_UPPER   = 'X'
      TABLES
          DYNPFIELDS           = DYNPRO_VALUES.

  READ TABLE DYNPRO_VALUES INDEX 1 INTO FIELD_VALUE.

  SELECT  CARRID CONNID
    FROM  SPFLI
    INTO  CORRESPONDING FIELDS OF TABLE VALUES_TAB
    WHERE CARRID = FIELD_VALUE-FIELDVALUE.
```

```
CALL FUNCTION 'F4IF_INT_TABLE_VALUE_REQUEST'
   EXPORTING
      RETFIELD        = 'CONNID'
      DYNPPROG        = PROGNAME
      DYNPNR          = DYNNUM
      DYNPROFIELD     = 'CONNECTION'
      VALUE_ORG       = 'S'
   TABLES
      VALUE_TAB       = VALUES_TAB.
ENDMODULE.
```

The next screen (statically defined) for screen 100 is itself. It has the following layout:



The input fields have been adopted from the program fields CARRIER and CONNECTION. The pushbutton has the function code CANCEL with function type E.

The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE INIT.

PROCESS AFTER INPUT.
  MODULE CANCEL AT EXIT-COMMAND.

PROCESS ON VALUE-REQUEST.
  FIELD CARRIER MODULE VALUE_CARRIER.
  FIELD CONNECTION MODULE VALUE_CONNECTION.
```

When the user chooses input help for the individual fields, the following is displayed:

- For the *Airline* field, the POV module VALUE_CARRIER is called. The function module F4IF_FIELD_VALUE_REQUEST displays the input help for the component CARRIER1 of the structure DEMOF4HELP from the ABAP Dictionary, namely the search help DEMOF4DE. The user's selection is returned to the screen field CARRIER.
- For the *Flight number* field, the POV module VALUE_CONNECTION is called. The function module DYNP_VALUE_READ transports the value of the screen field CARRIER into the program. The program then reads the corresponding values from the database table SPFLI into the internal table VALUES_TAB using a SELECT statement, and passes the internal table to F4IF_INT_TABLE_VALUE_REQUEST. This displays the internal table as input help, and places the user's selection into the screen field CONNECTION.

Dropdown Boxes

As well as input help, which appears in separate dialog boxes, you can also define input/output fields as dropdown boxes. A dropdown box offers the user a predefined set of input values from which to choose. It is not possible to type a entry into a dropdown box, instead, the user must use one of the values from the list. When the user chooses a value, the PAI event can be triggered simultaneously. If you use a dropdown box for a field, you cannot at the same time use the input help button.

List boxes are currently the only type of dropdown box supported. A list box is a value list containing a single text column of up to 80 characters. Internally, each text field has a key of up to 40 characters. When the user chooses a line, the contents of the text field are placed in the input field on the screen, and the contents of the key are placed in the screen field. The contents and length of the input/output field and the screen field are not necessarily the same.

To make an input/output field into a listbox, you must set the value **L** or LISTBOX in the Dropdown attribute in the Screen Painter. The *visLg* attribute determines the output width of the list box and the field. You can assign a function code to a listbox field. In this case, the PAI event is triggered immediately when the user chooses a value from the list, and the function code is placed in the SY-UCOMM and OK_CODE fields. If you do not assign a function code, the PAI event must be triggered in the usual way, that is, when the user chooses a pushbutton or an element from the GUI status.

If you have assigned a list box to an input/output field, you can use the *Value list* attribute of the screen element to determine how the value list should be compiled. There are two possibilities:

- Value list from input help.

  If you do not enter anything in the value list attribute, the text field uses the first column displayed in the input help [Page 595] assigned to the screen field. The input help can be defined in the ABAP Dictionary, the screen, or a POV dialog module. The key is automatically filled.

- Value list from the ABAP program.

  If you enter **A** in the *value list* attribute, you must fill the value list yourself before the screen is sent (for example, in the PBO event) using the function module VRM_SET_VALUES. When you do this, you must pass an internal table with the type VRM_VALUES to the import parameter VALUES of the function module. VRM_VALUES belongs to the type group VRM. The line type is a structure consisting of the two text fields KEY (length 40) and TEXT (length 80). In the table, you can combine possible user entries from the KEY field with any texts from the TEXT component. You specify the corresponding input/output field in the import parameter ID.

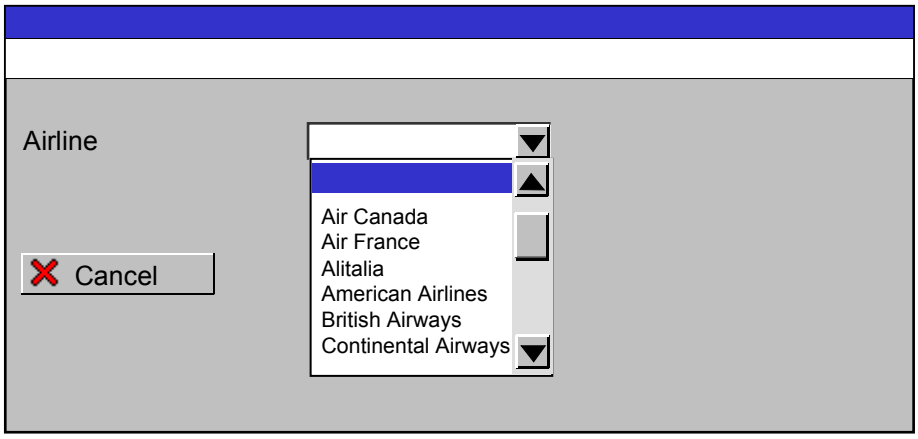
  Dropdown list boxes

  ```
  REPORT DEMO_DYNPRO_DROPDOWN_LISTBOX.

  TYPE-POOLS VRM.

  DATA: NAME  TYPE VRM_ID,
        LIST  TYPE VRM_VALUES,
        VALUE LIKE LINE OF LIST.
  ```

```
DATA: WA_SPFLI TYPE SPFLI,
      OK_CODE LIKE SY-UCOMM,
      SAVE_OK LIKE SY-UCOMM.

TABLES DEMOF4HELP.

NAME = 'DEMOF4HELP-CONNID'.

CALL SCREEN 100.

MODULE CANCEL INPUT.
  LEAVE PROGRAM.
ENDMODULE.

MODULE INIT_LISTBOX OUTPUT.

  CLEAR DEMOF4HELP-CONNID.

  SELECT  CONNID CITYFROM CITYTO DEPTIME
    FROM  SPFLI
    INTO  CORRESPONDING FIELDS OF WA_SPFLI
   WHERE  CARRID = DEMOF4HELP-CARRIER2.

    VALUE-KEY  = WA_SPFLI-CONNID.

    WRITE WA_SPFLI-DEPTIME TO VALUE-TEXT
                      USING EDIT MASK '__:__:__'.

    CONCATENATE VALUE-TEXT
             WA_SPFLI-CITYFROM
             WA_SPFLI-CITYTO
             INTO VALUE-TEXT SEPARATED BY SPACE.

    APPEND VALUE TO LIST.

  ENDSELECT.

  CALL FUNCTION 'VRM_SET_VALUES'
      EXPORTING
          ID              = NAME
          VALUES          = LIST.

ENDMODULE.

MODULE USER_COMMAND_100.
  SAVE_OK = OK_CODE.
  CLEAR OK_CODE.
  IF SAVE_OK = 'CARRIER'
     AND NOT DEMOF4HELP-CARRIER2 IS INITIAL.
    LEAVE TO SCREEN 200.
  ELSE.
    SET SCREEN 100.
  ENDIF.
ENDMODULE.

MODULE USER_COMMAND_200.
  SAVE_OK = OK_CODE.
  CLEAR OK_CODE.
  IF SAVE_OK = 'SELECTED'.
    MESSAGE I888(BCTRAIN) WITH TEXT-001 DEMOF4HELP-CARRIER2
```

```
                                    DEMOF4HELP-CONNID.
  ENDIF.
ENDMODULE.
```

The next screen (statically defined) for screen 100 is 200. It has the following layout:

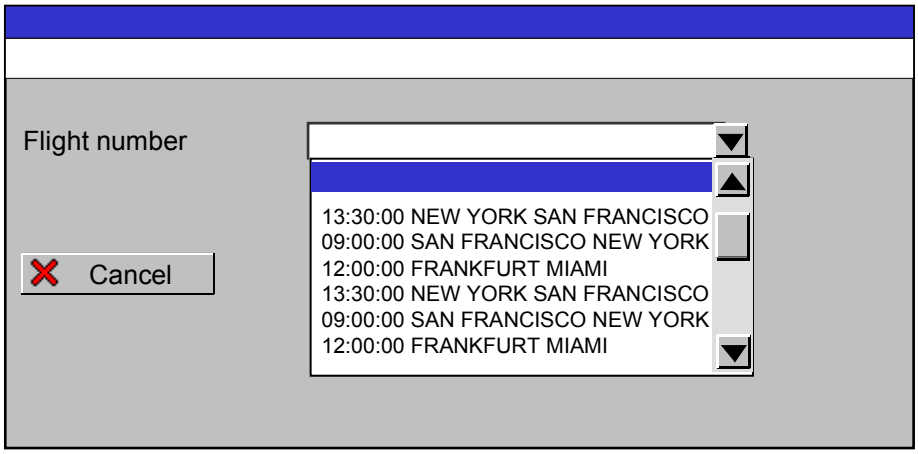


The component CARRIER2 of the ABAP Dictionary structure DEMOF4HELP is assigned to the input field. Its Dropdown attribute is set to **L**, and it has the output length 15. The *Value list* attribute is empty, and it has the function code CARRIER. The pushbutton has the function code CANCEL with function type E.

The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.

PROCESS AFTER INPUT.

  MODULE CANCEL AT EXIT-COMMAND.
  MODULE USER_COMMAND_100.
```

The next screen (statically defined) for screen 200 is 100. It has the following layout:



The component CONNID of the ABAP Dictionary structure DEMOF4HELP is assigned to the input field. Its *Dropdown* attribute is set to **L**, and it has the output

length 30. The *Value list* attribute is set to **A**, and it has the function code SELECTED. The pushbutton has the function code CANCEL with function type E.

The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE INIT_LISTBOX.

PROCESS AFTER INPUT.
  MODULE CANCEL AT EXIT-COMMAND.
  MODULE USER_COMMAND_200.
```

The user cannot type any values into the screen fields. When he or she chooses the input field on screen 100, a value list appears in the list box, compiled from the input help for the field DEMOF4HELP-CARRIER2. This is the search help H_SCARR, which is assigned to the check table SCARR. The value list contains the names of the airlines. When the user chooses an entry, the screen field is filled with the airline code, and the PAI event is triggered. The module USER_COMMAND_100 checks the OK_CODE field and calls screen 200.

In the PBO event of screen 200, an internal table LIST is filled with values from the database table SPFLI. The KEY field is filled with the flight numbers, and other relevant information is placed in the TEXT field. The table LIST is then passed to the function module VRM_SET_VALUES. When the user chooses the input field on screen 200, the TEXT column of the internal table is displayed in the list box. When the user chooses an entry, the screen field is filled with the corresponding entry from the KEY field, and the PAI event is triggered. The module USER_COMMAND_200 checks and processes the OK_CODE field.

Modifying Screens Dynamically

The attributes of screen elements are set statically in the Screen Painter when you define them. However, it is possible to override some of these attributes in ABAP programs with the help of a special internal table.

The R/3 System contains a function called field selection, which allows you to change the attributes of screens dynamically.

It is also possible to set the cursor on a screen to a particular position dynamically from your program.

Setting Attributes Dynamically [Page 612]

The Field Selection Function [Page 620]

Setting the Cursor Position [Page 631]

Setting Attributes Dynamically

Each field on a screen has a set of attributes that are fixed when you define the screen in the Screen Painter. When an ABAP program is running, a subset of the attributes of each screen field can be addressed using the system table SCREEN.

The SCREEN Table

SCREEN is like an internal table with a header line. However, you do not have to declare it in your program. You cannot display it in the Debugger, and cannot use any work area other than its header line to address it. It has the following structure:

Component	Length	Type	Description	Attribute
NAME	30	C	Name of the screen field	*Name*
GROUP1	3	C	Modification group 1	*Group 1*
GROUP2	3	C	Modification group 2	*Group2*
GROUP3	3	C	Modification group 3	*Group3*
GROUP4	3	C	Modification group 4	*Group4*
REQUIRED	1	C	Field input is mandatory	*Mandatory field*
INPUT	1	C	Field is ready for input	*Input*
OUTPUT	1	C	Field is for display only	*Output*
INTENSIFIED	1	C	Field is highlighted	*Highlighted*
INVISIBLE	1	C	Field is suppressed	*Invisible*
LENGTH	1	X	Field length	*VisLg*
ACTIVE	1	C	Field is active	*Input/Output/Invisible*
DISPLAY_3D	1	C	Three-dimensional box	*Two-dimensional*
VALUE_HELP	1	C	Input help button display	*Input help*
REQUEST	1	C	Input exists	-

The final column contains the corresponding attributes of the screen fields in the Screen Painter.

You can modify SCREEN in your ABAP program during the PBO event of a screen. Its contents override the static attributes of the screen fields for a single screen call. The only statements that you can use with SCREEN are:

```
LOOP AT SCREEN.
...
  MODIFY SCREEN.
...
ENDLOOP.
```

You cannot use any further additions in the LOOP AT SCREEN statement.

The component NAME contains the name of the screen field. The components GROUP1 to GROUP4 can contain any three-character code. These codes allow you to include screen fields in up to four modification groups. Modification groups are like an extra key field for the table SCREEN that allow you to change the attributes of all of the elements in a group simultaneously. You assign elements to modification groups statically in the Screen Painter, although you can overwrite them dynamically in a program.

The remaining components are for reading and activating or deactivating the display attributes of screen fields. For all components other than LENGTH, 1 means active and 0 means inactive.

ACTIVE, INPUT, OUTPUT, and INVISIBLE

There are certain hierarchy rules between the components ACTIVE, INPUT, OUTPUT, and INVISIBLE. They also have different effects depending on their respective static settings.

The ACTIVE component has no equivalent in the element attributes. Instead, it changes the components INPUT, OUTPUT, and INVISIBLE.

At the beginning of the PBO, ACTIVE is always set to 1, regardless of the static attribute settings. Setting ACTIVE to 0 automatically sets INPUT = 0, OUTPUT = 0, and INVISIBLE = 1. Any other changes to the settings of INPUT; OUTPUT, and INVISIBLE for the same table row are ignored. Conversely, setting INPUT = 0, OUTPUT = 0, and INVISIBLE = 1 sets ACTIVE to 0, and any further assignment to ACTIVE for the same table row will also be ignored. The setting ACTIVE = 1 has no other effect on the attributes. The only purpose of the ACTIVE component is to allow you to make a screen field inactive through a single assignment. You should particularly note that a module call linked to a FIELD statement in the screen flow logic is always executed, even when SCREEN-ACTIVE = 0 for the field in question. If you want to prevent a module from being processed for an inactive field, you must specify the FIELD and MODULE statements separately.

There are eight possible combinations of ACTIVE, INPUT, OUTPUT, and INVISIBLE, that have the following effect on screen fields:

ACTIVE	INPUT	OUTPUT	INVISIBLE	Effect
1	1	1	0	Screen field is displayed, even if *Invisible* is set statically. Field contents are displayed. Ready for input, even if *Input* is not set statically. However, not ready for input if the *Output only* is set statically.
1	1	0	0	Screen field is displayed, even if *Invisible* is set statically, except when *Output only* is set statically. Field contents are not displayed. Ready for input, even if *Input* is not set statically.
1	0	1	0	Screen field is displayed, even if *Invisible* is set statically. Field contents are displayed. Not ready for input, even if *Input* is set statically.
1	0	0	0	Screen field is displayed, even if *Invisible* is set statically, except when *Output only* is set statically. Field contents are not displayed. Not ready for input, even if *Input* is set statically.

1	1	1	1	Screen field is displayed, even if *Invisible* is set statically, except when *Output only* is set statically. Field contents are not displayed. Ready for input, even if Input is not set statically. User input is masked by asterisks (*).
1	1	0	1	Screen field is displayed, even if *Invisible* is set statically, except when *Output only* is set statically. Output is masked by asterisks (*). Ready for input, even if Input is not set statically. User input is masked by asterisks (*).
1	0	1	1	Screen field inactive. Screen field is not displayed, regardless of the static attributes.
0	0	0	1	Screen field inactive. Screen field is not displayed, regardless of the static attributes.

If a field is statically-defined as *Output only*, setting INPUT = 1 has no effect. INPUT is always 0 for these fields. Masking user input by asterisks can be used for entering user passwords.

If a whole line becomes invisible when you make fields invisible, the screen is automatically made smaller. You can, however, switch off this attribute in the static screen attributes by selecting *Switch off runtime compression*.

REQUIRED

When you set REQUIRED = 1, a field that is **ready for input** is made mandatory. Users can only leave the screen when all mandatory fields contain an entry. Exception: Function codes with type E and modules with the AT EXIT-COMMAND addition.

DISPLAY_3D

When you set DISPLAY_3D = 0, the three-dimensional frame for input/output fields is removed. You cannot use DISPLAY_3D = 1 to create a three-dimensional effect for text fields or screen fields with the *Output only* attribute.

VALUE_HELP

Setting VALUE_HELP to 0 or 1 switches the input help button off and on respectively.

INTENSIFIED

If you set INTENSIFIED = 1, the field contents of input fields are changed from black to red. The contents of output fields are changed from black to blue.

LENGTH

You can set the LENGTH component to a value **shorter** than the statically-defined output length (vislength) for input/output fields and *Output only* fields. This allows you to shorten their output length. You cannot shorten other screen elements, or lengthen any screen elements.

REQUEST

Setting REQUEST = 1 for a field that is **ready for input** has the same effect in the PAI event as if the user had changed the field contents. This means that a conditional module call [Page 572] using ON REQUEST or ON CHAIN-REQUEST would be executed regardless of whether the user really changed the field. REQUEST is automatically reset to 0.

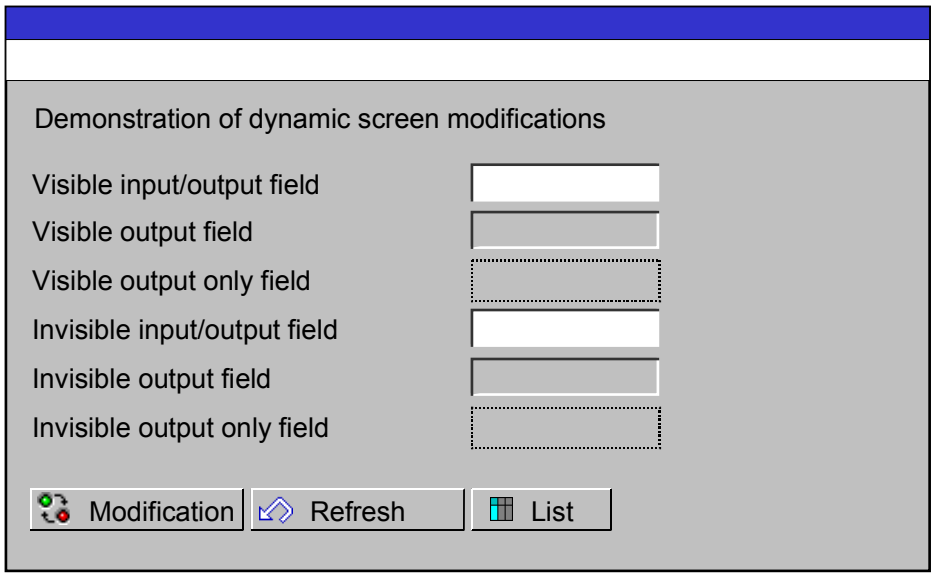
Dynamic screen modifications.

```
REPORT DEMO_DYNPRO_MODIFY_SCREEN.

INCLUDE DEMO_DYNPRO_MODIFY_SCREEN_SEL.

DATA: FIELD1(10), FIELD2(10), FIELD3(10),
      FIELD4(10), FIELD5(10), FIELD6(10).

DATA: OK_CODE LIKE SY-UCOMM,
      SAVE_OK LIKE SY-UCOMM.

DATA: ITAB LIKE TABLE OF SCREEN WITH HEADER LINE.

DATA LENGTH(2) TYPE C.

FIELD1 = FIELD2 = FIELD3 = '0123456789'.

CALL SCREEN 100.

MODULE STATUS_0100 OUTPUT.
  CLEAR: ITAB, ITAB[].
  SET PF-STATUS 'SCREEN_100'.
  IF SAVE_OK = 'MODIFY'.
    ITAB-NAME = TEXT-001.
    APPEND ITAB.
    LOOP AT SCREEN.
      IF SCREEN-GROUP1 = 'MOD'.
        MOVE-CORRESPONDING SCREEN TO ITAB.
        APPEND ITAB.
    ENDIF.
  ENDLOOP.
    PERFORM CHANGE_INPUT USING:
            ACT, INP, OUT, INV, REQ, INT, D3D, HLP, RQS.
    CALL SELECTION-SCREEN 1100 STARTING AT 45 5.
    PERFORM CHANGE_INPUT USING:
            ACT, INP, OUT, INV, REQ, INT, D3D, HLP, RQS.
    MESSAGE S159(AT) WITH ACT INP OUT INV.
    CLEAR ITAB.
    APPEND ITAB.
    LOOP AT SCREEN.
      IF SCREEN-GROUP1      = 'MOD'.
         SCREEN-ACTIVE      = ACT.
         SCREEN-INPUT       = INP.
```

```
            SCREEN-OUTPUT      = OUT.
            SCREEN-INVISIBLE   = INV.
            SCREEN-REQUIRED    = REQ.
            SCREEN-INTENSIFIED = INT.
            SCREEN-DISPLAY_3D  = D3D.
            SCREEN-VALUE_HELP  = HLP.
            SCREEN-REQUEST     = RQS.
            SCREEN-LENGTH      = LEN.
            MODIFY SCREEN.
          ENDIF.
        ENDLOOP.
        CLEAR ITAB.
        ITAB-NAME        = TEXT-002.
        ITAB-ACTIVE      = ACT.
        ITAB-INPUT       = INP.
        ITAB-OUTPUT      = OUT.
        ITAB-INVISIBLE   = INV.
        ITAB-REQUIRED    = REQ.
        ITAB-INTENSIFIED = INT.
        ITAB-DISPLAY_3D  = D3D.
        ITAB-VALUE_HELP  = HLP.
        ITAB-REQUEST     = RQS.
        ITAB-LENGTH      = LEN.
        APPEND ITAB.
        CLEAR  ITAB.
        APPEND ITAB.
      ENDIF.
    ENDMODULE.

    MODULE CANCEL INPUT.
      LEAVE PROGRAM.
    ENDMODULE.

    MODULE USER_COMMAND_0100 INPUT.
      SAVE_OK = OK_CODE.
      CLEAR OK_CODE.
      CASE SAVE_OK.
        WHEN 'MODIFY'.
          LEAVE TO SCREEN 100.
        WHEN 'LIST'.
          CLEAR ITAB.
          ITAB-NAME = TEXT-003.
          APPEND ITAB.
          LOOP AT SCREEN.
            IF SCREEN-GROUP1 = 'MOD'.
              MOVE-CORRESPONDING SCREEN TO ITAB.
              APPEND ITAB.
            ENDIF.
          ENDLOOP.
          CALL SCREEN 200 STARTING AT 45 5
                          ENDING   AT 95 22.
      ENDCASE.
    ENDMODULE.
```

```
MODULE REQUESTED INPUT.
  MESSAGE S888(SABAPDOCU) WITH TEXT-004.
ENDMODULE.

MODULE STATUS_0200 OUTPUT.
  SET PF-STATUS 'SCREEN_200'.
  SUPPRESS DIALOG.
  LEAVE TO LIST-PROCESSING AND RETURN TO SCREEN 0.
  FORMAT COLOR COL_HEADING ON.
  WRITE: 10 'ACT', 14 'INP', 18 'OUT', 22 'INV', 26 'REQ',
         30 'INT', 34 'D3D', 38 'HLP', 42 'RQS', 46 'LEN'.
  FORMAT COLOR COL_HEADING OFF.
  ULINE.
  LOOP AT ITAB.
    IF ITAB-NAME = ' '.
       ULINE.
    ELSEIF ITAB-NAME = TEXT-001 OR ITAB-NAME = TEXT-003.
      FORMAT COLOR COL_NORMAL ON.
    ELSE.
      FORMAT COLOR COL_NORMAL OFF.
    ENDIF.
    LEN = ITAB-LENGTH.
    LENGTH = ' '.
    IF LEN NE 0.
      LENGTH = LEN.
    ENDIF.
    WRITE: /(8) ITAB-NAME,
              11 ITAB-ACTIVE,
              15 ITAB-INPUT,
              19 ITAB-OUTPUT,
              23 ITAB-INVISIBLE,
              27 ITAB-REQUIRED,
              31 ITAB-INTENSIFIED,
              35 ITAB-DISPLAY_3D,
              39 ITAB-VALUE_HELP,
              43 ITAB-REQUEST,
              47 LENGTH.
  ENDLOOP.
ENDMODULE.

FORM CHANGE_INPUT CHANGING VAL.
  IF VAL = 'X'.
     VAL = '1'.
  ELSEIF VAL = ' '.
     VAL = '0'.
  ELSEIF VAL = '1'.
     VAL = 'X'.
  ELSEIF VAL = '0'.
     VAL = ' '.
  ENDIF.
ENDFORM.
```

The next screen (statically defined) for screen 100 is itself, and it has the following layout:



The input/output fields are assigned to the fields FIELD1 to FIELD6 in the ABAP program. These fields, along with the text field TEXT in the top line, are assigned to the modification group MOD. The remaining screen elements are not assigned to a modification group. The function codes of the pushbuttons are MODIFY, UNDO, and LIST.

The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0100.

PROCESS AFTER INPUT.
  FIELD FIELD1 MODULE REQUESTED ON REQUEST.
  MODULE USER_COMMAND_0100.
  MODULE CANCEL AT EXIT-COMMAND.
```

If you choose *Modification*, a selection screen appears, on which you can select which of the components of the SCREEN table should be set to active or inactive. In the subroutine CHANGE_INPUT, the user input from the checkboxes is converted into the digits 0 and 1. Selection screen 1100 is defined in the following include program:

```
*---------------------------------------------------------------
---*
*   INCLUDE
DEMO_DYNPRO_MODIFY_SCREEN_SEL                          *
*---------------------------------------------------------------
---*

SELECTION-SCREEN BEGIN OF SCREEN 1100.

SELECTION-SCREEN BEGIN OF BLOCK B1 WITH FRAME NO INTERVALS.
PARAMETERS: ACT AS CHECKBOX DEFAULT '1',
            INP AS CHECKBOX DEFAULT '1',
            OUT AS CHECKBOX DEFAULT '1',
```

```
            INV AS CHECKBOX DEFAULT '0'.
SELECTION-SCREEN END OF BLOCK B1.

SELECTION-SCREEN BEGIN OF BLOCK B2 WITH FRAME NO INTERVALS.
PARAMETERS: REQ AS CHECKBOX DEFAULT '0',
            D3D AS CHECKBOX DEFAULT '1',
            HLP AS CHECKBOX DEFAULT '0',
            INT AS CHECKBOX DEFAULT '0'.
SELECTION-SCREEN END OF BLOCK B2.

SELECTION-SCREEN BEGIN OF BLOCK B3 WITH FRAME NO INTERVALS.
PARAMETERS RQS AS CHECKBOX DEFAULT '0'.
SELECTION-SCREEN END OF BLOCK B3.

SELECTION-SCREEN BEGIN OF BLOCK B4 WITH FRAME NO INTERVALS.
PARAMETERS  LEN TYPE I DEFAULT 10.
SELECTION-SCREEN END OF BLOCK B4.

SELECTION-SCREEN END OF SCREEN 1100.
```

When you modify the screen, the current contents of the table SCREEN for the modification group MOD are placed in the auxiliary table ITAB in the PBO event. This represents the static settings of the corresponding screen fields. SCREEN is then modified according to the settings chosen by the user. The user input is also stored in ITAB.

The screen is displayed, and the fields that belong to the modification group MOD are displayed according to the new attributes stored in SCREEN. The contents of the entries for ACTIVE, INPUT, OUTPUT, and INVISIBLE are visible in the status bar.

In the PAI event, the module REQUESTED is called if the user changes the contents of FIELD1 or sets the REQUEST component of the table SCREEN.

If the user chooses *List*, the current contents of the table SCREEN are appended to ITAB in the PAI module USER_COMMAND_0100. Screen 200 is then called as a modal dialog box. Screen 200 is used to display the list of table ITAB. The contents of the table SCREEN are displayed before and after modification. This makes it possible to compare directly the effects of the input on the individual components and the dependencies between the entries.

The Field Selection Function

This topic describes how a special function *Field selection* (transaction SFAW and some function modules) support you in changing screen field attributes dynamically.

Field Selection - Overview

The function *Field selection* allows you to change the attributes of screen fields dynamically at runtime. However, you should only use this option if you often need to assign different field attributes to the same screen for technical reasons. In this case, the same rules apply for all fields, so any field modification is clear.

The following basic rules apply:

- All fields involved in the field selection process are grouped together in field selection tables and maintained using the *Field selection* function.
- You can maintain fields for an ABAP program or screen group (see below).
- On screens belonging to the screen group "blank" ('_'), there is no dynamic field selection.
- Central field selection uses the modification group SCREEN-GROUP1, which therefore cannot be used for anything else.
- If you use fixed special rules in the field selection which are the same as changing the program, you should make the changes directly in the program, not in the maintenance transaction.

With field selection, you can activate or deactivate the following attributes at runtime:

- Input
- Output
- Required
- Active
- Highlighted
- Invisible

You can also determine the conditions and the type of changes involved. During the event PROCESS BEFORE OUTPUT, you call a function module which checks the conditions and, if necessary, changes the attributes.

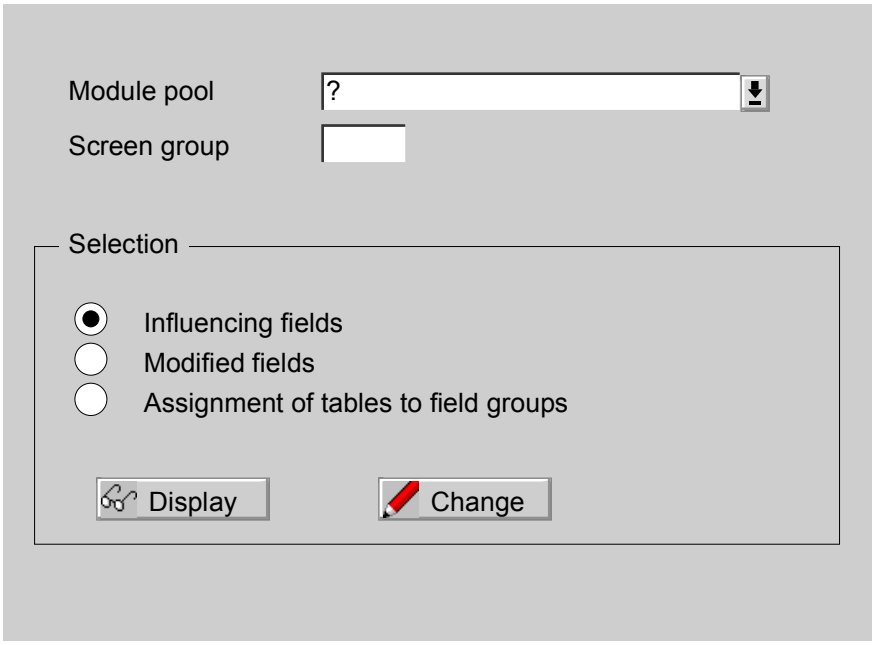
Field selection distinguishes between influencing fields and modified fields. Modified fields must, of course, be screen fields. All fields should relate to the ABAP Dictionary. You can do this by creating corresponding interface work areas in the ABAP program using TABLES. At runtime, a function module analyzes the field contents of the influencing fields and then sets the attributes of the modified fields accordingly.

Combining Screens into Screen Groups

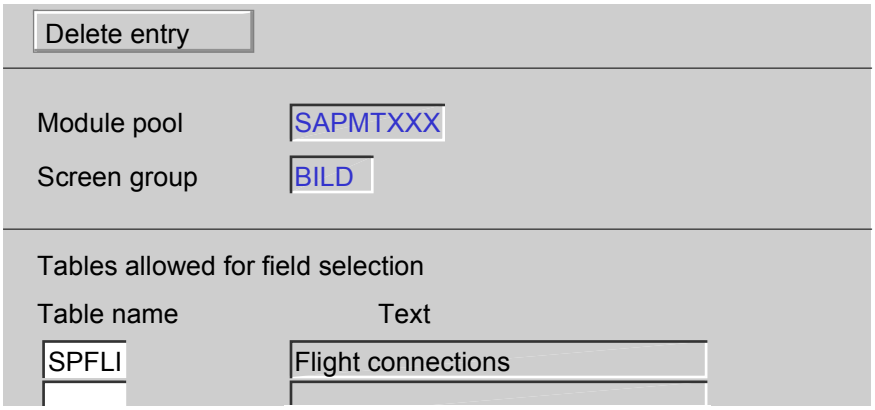
Rather than maintaining field selection separately for each screen of a program, you can combine logically associated screens together in a screen group. This reduces the amount of maintenance required. To assign a screen to a screen group, enter the group in the field ***Screen group*** on the attributes screen in the Screen Painter.

Calling Field Selection

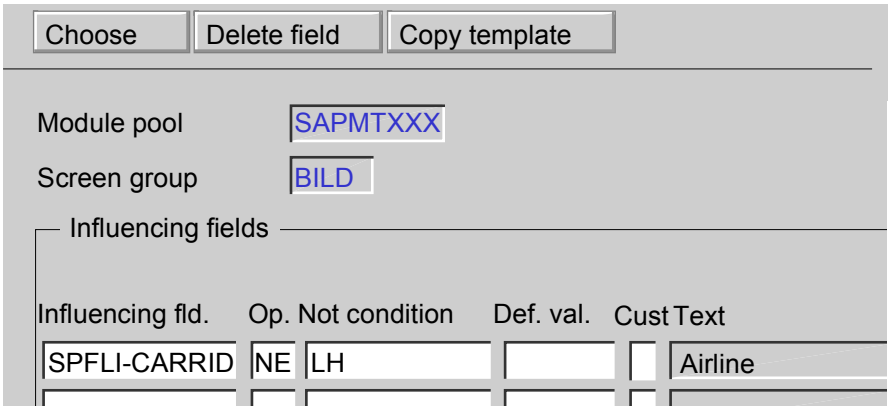
To call field selection, choose *Tools* → *ABAP Workbench* → *Development* → *Other tools* → *Field selection*. Maintenance is by program and screen group.



First, you must declare the ABAP Dictionary names of the fields involved. Choose *Assign tables to screen group* and enter the tables, for example:



Save your entries and choose *Influencing fields* to enter the desired influencing fields into the list and optionally specify a NOT condition, a default value, and a field *Cust*, for example:



The NOT condition is to be understood as a preselection. If one of the fields satisfies the NOT condition, it is not relevant for the following screen modification. Using the NOT condition may improve performance.

Influencing field: SPFLI-CARRID

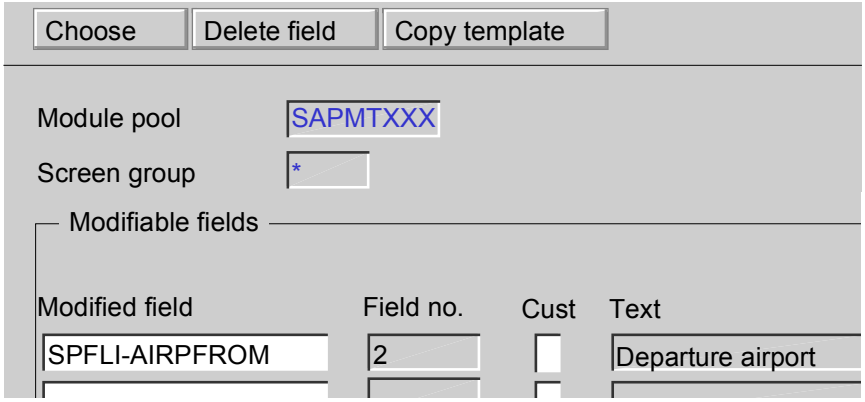
NOT condition: NE LH

SPFLI-CARRID is relevant for the field selection only if its contents are not equal to LH at runtime.

At runtime, the system uses the default value for the field modification if it cannot find the current value of the influencing field in the list of maintained values. To achieve this, you need to define an influence for the default value. This option allows you to maintain all the forms of an influencing field, which have the same influence, with a single entry.

By setting the field *Cust* accordingly, you can decide whether to allow customers to use the corresponding field for field selection or not. This field selection option is based on the predefined SAP field selection and allows customers to set screen attributes to suit their own requirements within a framework determined by application development. Many application areas have their own special Customizing transactions for this purpose (these are parameter transactions related to the Transaction SFAC; refer here to the documentation on the function module FIELD_SELECTION_CUSTOMIZE)

Then, choose *Modified fields* to enter all modifiable fields into a list, for example:

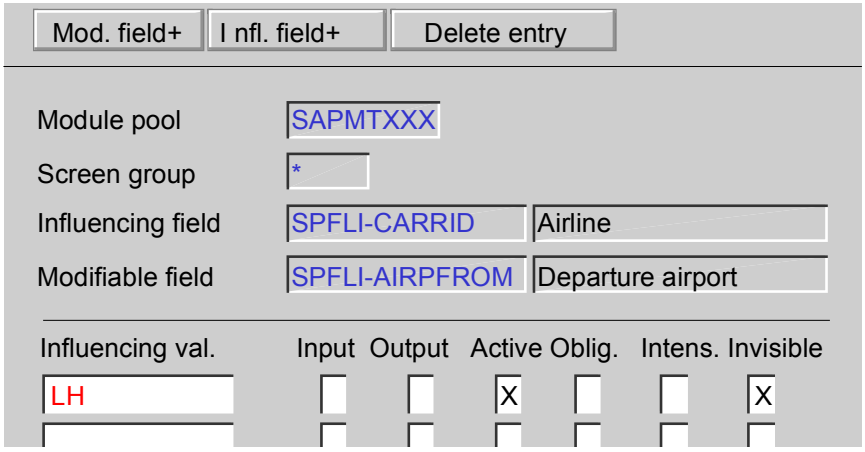


You can again set the field *Cust* accordingly if you want to allow customers to use these modifiable fields in special Customizing transactions. If *Cust* is selected, customers can also modify the field.

Each of these influencing and modifiable fields has an internal number which is unique for each program. When it is generated, the number is automatically placed in SCREEN-GROUP1 of the appropriate screens and cannot be changed in Screen Painter. This enables the system to establish a one-to-one relationship between the field name and SCREEN-GROUP1.

Finally, you create links between influencing and modifiable fields from the two lists: specify which contents of an influencing field influences the modifiable field in which way.

To link fields, select the fields from both lists with *Choose* or double-click. If you select an influencing field, the list of modifiable fields appears and vice versa. From this list, select the desired link. A list appears in which you can enter the relevant conditions, for example:



The entry above results in suppressing the display of field SPFLI-AIRPFROM on those screens, in whose PBO the corresponding field selection function modules are called and if SPFLI-CARRID then contains 'LH'.

Combination Rules For Attributes

If several influencing fields influence the same modified field, there must be a combination rule to determine how these influences are linked. You use the tables to below establish how a single field attribute is set, if it is activated and/or deactivated by different influences. The screen processor controls the combination of several attributes.

Input			
		Field 1	
		‘ ’	'X'
Field 2	‘ ’	‘ ’	‘ ’
	'X'	‘ ’	'X'

Output			
		Field 1	
		‘ ’	'X'
Field 2	‘ ’	‘ ’	‘ ’
	'X'	‘ ’	'X'

Active			
		Field 1	
		‘ ’	'X'
Field 2	‘ ’	‘ ’	‘ ’
	'X'	‘ ’	'X'

Mandatory			
		Field 1	
		‘ ’	'X'
Field 2	‘ ’	‘ ’	'X'
	'X'	'X'	'X'

Highlighted			
		Field 1	
		‘ ’	'X'
Field 2	‘ ’	‘ ’	'X'
	'X'	'X'	'X'

Invisible			
		Field 1	
		‘ ’	'X'
Field 2	‘ ’	‘ ’	'X'
	'X'	'X'	'X'

Description of characters:
_ = switched off , X = switched on

If Field 1 makes a screen field invisible (X), Field 2 cannot change this.

Static Attributes in the Screen Painter

In screen modifications, the system takes into account not only the entries you make during field selection, but also any entries made in Screen Painter. This means that the result of the above combination is linked to the screen field attributes according to the same linking rules as individual attributes.

To take advantage of the full dynamic modification range, you should use the following attributes in the Screen Painter:

Input = 'X'
Output = 'X'
Mandatory = '_'
Invisible = '_'
Highlighted = '_'.

Conversely, you cannot change the values defined on the screen in the following manner:

Input = '_'
Output = '_'
Mandatory = 'X'
Invisible= 'X'
Highlighted = 'X'

Furthermore, the following applies: If you enter the following combination of influences, it is not really a valid combination, since the combination rules stipulate that the specified display attributes cannot be changed by another influencing field (or the screen).

Input = 'X'
Output = 'X'
Active = 'X'
Mandatory = '_'
Highlighted = '_'
Invisible = '_'

When you reenter the field selection, influences that have no effect, such as the one above, are not displayed. However, if you have defined a default value for the influencing field, it can make sense to display and maintain influences of this kind.

Generating the Field Selection

If the list of modified fields has changed at all, you must generate the field selection. This produces consecutive numbers for the modified SCREEN-GROUP1 fields in the screens of the relevant ABAP program.

To generate the field selection, choose *Generate* from Transaction SFAW.

Function Modules for Field Selection

To activate field selection for a screen in the PROCESS BEFORE OUTPUT event, you can call one of the function modules FIELD_SELECTION_MODIFY_ALL or FIELD_SELECTION_MODIFY_SINGLE. Both these function modules determine the contents of the influencing fields, refer if necessary to the combination rules and execute the screen modification. FIELD_SELECTION_MODIFY_ALL executes the LOOP AT SCREEN statement itself. However, with FIELD_SELECTION_MODIFY_SINGLE, you must the LOOP AT SCREEN

yourself and call the function module within the loop. You can thus perform your own additional screen modifications within the LOOP.

Examples of calling the function modules in the event PBO:

1.

```
CALL FUNCTION 'FIELD_SELECTION_MODIFY_ALL'
    EXPORTING MODULPOOL    = MODULPOOL
              DYNPROGRUPPE = DYNGRP.
```

2.

```
LOOP AT SCREEN.
  IF SCREEN_GROUP1 NE SPACE AND
    SCREEN-GROUP1 NE '000'.
    CALL FUNCTION 'FIELD_SELECTION_MODIFY_SINGLE'
        EXPORTING MODULPOOL    = MODULPOOL
                  DYNPROGRUPPE = DYNGRP.
* Special rules
    MODIFY SCREEN.
  ENDIF.
ENDLOOP.
```

3.

as 1, but includes your own LOOP AT SCREEN for special rules.

You must decide in each individual case which of the options 2 or 3 produces the best performance.

Since the *Module pool* and *Screen group* parameters determine the field selection, you must maintain influences for these.

The *Module pool* parameter defines, in main memory, which loaded module pool you use to search for the current values of the influencing fields.

When you call the function modules, you may not use the system fields SY-REPID or SY-DYNGR directly as actual parameters. Instead, you must assign them to other fields at an appropriate point. For example:

```
MODULPOOL = SY-REPID.
DYNGRP    = SY-DYNGR
```

Sometimes, the *Module pool* values may differ from the current SY_REPID value.

If the *Screen group* parameter is blank, the system uses the current contents of SY-DYNGR. This is not possible for the *Module pool* parameter because the value '_' (blank) prevents any field modification.

Let us consider an ABAP program in which the second screen contains the following module call in its PBO event:

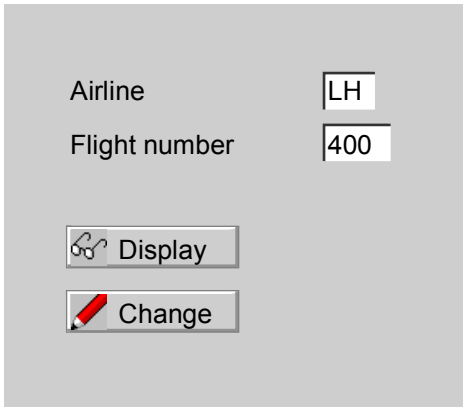
```
PROCESS BEFORE OUTPUT.

...
  MODULE MODIFY_SCREEN.
```

Suppose the module MODIFY_SCREEN contains the following function call:

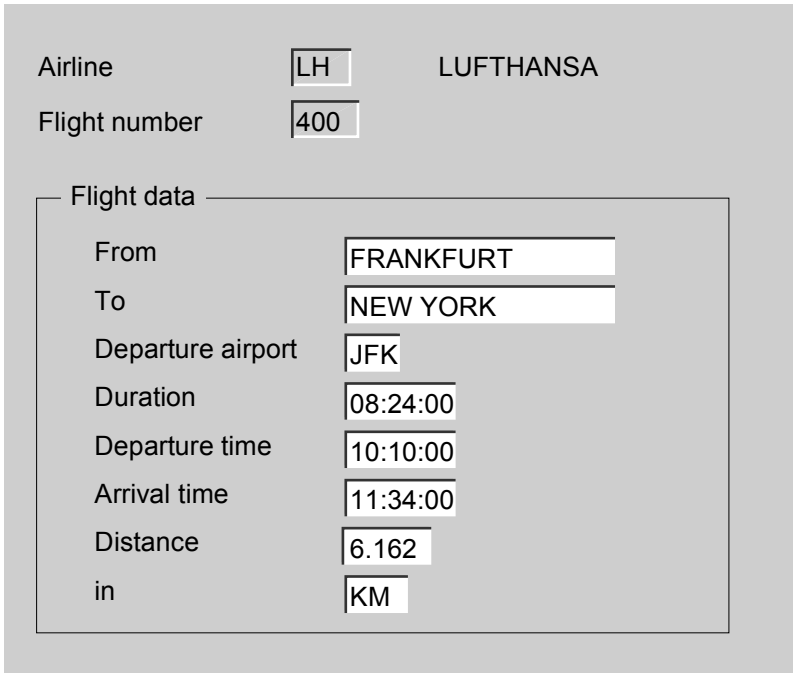
```
MODULE MODIFY_SCREEN OUTPUT.

  CALL FUNCTION 'FIELD_SELECTION_MODIFY_ALL'
     EXPORTING
        DYNPROGRUPPE =  'BILD'
        MODULPOOL    =  'SAPMTXXX'
     EXCEPTIONS
        OTHERS       = 1.
```

Let us also suppose that the influences for the screen group BILD and the corresponding ABAP program are maintained as described above.

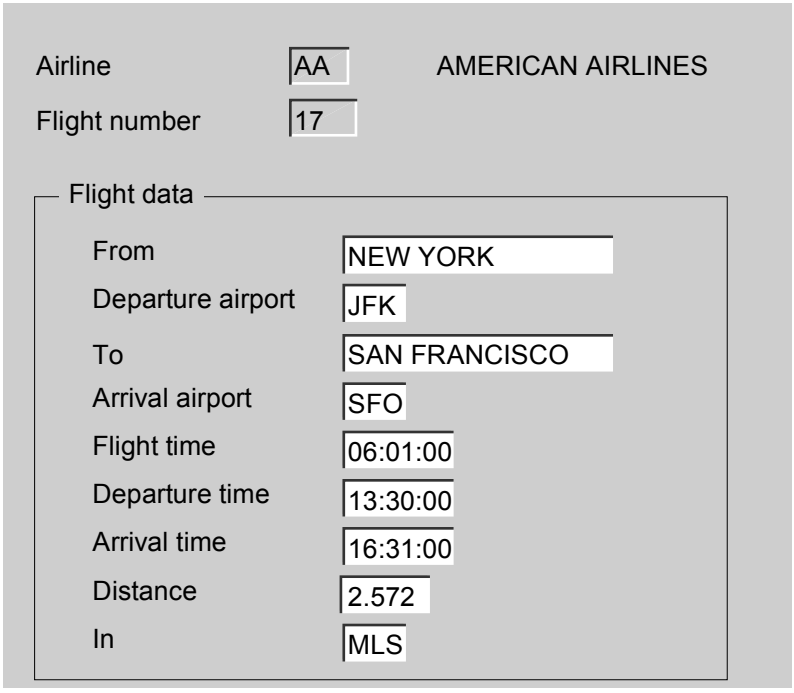
After calling the transaction, suppose these entries are made:



After choosing *Change*, the following screen appears:



However, if instead of 'LH' as airline carrier 'AA' is entered, the following screen appears:



When entering 'LH', the field SPFLI-AIRPFROM is invisible. When entering 'AA', it is visible as *Dep. airport*.

Linking Fields

Every influencing field influences a field which can be modified regardless of other fields. Linking influencing fields can be desirable in some cases but then only possible by defining help fields, which you must set in the application program before calling the function module.

This restriction helps the transparency of the field selection.

Examples of Links

Suppose we have the following fields:

- Influencing fields: F4711, F4712
  Field that can be modified: M4711

The following cases can only be implemented using a workaround:

OR Condition and "Ready for Input"

- IfF4711 = 'A' OR F4712 = 'B', then
  M4711 is ready for input.

Solution:

- Define H4711 as an influencing field in SFAW;
  define the following condition in SFAW:
  if H4711 = 'AB'
  then M4711 input on (that is, input = 'X')

In the application program, you must program the following before calling the function module:

```
IF F4711 = 'A' OR F4712 = 'B'.
  H4711 = 'AB'.
ENDIF.
```

AND Condition and "Mandatory"

- IfF4711 = 'A' AND F4712 = 'B',
  then M4711 obligatory and only then.

Solution:

- Maintenance in the field selection:
  If H4711 = 'AB',
  then M4711 is a required-entry field
  (H4711 = 'AB' only precisely with the above AND condition)

In the application program, you would write the following:

```
IF F4711 = 'A' AND F4712 = 'B'
  H4711 = 'AB'
ELSE.
  H4711 =....
ENDIF.
```

The following cases, on the other hand, can be represented directly:

AND Condition and "Ready for Input"

- If F4711 = 'A' AND F4712 = 'B',
  then M4711 is ready for input. F4711 <> ‘A’ OR F4712 <> ‘B’,
  then M4711 is not ready for input.

Solution:

Screen: M4711 ready for input						
Field selection:						
Influencing field	F4711	Value 'A'	Input = 'X'			
					Value 'A1'	Input = ' '
					Value 'AX'	Input = ' '
Influencing field	F4712	Value 'B'	Input = 'X'			
					Value 'B1'	Input = ' '
					Value 'BX'	Input = ' '

OR Condition and "Mandatory"

IfF4711 = 'A' OR F4712 = 'B’, then
M4711 is a required-entry field.

Solution:

Screen: Mandatory is switched off				

Influencing field	F4711	Value 'A'		
				mandatory = 'X'
Influencing field	F4712	Value 'B'		
				mandatory = 'X'

The possibility to define a NOT condition for an influencing field gives further variants of the field selection definition.

Authorization for Field Selection

The authorization object for *Field selection* is "central field selection" (S-FIELDSEL). This object consists of an activity and a program authorization group. The latter is taken from the program authorizations (ABAP Editor).

Possible activities:

'02' = Change
'03' = Display
'14' = Generate field selection information on screens
'15' = Assign relevant tables to field selection

Changing a field selection modifies the program flow. It is also possible in some applications to change the screen attributes of a field from within Customizing. This kind of change does not count as a modification.

Setting the Cursor Position

When a screen is displayed, the system automatically places the cursor in the first field that is ready for input. However, you can also define on which screen element the cursor should appear in your program. The screen element does not have to be an input field. Positioning the cursor can make applications more user-friendly.

You can set the cursor position either statically in the Screen Painter or dynamically in your ABAP program.

Static Cursor Position

To define the cursor position statically, enter the name of the required screen element in the Cursor position screen attribute [Page 527] in the Screen Painter.

Dynamic Cursor Position

To set the cursor position dynamically, use the following statement in an ABAP dialog module in the PBO event:

SET CURSOR FIELD <f> [OFFSET <off>].

<f> can be a literal or a variable containing the name of a screen element. You can use the OFFSET addition to place the cursor at a particular point within an input/output field.

Setting the cursor position

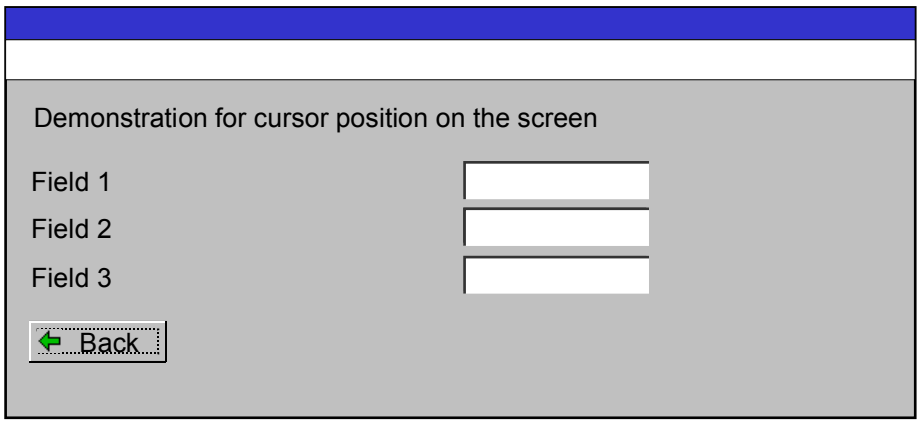
```
REPORT DEMO_DYNPRO_SET_CURSOR.

DATA: FIELD1(14), FIELD2(14), FIELD3(14),
      NAME(10).

SELECTION-SCREEN BEGIN OF BLOCK BLOC WITH FRAME.
  PARAMETERS: DEF RADIOBUTTON GROUP RAD,
              TXT RADIOBUTTON GROUP RAD,
              F1  RADIOBUTTON GROUP RAD,
              F2  RADIOBUTTON GROUP RAD,
              F3  RADIOBUTTON GROUP RAD.
SELECTION-SCREEN END OF BLOCK BLOC.

PARAMETERS POS TYPE I.

IF TXT = 'X'.
  NAME = 'TEXT'.
ELSEIF F1 = 'X'.
  NAME = 'FIELD1'.
ELSEIF F2 = 'X'.
  NAME = 'FIELD2'.
ELSEIF F3 = 'X'.
  NAME = 'FIELD3'.
ENDIF.

CALL SCREEN 100.
```

Setting the Cursor Position

```
MODULE CURSOR OUTPUT.
  IF DEF NE 'X'.
    SET CURSOR FIELD NAME OFFSET POS.
  ENDIF.
  SET PF-STATUS 'SCREEN_100'.
ENDMODULE.

MODULE BACK INPUT.
  LEAVE SCREEN.
ENDMODULE.
```

At the start of the program, a selection screen appears on which you can select a cursor position.

Screen 100 is then called. The next screen (statically defined) for screen 100 is itself, and it has the following layout:



The input/output fields are assigned to the fields FIELD1 to FIELD3 in the ABAP program. The heading is the text field TEXT, and the pushbutton is the screen element PUSH.

The static cursor position in the screen attributes is set to PUSH.

The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE CURSOR.

PROCESS AFTER INPUT.
  MODULE BACK AT EXIT-COMMAND.
```

During the PBO event, before screen 100 is displayed, the cursor is set according to the user's choice on the selection screen. If the user chooses the static default, the cursor is placed on the pushbutton, otherwise on the header or one of the input fields. The position POS is only taken into account for the input fields.

Switching on Hold Data Dynamically

In the attributes of a screen [Page 527], you can enable the following standard menu entries by setting the *Hold data* attribute:

- *System* → *User profile* → *Hold data*

  Hold data allows users to retain values that they have entered on a screen so that they appear the next time they start the same transaction. Only values actually entered by the user are retained. They are placed in the corresponding field as default values each time the screen is processed, and they overwrite the values transferred from the ABAP program in the PBO event.

- *System* → *User profile* → *Set data*

  This has the same effect as *Hold data*. Additionally, when the held data is placed in the screen fields, these fields are no longer ready for input.

- *System* → *User profile* → *Delete data*

  This deletes the held data, and makes the relevant fields on the screen ready for input again.

If *Hold data* is not activated in the screen attributes, the above menu entries have no effect.

In the PBO event of a screen, you can overwrite the *Hold data* attribute dynamically using the statement

SET HOLD DATA ON|OFF.

The ON addition activates the attribute, OFF deactivates it.

Example

Hold data

```
REPORT DEMO_DYNPRO_SET_HOLD_DATA.

DATA FIELD(10).

CALL SCREEN 100.

FIELD = 'XXXXXXXXXX'.

CALL SCREEN 100.

MODULE HOLD_DATA OUTPUT.
  SET HOLD DATA ON.
ENDMODULE.
```

The statically-defined next screen for screen 100 is 0, and it contains a single input/output field called FIELD.

The flow logic is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE HOLD_DATA.

PROCESS AFTER INPUT.
```

Switching on Hold Data Dynamically

In the PBO event, the *Hold data* attribute is activated dynamically. If the user enters a value and then chooses *System* → *User profile* → *Hold data* or *Set data*, the same value is displayed in the field when the screen is next displayed. This value is displayed each time the screen is called until the user chooses *Delete data*. This overwrites any value assigned to the field FIELD in the ABAP program.

Complex Screen Elements

Status Icons

Status icons are display elements that you can use to represent the state of a program graphically. It is possible to use any SAPgui icon. However, in practice, you should only use the icons for status display [Ext.] listed in the SAP Style Guide.

You can only create status icons in the graphical Screen Painter. When you create one, you assign a name and a screen field, called a status field, to it. The visible length of the screen field determines the amount of space that the icon can take up on the screen. As well as the icon itself, you can also place text in the field. The actual length (defLg) of the status field must be long enough for the internal representation of the icon, plus any text and quickinfo text that you specify. When you create the status icon in the Screen Painter, a placeholder appears on the screen. You must specify the icon itself and its text and quickinfo text in the PBO event of your ABAP program.

In order to define the icon in your ABAP program, you must create a field with the same name as the status field on the screen and the ABAP Dictionary type ICONS-TEXT. You can then fill this field with the required technical information in the PBO event. When the screen is displayed, the information is transferred to the status field and the icon appears.

To fill the field in your ABAP program, use the function module ICON_CREATE. It has the following import parameters:

- **NAME**

  The name of the required icon. These are listed in the SAP Style Guide [Ext.], in the include program <ICON>, or the corresponding input help in the Screen Painter or Menu Painter.

- **TEXT**

  This parameter allows you to enter a text that will appear after the icon on the screen.

- **INFO**

  In this parameter, you can specify a quickinfo text, which appears whenever the mouse pointer is positioned over the icon.

- **ADD_STDINF**

  This flag switches the quickinfo display on or off.

The function module converts these parameters into a single string, which is returned in the export parameter RESULT. When you assign the RESULT parameter to the status field, it contains all the information required to display the status icon.

Status icons

```
REPORT DEMO_DYNPRO_STATUS_ICONS.

DATA VALUE TYPE I VALUE 1.

DATA: STATUS_ICON TYPE ICONS-TEXT,
      ICON_NAME(20),
      ICON_TEXT(10).

CALL SCREEN 100.
```

```
MODULE SET_ICON OUTPUT.

  SET PF-STATUS 'SCREEN_100'.

CASE VALUE.
    WHEN 1.
      ICON_NAME = 'ICON_GREEN_LIGHT'.
      ICON_TEXT =  TEXT-003.
    WHEN 2.
      ICON_NAME = 'ICON_YELLOW_LIGHT'.
      ICON_TEXT =  TEXT-002.
    WHEN 3.
      ICON_NAME = 'ICON_RED_LIGHT'.
      ICON_TEXT =  TEXT-001.
  ENDCASE.

  CALL FUNCTION 'ICON_CREATE'
       EXPORTING
            NAME                  = ICON_NAME
            TEXT                  = ICON_TEXT
            INFO                  = 'Status'
            ADD_STDINF            = 'X'
      IMPORTING
            RESULT                =  STATUS_ICON
      EXCEPTIONS
            ICON_NOT_FOUND        = 1
            OUTPUTFIELD_TOO_SHORT = 2
            OTHERS                = 3.

  CASE SY-SUBRC.
    WHEN 1.
      MESSAGE E888(BCTRAIN) WITH TEXT-004.
    WHEN 2.
      MESSAGE E888(BCTRAIN) WITH TEXT-005.
    WHEN 3.
      MESSAGE E888(BCTRAIN) WITH TEXT-006.
  ENDCASE.

ENDMODULE.

MODULE CANCEL INPUT.
  LEAVE PROGRAM.
ENDMODULE.

MODULE CHANGE.
CASE VALUE.
    WHEN 1.
      VALUE = 2.
    WHEN 2.
      VALUE = 3.
    WHEN 3.
      VALUE = 1.
  ENDCASE.
ENDMODULE.
```

The next screen (statically defined) for screen 100 is itself, and it has the following layout:

The screen contains a status field called STATUS_ICON with a visible length of 12 and defined length 26. The status icon and the space for the text are represented by placeholders in the Screen Painter.

The screen flow logic is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE SET_ICON.

PROCESS AFTER INPUT.
  MODULE CANCEL AT EXIT-COMMAND.
  MODULE CHANGE.
```

The dialog module SET_ICON passes various values to the function module ICON_CREATE, depending on the values of the fields in the program. The status field STATUS_ICON is filled with the contents of the export parameter RESULT of the function module. The corresponding icon with its text and quickinfo is then displayed on the screen. When the user chooses *Continue*, the contents of the field VALUE are changed in the PAI, and consequently a new icon is defined in the PBO event. The following icons and texts are displayed:

 **Low**

 **Middle**

 **High**

The quickinfo text is ‘Status’ for all of the icons.

Context Menus

The user interface of a screen is defined by a GUI status [Page 548], which you define in the Menu Painter and assign the type *Dialog status*. For each dialog status, the system automatically creates a **standard context menu**, which the user can display by clicking the right-hand mouse button on the screen (or choosing `Shift+F10`). The standard context menu contains all of the function keys to which functions are assigned. It therefore makes it easy to access **any** function code that is available using the keyboard, since normally only the most important are assigned to the application toolbar.

However, as well as the standard context menu, you can define context-specific menus for any of the following screen elements:

- Input/output fields
- Text fields
- Table controls
- Group boxes
- Subscreens

When you select one of these elements using the right-hand mouse button, you can create a dynamic context menu in the ABAP program. This may contain any functions, and is not restricted to function keys. You cannot assign context menus to pushbuttons, checkboxes, or radio buttons. However, you can assign unique function codes to them instead.

Global Class CL_CTMENU

Context menus are objects of the global ABAP Objects [Page 1291] class CL_CTMENU. In the class library, the class belongs to the Frontend Services component, in which the classes of the Control Framework are also stored (see Custom Controls [Page 661]). It contains methods that allow you to define context menus dynamically in a program. As a template, you can create your own context menus statically in the Menu Painter.

The most important methods of the class CL_CTMENU are:

Method	Function
LOAD_GUI_STATUS	Assigns a context menu that has already been defined statically in the Menu Painter to a local context menu in a program
ADD_FUNCTION	Assigns a single function to a context menu in a program
ADD_MENU	Assigns another local context menu to the current local context menu in the program
ADD_SUBMENU	Assigns another local context menu to the current local context menu in the program as a cascading menu
ADD_SEPARATOR	Adds a separator
HIDE_FUNCTIONS	Hides functions
SHOW_FUNCTIONS	Shows functions

DISABLE_FUNCTIONS	Deactivates functions
ENABLE_FUNCTIONS	Activates functions
SET_DEFAULT_FUNCTION	Sets a default function, which is highlighted when the menu is displayed

In the above table, “local context menu in a program” means an object of class CL_CTMENU. You use CL_CTMENU in different ways depending on the context:

If you use a context menu in a control [Page 661], whether or not you need to create objects of the class CL_CTMENU depends on the wrapper of the class. (The relevant functions may already be encapsulated in the control class.) Normally, control users do not have to create their own context menus. This ensures that no conflicts occur with the event handling of the control. For further information, refer to the documentation of the individual control classes.

When you define a context menu on a screen (or list [Page 866]), the relevant objects of class CL_CTMENU are created automatically by the runtime environment, not explicitly in the program. References to the object are passed as the parameters of special callback routines in the ABAP program.

Context Menus for Elements on Screens

In order to link a context menu with one of the screen elements above, you only need to enter an ID <context> in the *Context menu* field in the element attributes in the Screen Painter. If you do not define a context menu for a particular screen element, it inherits the context menu from the hierarchically next-highest element. For example, all screen elements in a group box that do not have their own context menu would inherit the context menu of the group box. The highest hierarchy level is the default context menu, containing all of the key settings of the current dialog status.

If a screen element is linked with a context menu – either its own or one that it has inherited, a special subroutine

ON_CTMENU_<context>

is called in the ABAP program when the user clicks the right-hand mouse button. The PAI event is not triggered. You use this subroutine (callback routine) to define the context menu dynamically. You must program it in the processing logic. If the subroutine does not exist, the context menu is not displayed.

You can link the same context menu <context> to any number of screen elements. They then all work with the same subroutine.

Defining Context Menus in the Processing Logic

For each context menu that you want to call for an element on a screen, you need to program a corresponding callback routine:

```
FORM ON_CTMENU_<context> USING <l_menu> TYPE REF TO cl_ctmenu.
  ...
ENDFORM.
```

Each routine must have a single USING parameter, typed as a reference variable to class CL_CTMENU. For each context menu assigned to an element on a screen, the runtime environment automatically creates an object of the class. When the user requests the context menu by clicking the right-hand mouse button, the system calls the corresponding subroutine and passes a reference to the corresponding object to the formal parameter.

When the object is passed it is initial – the context menu contains no entries. In the subroutine, you can work with the methods of the object (as listed above) to construct the context menu dynamically.

Using Predefined Context Menus

As well as dialog statuses and dialog box statuses, there is a third kind of GUI status [Page 548] that you can define in the Menu Painter, namely a context menu. To find out how to create context menus in the Menu Painter, refer to Creating Context Menus [Ext.].

Predefined context menus allow you to make groups of statically-defined function coded available context-specifically. The method LOAD_GUI_STATUS allows you to load a context menu from any ABAP program into a local context menu in a program. As a rule, you use predefined context menus to reuse the function codes from a dialog status with the same semantics, but context-specifically. Once you have loaded a predefined context menu into a local context menu in a program, you can modify it in any way (append other predefined context menus, add or remove functions, add other context menus).

Defining New Context Menus

You can create new context-specific menus either by modifying existing ones or by constructing new menus.

You can add any number of new functions to a context menu. When you add a new function, you must specify the function text, function code, and function type (for example, E for an unconditional module call [Page 568]).

However, you can also add any other local context menu from the program. In this case, you only have to pass a reference to another context menu (see example below). You can create a collection of context menu objects in your program and use and combine them as necessary. You can also construct submenus. You can have deep-nested menus by adding submenus to existing submenus.

Ergonomic Guidelines

When you create context menus, you should observe the following rules:

- The functions in a context menu should be a subset of the functions in the program You can ensure this by using predefined context menus
- Context menus should not contain more than ten entries at a single level.
- If you use a context menu for a screen element, it should contain all of the functions possible for that element, but at least the standard comments such as *Select*, *Copy*, *Cut*, and *Paste*.
- The sequence of the functions should be as follows: Object-specific commands, copy commands, other commands.
- You should not duplicate functions that can be selected using the left-hand mouse button in a context menu.

Displaying the Context Menu

Once you have defined the context menu dynamically in the callback routine, the system displays it on the screen immediately. When the user chooses a function from the menu, the system triggers the PAI event and places the corresponding function code in SY-UCOMM and the OK CODE field.

Example

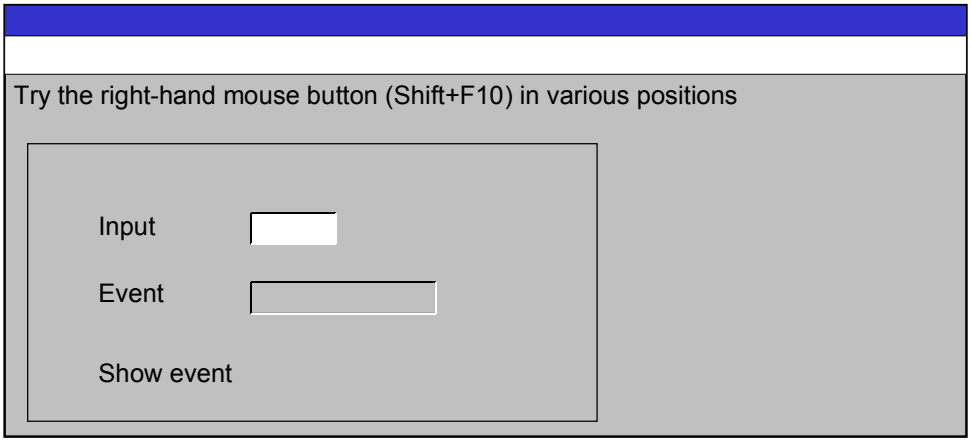
The following example shows some of the technical possibilities for creating context menus, but does not necessarily observe all of the style guidelines.

```
REPORT demo_dynpro_context_menu.

DATA: field1 TYPE i VALUE 10,
      field2 TYPE p DECIMALS 4.

DATA: prog TYPE sy-repid,
      flag(1) TYPE c VALUE 'X'.

DATA: ok_code TYPE sy-ucomm,
      save_ok TYPE sy-ucomm.

prog = sy-repid.

CALL SCREEN 100.

MODULE status_0100 OUTPUT.
  SET TITLEBAR 'TIT100'.
  IF flag = 'X'.
    SET PF-STATUS 'SCREEN_100' EXCLUDING 'REVEAL'.
  ELSEIF flag = ' '.
    SET PF-STATUS 'SCREEN_100' EXCLUDING 'HIDE'.
  ENDIF.
  LOOP AT SCREEN.
    IF screen-group1 = 'MOD'.
      IF flag = 'X'.
        screen-active = '1'.
      ELSEIF flag = ' '.
        screen-active = '0'.
      ENDIF.
      MODIFY SCREEN.
    ELSEIF screen-name = 'TEXT_IN_FRAME'.
      IF flag = 'X'.
        screen-active = '0'.
      ELSEIF flag = ' '.
        screen-active = '1'.
      ENDIF.
      MODIFY SCREEN.
    ENDIF.
  ENDLOOP.
ENDMODULE.

MODULE cancel INPUT.
  LEAVE PROGRAM.
ENDMODULE.

MODULE user_command_0100.
  save_ok = ok_code.
  CLEAR ok_code.
  CASE save_ok.
    WHEN 'HIDE'.
      flag = ' '.
```

```
    WHEN 'REVEAL'.
      flag = 'X'.
    WHEN 'SQUARE'.
      field2 = field1 ** 2.
    WHEN 'CUBE'.
      field2 = field1 ** 3.
    WHEN 'SQUAREROOT'.
      field2 = field1 ** ( 1 / 2 ).
    WHEN 'CUBICROOT'.
      field2 = field1 ** ( 1 / 3 ).
  ENDCASE.
ENDMODULE.

*******************************************************
* Callback-Routines
*******************************************************

FORM on_ctmenu_text USING l_menu TYPE REF TO cl_ctmenu.
  CALL METHOD:l_menu->load_gui_status
                       EXPORTING program = prog
                                 status  = 'CONTEXT_MENU_1'
                                 menu    = l_menu.
ENDFORM.

FORM on_ctmenu_frame USING l_menu TYPE REF TO cl_ctmenu.
  CALL METHOD:l_menu->load_gui_status
                       EXPORTING program = prog
                                 status  = 'CONTEXT_MENU_2'
                                 menu    = l_menu,
              l_menu->load_gui_status
                       EXPORTING program = prog
                                 status  = 'CONTEXT_MENU_1'
                                 menu    = l_menu,
              l_menu->set_default_function
                       EXPORTING fcode = 'HIDE'.
ENDFORM.

FORM on_ctmenu_reveal USING l_menu TYPE REF TO cl_ctmenu.
  CALL METHOD:l_menu->load_gui_status
                       EXPORTING program = prog
                                 status  = 'CONTEXT_MENU_3'
                                 menu    = l_menu,
              l_menu->load_gui_status
                       EXPORTING program = prog
                                 status  = 'CONTEXT_MENU_1'
                                 menu    = l_menu,
              l_menu->set_default_function
                       EXPORTING fcode = 'REVEAL'.
ENDFORM.

FORM on_ctmenu_input USING l_menu TYPE REF TO cl_ctmenu.
  DATA calculate_menu TYPE REF TO cl_ctmenu.
  CREATE OBJECT calculate_menu.
  CALL METHOD: calculate_menu->add_function
                       EXPORTING fcode = 'SQUARE'
                                 text  = text-001,
```

```
                    calculate_menu->add_function
                            EXPORTING fcode = 'CUBE'
                                      text  = text-002,
                    calculate_menu->add_function
                            EXPORTING fcode = 'SQUAREROOT'
                                      text  = text-003,
                    calculate_menu->add_function
                            EXPORTING fcode = 'CUBICROOT'
                                      text  = text-004,
                    l_menu->add_submenu
                            EXPORTING menu = calculate_menu
                                      text = text-005.
ENDFORM.
```

The next screen (statically defined) for screen 100 is itself. It has the following layout:



The relevant extract of the element list is as follows:

Name	Type	Contents	Context menu
TEXT	Text	Try the...	TEXT
FRAME	Frame		FRAME
TEXT1	Text	Input	INPUT
FIELD1	I/O		INPUT
TEXT2	Text	Result	
FIELD2	I/O		
TEXT_IN_FRAME	Text	Show event	REVEAL

Elements TEXT2 and FIELD2 have no context menus of their own. They inherit the context menu FRAME from the group box. They are assigned to modification group MOD.

The flow logic is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE status_0100.

PROCESS AFTER INPUT.
  MODULE cancel AT EXIT-COMMAND.
  MODULE user_command_0100.
```

The following function codes and GUI status are assigned to this ABAP program:

Function codes	BACK	CANCEL	EXIT	HIDE	REVEAL
Dialog status					
SCREEN_100	X	X	X	X	X
Context menus					
CONTEXT_MENU_1	X				
CONTEXT_MENU_2				X	
CONTEXT_MENU_3					X

The table shows the function codes that each GUI status contains.

The dialog status SCREEN_!00 is set statically in the PBO. Function codes HIDE and REVEAL are displayed or hidden, depending on the contents of the FLAG field.

The context menus for the screen elements are constructed in the callback routines as follows:

TEXT:
Loads the static context menu CONTEXT_MENU_1 without modification. This context menu has a single line- *Cancel*.

FRAME:
Constructed from the static context menus CONTEXT_MENU_2 and CONTEXT_MENU_1. This has two lines – *Hide result* and *Cancel*. The entry for function code HIDE is highlighted.

REVEAL:
Constructed from the static context menus CONTEXT_MENU_3 and CONTEXT_MENU_1. This has two lines – *Show result* and *Cancel*. The entry for function code REVEAL is highlighted.

INPUT:
Constructed by including the four-line local context menu CALCULATE_MENU as a submenu. To do this, the program declares a local reference variable with reference to class CL_CTMENU, then creates the object and adds the function codes SQUARE, CUBE, SQUAREROOT, and CUBICROOT. When we include it in the context menu for INPUT, we must specify a text for the menu entry behind which it stands.

When a user runs the program and right-clicks the first line, the context menu TEXT is displayed. When he or she right-clicks the second line, context menu INPUT is displayed, and for the third line, FRAME is displayed. The fourth line is hidden by the program. For all other areas of the screen, the system displays the standard context menu, with all of the static function codes plus F1 and F4.

When the user chooses one of the new dynamic functions, the system calculates using the number in input field FIELD1 and places the result in FIELD2.

When the user chooses *Hide result* (HIDE), the program modifies the screen dynamically. The fourth line becomes visible, and the user can now display the context menu REVEAL.

Subscreens

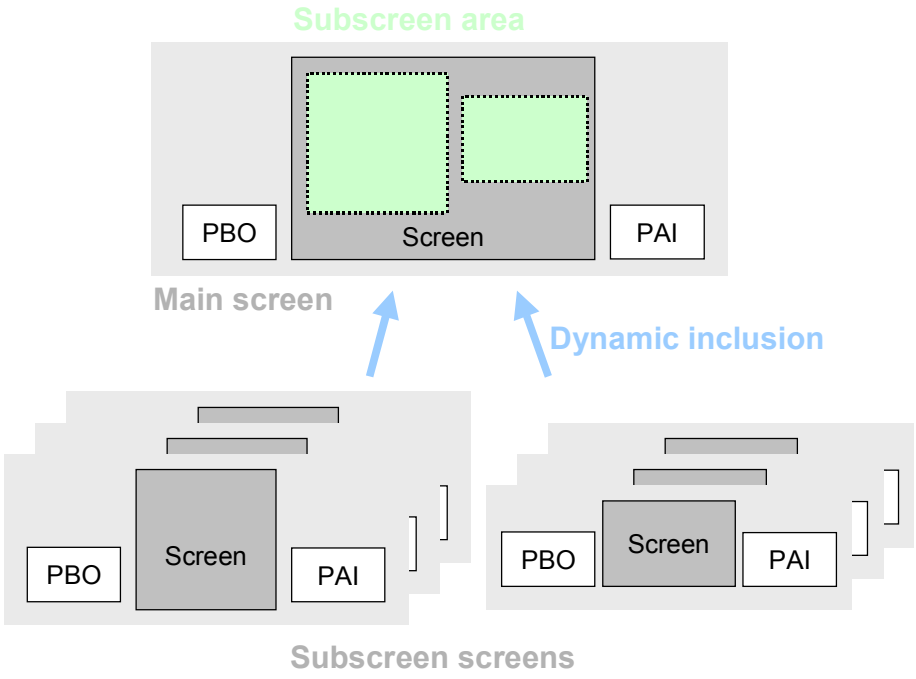
Selektionsbilder als Subscreens [Page 754]

Subscreens allow you to embed one screen within another at runtime. The term subscreen applies both to the screen that you embed, and the area on the main screen in which you place it. This section is about subscreen areas. The actual screens that you embed are called subscreen screens. When you use a subscreen, the flow logic of the embedded screen is also embedded in the flow logic of the main screen. Using subscreens on screens is like using includes in ABAP programs.

Subscreens allow you to expand the content of a screen dynamically at runtime. For example, screen exits, which are part of the enhancement concept, use the subscreen technique. Some complex screen elements, like tabstrip controls [Page 653], also use them.

To use a subscreen, you must:

1. Define the subscreen area(s) on a screen
2. Define suitable subscreen screens
3. Include the subscreen screen in the subscreen area.



Defining Subscreen Areas

You define subscreen areas using the Screen Painter in the layout of the screen on which you want to embed a subscreen. Each subscreen area on a screen has a unique name, and a position, length, and height. Subscreen areas may not overlap either with each other or with other screen elements. You can also specify whether a subscreen area can be resized vertically or horizontally when the user resizes the window. If the area supports resizing, you can specify a

minimum size for it. If the resizing attributes are selected, the PAI event is triggered whenever the user resizes the main screen.

Defining Subscreen Screens

You can create subscreen screens either in the same program or a different program. To create a subscreen screen, enter the screen type *Subscreen* in the screen attributes. The statically-defined next screen must be the number of the subscreen itself. Choose a size for the screen, making sure that it fits within the subscreen area into which you want to place it. If the subscreen screen is too big for the subscreen area, only the top left-hand corner of it will be displayed.

You create the layout, element list, and flow logic of a subscreen screen in the same way as a normal screen. Subscreens may also include other subscreens. However, the following restrictions apply:

- You should arrange the screen elements so that they are not truncated if a subscreen area is too small.
- If you want to create several subscreen screens in an ABAP program, you should make sure that the individual screen elements have names unique among the subscreens. If the subscreen screens belong to the same program as the main screen, you should also make sure that names are not used twice there. Otherwise, you must separate data transported from the screen in your ABAP program after each user action (see example).
- Subscreens cannot have their own OK_CODE field. Function codes linked to user actions on subscreens are placed in the OK_CODE field of the main screen. This also applies to subscreen screens defined in a different program to the main screen.
- The flow logic of a subscreen screen may not contain a MODULE ... AT EXIT-COMMAND statement. Type E functions may only be handled in the main screen.
- The flow logic of a subscreen screen may not contain any dialog modules containing the statements SET TITLEBAR, SET PF-STATUS, SET SCREEN, LEAVE SCREEN, or LEAVE TO SCREEN. Any of these statements causes a runtime error. You cannot change the GUI status of a main screen in a subscreen screen.

Including Subscreen Screens

You include a subscreen screen using the CALL SUBSCREEN statement in the flow logic of the main screen.

To include a subscreen screen in the subscreen area of the main screen and call its PBO flow logic, use the following statement in the PBO event of the main screen:

```
PROCESS BEFORE OUTPUT.
...
  CALL SUBSCREEN <area> INCLUDING [<prog>] <dynp>.
...
```

This statement assigns the subscreen screen with number <dynp> to the subscreen area called <area>. You can also specify the program in which the subscreen screen is defined (optional). If you do not specify the program explicitly, the system looks for the subscreen screen in the same ABAP program as the main program. If it does not find a corresponding subscreen screen, a runtime error occurs. The PBO flow logic of the subscreen screen is also included at the same point. This can call PBO modules of the ABAP program in which the subscreen screen is defined. At the end of the subscreen PBO, the global fields from the program are passed to any

identically-named screen fields in the subscreen screen. The PBO flow logic of the subscreen screen can itself include further subscreens.

The name <area> of the subscreen area must be entered directly without inverted commas. You can specify the names <prog> and <dynp> either as literals or variables. If you use variables, you must declare and fill identically-named variables in the ABAP program. The screen number <dynp> must be 4 characters long. If you do not assign a subscreen screen to an area, it remains empty.

To call the PAI flow logic of the subscreen screen, use the following statement in the PAI flow logic of the main screen:

```
PROCESS AFTER INPUT.
...
  CALL SUBSCREEN <area>.
...
```

This statement includes the PAI flow logic of the subscreen screen included in the subscreen area <area> in the PBO event. This can call PAI modules of the ABAP program in which the subscreen screen is defined. Data is transported between identically-named fields in the subscreen screen and the ABAP program either when the PAI event is triggered, or at the corresponding FIELD statements in the PAI flow logic of the subscreen screen.

You cannot place the CALL SUBSCREEN statement between CHAIN and ENDCHAIN or LOOP and ENDLOOP (see table controls [Page 669]). While a subscreen screen is being processed, the system field SY-DYNNR contains its screen number. Its contents therefore change when the CALL SUBSCREEN statement occurs and when you return to the main screen.

When you use subscreen screens, remember that the data from their input/output fields is transported to and from the programs in which they are defined. The function codes of user actions on the screen, on the other hand, are always placed in the OK_CODE field of the main screen, and transported into the program in which it is defined. You should therefore name the function codes of your subscreen screens differently from those of the main screen.

If, for encapsulation reasons, you define subscreen screens in other ABAP programs, you must ensure that the required global data of your ABAP programs is transported into the program of the calling screen after you have called their PAI flow logic.

- For example, you can define subscreen screens in function groups and pass their global data to and from function module parameters. You must then call the function modules in appropriate dialog modules in the main program to transport the data.
- You can also export the global data from the main program as a data cluster to ABAP memory [Page 363] before the PBO of a subscreen screen and import it into a PBO module of the subscreen.

Example

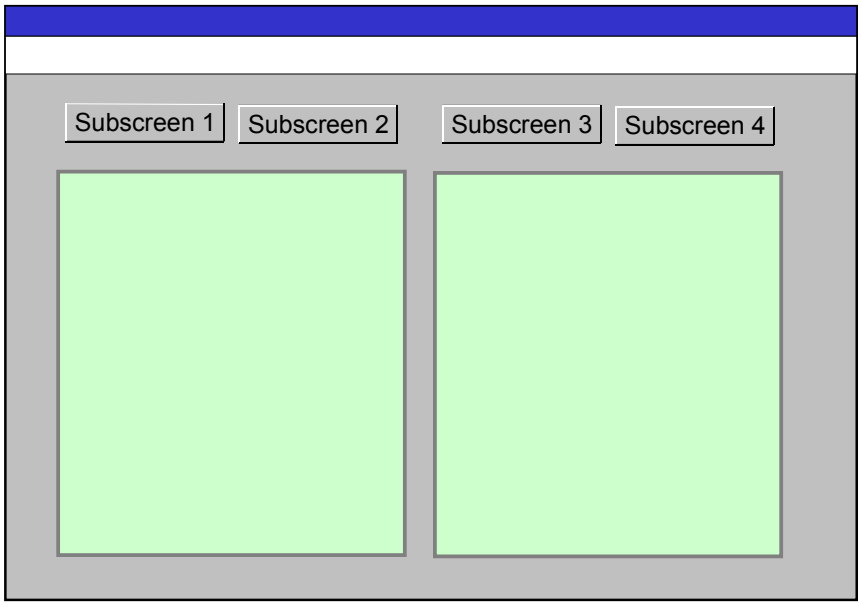
Subscreens

```
REPORT DEMO_DYNPRO_SUBSCREENS.

DATA: OK_CODE LIKE SY-UCOMM,
      SAVE_OK LIKE SY-UCOMM.
```

```
DATA: NUMBER1(4) TYPE N VALUE '0110',
      NUMBER2(4) TYPE N VALUE '0130',
      FIELD(10), FIELD1(10), FIELD2(10).

CALL SCREEN 100.

MODULE STATUS_100 OUTPUT.
  SET PF-STATUS 'SCREEN_100'.
ENDMODULE.

MODULE FILL_0110 OUTPUT.
  FIELD = 'Eingabe 1'(001).
ENDMODULE.

MODULE FILL_0120 OUTPUT.
  FIELD = FIELD1.
ENDMODULE.

MODULE FILL_0130 OUTPUT.
  FIELD = 'Eingabe 2'(002).
ENDMODULE.

MODULE FILL_0140 OUTPUT.
  FIELD = FIELD2.
ENDMODULE.

MODULE CANCEL INPUT.
  LEAVE PROGRAM.
ENDMODULE.

MODULE SAVE_OK INPUT.
  SAVE_OK = OK_CODE.
  CLEAR OK_CODE.
ENDMODULE.

MODULE USER_COMMAND_0110 INPUT.
  IF SAVE_OK = 'OK1'.
    NUMBER1 = '0120'.
    FIELD1 = FIELD.
    CLEAR FIELD.
  ENDIF.
ENDMODULE.

MODULE USER_COMMAND_0130 INPUT.
  IF SAVE_OK = 'OK2'.
    NUMBER2 = '0140'.
    FIELD2 = FIELD.
    CLEAR FIELD.
  ENDIF.
ENDMODULE.

MODULE USER_COMMAND_100 INPUT.
  CASE SAVE_OK.
    WHEN 'SUB1'.
      NUMBER1 = '0110'.
    WHEN 'SUB2'.
      NUMBER1 = '0120'.
      CLEAR FIELD1.
```

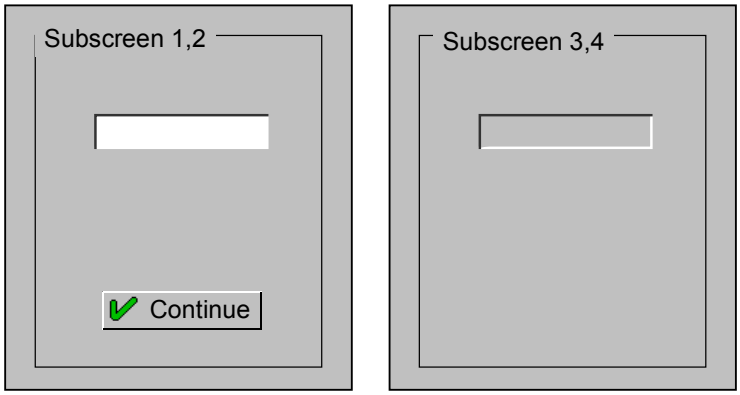
```
  WHEN 'SUB3'.
    NUMBER2 = '0130'.
  WHEN 'SUB4'.
    NUMBER2 = '0140'.
    CLEAR FIELD2.
 ENDCASE.
ENDMODULE.
```

The next screen (statically defined) for screen 100 is itself. It has the following layout:



There are four pushbuttons with the function codes SUB1 to SUB4, and two subscreen areas AREA1 and AREA2.

In the same ABAP program, there are four subscreen screens 110 to 140. Each of these fits the subscreen area exactly. The layout is:



Subscreens 110 and 130 Subscreens 120 and 140

Subscreens

The input/output field of all four subscreen screens has the name FIELD. The function codes of the pushbuttons on the subscreen screens 110 and 130 are OK1 and OK2.

The screen flow logic for screen 100 is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_100.
  CALL SUBSCREEN: AREA1 INCLUDING SY-REPID NUMBER1,
                  AREA2 INCLUDING SY-REPID NUMBER2.

PROCESS AFTER INPUT.
  MODULE CANCEL AT EXIT-COMMAND.
  MODULE SAVE_OK.
  CALL SUBSCREEN: AREA1,
                  AREA2.
  MODULE USER_COMMAND_100.
```

The screen flow logic of subscreen screens 110 and 130 is:

```
PROCESS BEFORE OUTPUT.
  MODULE FILL_0110|0130.

PROCESS AFTER INPUT.
  MODULE USER_COMMAND_0110|0130.
```

Die Bildschirmablauflogik der Subscreen-Dynpros 120 und 140 ist:

```
PROCESS BEFORE OUTPUT.
  MODULE FILL_0120|0150.

PROCESS AFTER INPUT.
```

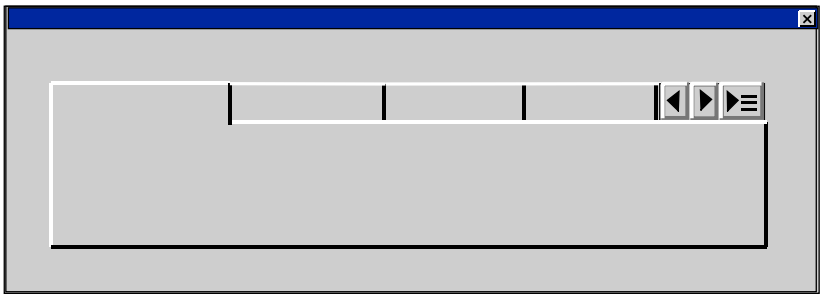
When you run the program, a screen appears on which subscreens 110 and 130 are displayed. The pushbuttons on the main screen allow you to choose between two subscreen screens for each screen area. The pushbuttons on the subscreens allow you to transfer the data from subscreens 110 and 130 to subscreens 120 and 140.

Since the same field name FIELD is used on all subscreens, the identically-named ABAP field is transferred **more than once** in each PBO and PAI event of the main screen. For this reason, the values have to be stored in the auxiliary fields FIELD1 and FIELD2 in the ABAP program.

The pushbuttons on the subscreen screens have different function codes, and they are handled normally in an ABAP field. If the function codes had had the same names, it would again have been necessary to use auxiliary fields.

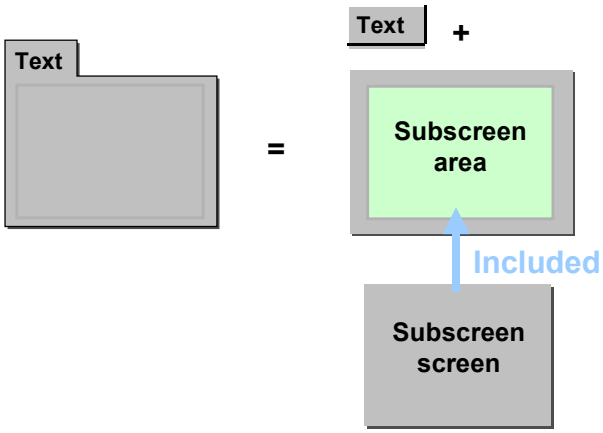
Tabstrip Controls

A tabstrip control is a screen object consisting of two or more pages. Each tab page consists of a tab title and a page area. If the area occupied by the tabstrip control is too narrow to display all of the tab titles, a scrollbar appears, allowing you to reach the titles that are not displayed. There is also a pushbutton that allows you to display a list of all tab titles.



Tabstrip controls allow you to place a series of screens belonging to an application on a single screen, and to navigate between them easily. The recommended uses and ergonomic considerations for tabstrip controls are described in the Tabstrip Control section of the SAP Style Guide.

From a technical point of view, a tab page is a subscreen [Page 647] with a pushbutton [Page 542] assigned to it, which is displayed as the tab title.



The tabstrip control is the set of all the tab pages. Tabstrip controls are therefore subject to the same restrictions as subscreens. In particular, you cannot change the GUI status when you switch between pages in the tabstrip control. However, they are fully integrated into the screen environment, so present no problems with batch input.

To use a tabstrip control on a screen, you must be using a SAPgui with Release 4.0 or higher, and its operating system must be Motif, Windows 95, MacOS, or Windows NT with version 3.51 or higher.

Tabstrip Controls

When you create a tabstrip control, you must:

1. Define the tab area on a screen and the tab titles.
2. Assign a subscreen area to each tab title.
3. Program the screen flow logic.
4. Program the ABAP processing logic.

You must then decide whether you want to page through the tabstrip control at the SAPgui or on the application server. In the first case, each tab page has its own subscreen. In the second, there is a single subscreen area that is shared by all tab pages.

Defining the Tabstrip Control Area and Tab Titles

You define both the tabstrip area and the tab titles in the screen layout.

The tabstrip area has a unique name and a position, length, and height. You can also specify whether the tabstrip area can be resized vertically or horizontally when the user resizes the window. If the area supports resizing, you can specify a minimum size for it.

When you define a tabstrip area, it already has two tab titles. Tab titles are technically exactly the same as pushbuttons. To create additional tab titles, simple create pushbuttons in the row containing the tab titles. Tab titles have the same attributes as pushbuttons, that is, each has a name, a text, and a function code. You can also use icons and dynamic texts with tab titles.

Assigning a Subscreen Area to a Tab Title

You must assign a subscreen area to each tab title. There are two ways of doing this:

Paging in the SAPgui

You need to assign a separate subscreen area to each tab title, and define the function codes of the tab titles with type **P** (local GUI function). In the screen flow logic, you call all the subscreens in the PBO event. This means that all of the tab pages reside locally on the SAPgui.

When the user chooses a tab title, paging takes place within the SAPgui. In this respect, the tabstrip control behaves like a single screen. In particular, the PAI event is **not** triggered when the user chooses a tab title, and **no** data is transported. While this improves the performance of your tabstrip control, it also has the negative effect that when the user does trigger the PAI event, all of the input checks [Page 577] for all of the subscreens are performed. This means that when the user is working on one tab page, the input checks may jump to an unfilled mandatory field on another page.

Local paging at the SAPgui is therefore most appropriate for screens that display data rather than for input screens.

Paging on the Application Server

One subscreen area is shared by all tab titles and called in the PBO event. You define the function codes of the individual tab titles without a special function type. When the user chooses a tab page, the PAI event is triggered, and you must include a module in your flow logic that activates the appropriate tab page and assigns the correct subscreen to the subscreen area.

Since the PAI event is triggered each time the user chooses a tab title, this method is less economical for the application server, but the input checks [Page 577] that are performed only affect the current tab page.

Procedure in Either Case

You create the subscreen areas within the tabstrip area. You assign the subscreen areas to one or more tab titles in the Screen Painter by selecting one or more titles. You can also assign a subscreen area to a tab title in the tab title attributes by entering the name of the subscreen area in the *Reference field* attribute.

The procedure for the alphanumeric Screen Painter is described under Creating Tabstrip Controls [Ext.].

If you are paging at the SAPgui, create a subscreen area for each tab title. If you are paging at the application server, select all tab titles and create a single subscreen area. The subscreen areas may not cover the top line of the tab area. However, within a tab area, more than one subscreen area can overlap.

Programming the Flow Logic

In the flow logic, all you have to do by hand is include the correct subscreens [Page 647]. The screen flow and data transport to the ABAP program is the same as for normal subscreens. There are two ways of programming the screen flow logic, depending on how you have decided to page through the tabstrip control.

Paging in the SAPgui

When you page in the SAPgui, you must include a subscreen for each subscreen area:

```
PROCESS BEFORE OUTPUT.
  ...
  CALL SUBSCREEN: <area1> INCLUDING [<prog1>] <dynp1>,
                  <area2> INCLUDING [<prog2>] <dynp2>,
                  <area3> INCLUDING [<prog3>] <dynp3>,
                  ...
  ...
PROCESS AFTER INPUT.
  ...
  CALL SUBSCREEN: <area1>,
                  <area2>,
                  <area3>,
                  ...
  ...
```

Paging on the Application Server

When you page on the application server, you only have to include a subscreen for the one subscreen area:

```
PROCESS BEFORE OUTPUT.
  ...
  CALL SUBSCREEN <area> INCLUDING [<prog>] <dynp>.
  ...
PROCESS AFTER INPUT.
  ...
  CALL SUBSCREEN <area>.
  ...
```

Handling in the ABAP Program

Before you can use a tabstrip control in your ABAP program, you must create a control for each control in the declaration part of your program using the following statement:

CONTROLS <ctrl> TYPE TABSTRIP.

where <ctrl> is the name of the tabstrip area on a screen in the ABAP program. The control allows the ABAP program to work with the tabstrip control. The statement declares a structure with the name <ctrl> . The only component of this structure that you need in your program is called ACTIVETAB.

- Use in the PBO event

  Before the screen is displayed, you use the control to set the tab page that is currently active. To do this, assign the function code of the corresponding tab title to the component ACTIVETAB:

  ```
  <ctrl>-ACTIVETAB = <fcode>.
  ```

  When you page at the SAPgui, you only need to do this **once** before the screen is displayed. This initializes the tabstrip control. The default active tab page is the first page. After this, the page activated when the user chooses a tab title is set within SAPgui.

  When you page on the application server, you must assign the active page both before the screen is displayed for the first time, and **each time the user pages**. At the same time, you must set the required subscreen screen.

  You can suppress a tab page dynamically by setting the ACTIVE field of table SCREEN [Page 612] to 0 for the corresponding tab title.

- Use in the PAI event

  In the PAI event, ACTIVETAB contains the function code of the last active tab title on the screen.

  When you page in the SAPgui, this allows you to find out the page that the user can currently see. When you page at the application server, the active tab page is controlled by the ABAP program anyway.

The OK_CODE field behaves differently according to the paging method:

- Paging in the SAPgui

  When you page in the SAPgui, the PAI event is not triggered when the user chooses a tab title, and the OK_CODE field is not filled. The OK_CODE field is only filled by user actions in the GUI status or when the user chooses a pushbutton either outside the tabstrip control or on one of the subscreens.

- Paging on the application server

  If you are paging at the application server, the PAI event is triggered when the user chooses a tab title, and the OK_CODE field is filled with the corresponding function code.

  To page through the tabstrip control, you must assign the function code to the ACTIVETAB component of the control:

  ```
  <ctrl>-ACTIVETAB = <ok_code>.
  ```

This statement overwrites the function code of the last active tab page with that of the new tab title. At the same time, you must ensure that the correct subscreen is inserted in the subscreen area.

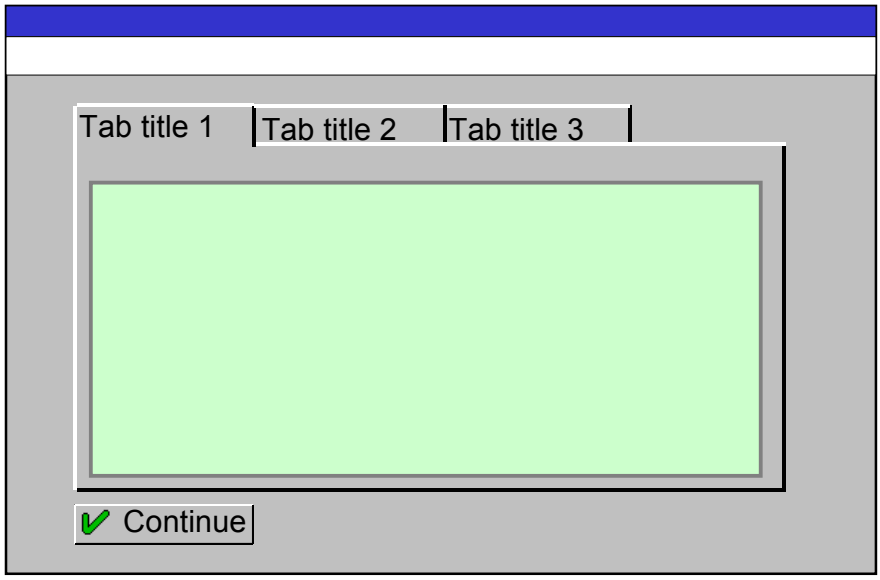
Otherwise, tabstrip controls are handled like normal subscrens in ABAP programs, that is, the ABAP program of a subscreen screen must contain the dialog modules called from the flow logic of the subscreen.

Examples

Tabstrip control, paging at SAPgui

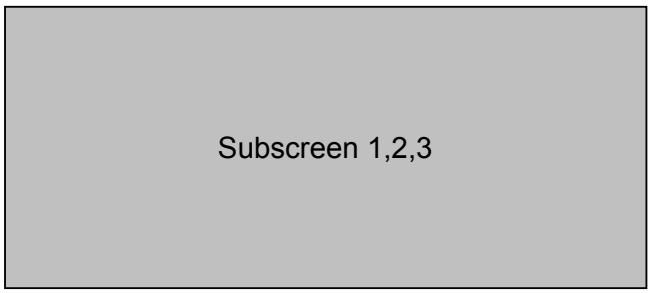
```
REPORT DEMO_DYNPRO_TABSTRIP_LOCAL.

CONTROLS MYTABSTRIP TYPE TABSTRIP.

DATA: OK_CODE TYPE SY-UCOMM,
      SAVE_OK TYPE SY-UCOMM.

MYTABSTRIP-ACTIVETAB = 'PUSH2'.

CALL SCREEN 100.

MODULE STATUS_0100 OUTPUT.
  SET PF-STATUS 'SCREEN_100'.
ENDMODULE.

MODULE CANCEL INPUT.
  LEAVE PROGRAM.
ENDMODULE.

MODULE USER_COMMAND INPUT.
  SAVE_OK = OK_CODE.
  CLEAR OK_CODE.
  IF SAVE_OK = 'OK'.
    MESSAGE I888(SABAPDOCU) WITH 'MYTABSTRIP-ACTIVETAB ='
                                 MYTABSTRIP-ACTIVETAB.
  ENDIF.
ENDMODULE.
```

The next screen (statically defined) for screen 100 is itself. It has the following layout:



The screen contains a tabstrip area called MYTABSTRIP with three tab titles PUSH1, PUSH2 and PUSH3. The function codes have the same name, and all have the function type P. One of the subscreen areas SUB1 to SUB3 is assigned to each tab title. The pushbutton has the name BUTTON and the function code 'OK'.

In the same ABAP program, there are three subscreen screens 110 to 130. Each of these fits the subscreen area exactly. The layout is:



The screen flow logic for screen 100 is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0100.
  CALL SUBSCREEN: SUB1 INCLUDING SY-REPID '0110',
                  SUB2 INCLUDING SY-REPID '0120',
                  SUB3 INCLUDING SY-REPID '0130'.

PROCESS AFTER INPUT.
  MODULE CANCEL AT EXIT-COMMAND.
  CALL SUBSCREEN: SUB1,
                  SUB2,
                  SUB3.
  MODULE USER_COMMAND.
```

The screen flow logic of subscreens 110 to 130 does not contain any module calls.

When you run the program, a screen appears on which the second tab page is active, since the program sets the ACTIVETAB component of the structure MYTABSTRIP to PUSH2 before the screen is displayed. The user can page through the tabstrip control without the PAI event being triggered. One of the three subscreens is included on each tab page.

When the user chooses *Continue*, the PAI event is triggered, and an information message displays the function code of the tab title of the page that is currently active.

Tabstrip control with paging on the application server.

```
REPORT DEMO_DYNPRO_TABSTRIP_LOCAL.

CONTROLS MYTABSTRIP TYPE TABSTRIP.

DATA: OK_CODE TYPE SY-UCOMM,
      SAVE_OK TYPE SY-UCOMM.

DATA  NUMBER TYPE SY-DYNNR.

MYTABSTRIP-ACTIVETAB = 'PUSH2'.
NUMBER = '0120'.

CALL SCREEN 100.

MODULE STATUS_0100 OUTPUT.
  SET PF-STATUS 'SCREEN_100'.
ENDMODULE.

MODULE CANCEL INPUT.
  LEAVE PROGRAM.
ENDMODULE.

MODULE USER_COMMAND INPUT.
  SAVE_OK = OK_CODE.
  CLEAR OK_CODE.
  IF SAVE_OK = 'OK'.
    MESSAGE I888(SABAPDOCU) WITH 'MYTABSTRIP-ACTIVETAB ='
                                 MYTABSTRIP-ACTIVETAB.
  ELSE.
    MYTABSTRIP-ACTIVETAB = SAVE_OK.
    CASE SAVE_OK.
      WHEN 'PUSH1'.
        NUMBER = '0110'.
      WHEN 'PUSH2'.
        NUMBER = '0120'.
      WHEN 'PUSH3'.
        NUMBER = '0130'.
    ENDCASE.
  ENDIF.
ENDMODULE.
```

The statically-defined next screen for screen 100 is itself, and its layout is the same as in the above example. However, the function codes of the three tab titles have the function type <blank> and they all share a single subscreen area SUB.

The same subscreen screens 110 to 130 are defined as in the last example.

The screen flow logic for screen 100 is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0100.
  CALL SUBSCREEN SUB INCLUDING SY-REPID NUMBER.

PROCESS AFTER INPUT.
  MODULE CANCEL AT EXIT-COMMAND.
  CALL SUBSCREEN SUB.
  MODULE USER_COMMAND.
```

In this example, the program includes a subscreen screen in the subscreen area SUB dynamically during the PBO event.

The screen flow logic of subscreens 110 to 130 does not contain any module calls.

This example has the same function as the previous example, but the paging within the tabstrip control is implemented on the application server. Each time the user chooses a tab title, the function code from the OK_CODE field is assigned to the ACTIVETAB component of structure MYTABSTRIP. At the same time, the variable NUMBER is filled with the screen number of the subscreen that has to be displayed in the subscreen area SUB of the tabstrip control.

Custom Controls

A custom control is an area on a screen. You create them in the Screen Painter, and, like all other screen objects, they have a unique name. You use custom controls to embed controls. A control is a software component on the presentation server, which can be either an ActiveX control or a JavaBean, depending on the SAPgui you are using. They allow you to perform tasks, such as editing texts, locally on the presentation server. The control is driven by the application logic, which still runs on the application server.

The SAP Control Framework

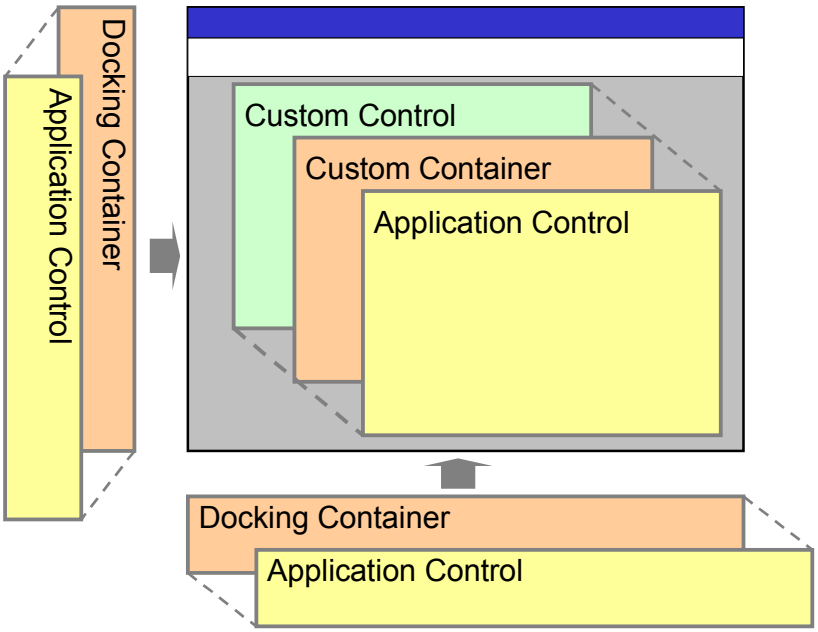
The controls on the presentation server and the ABAP application programs on the application server communicate using the SAP Control Framework [Ext.]. This is programmed in ABAP Objects [Page 1291], and contains a set of global classes that you can find in the Class Browser under *Basis* → *Frontend services*. These classes encapsulate the communication between the application server and presentation server, which is implemented using Remote Function Call.

All application controls are encapsulated in a global class. You can find the SAP Basis controls in the Class Browser under *Basis* → *Frontend Services* or *Basis* → *Component Integration*. Programs that use controls on a screen work with the methods and events of the global classes that encapsulates them.

Container Controls

Before you can work with a custom control on a screen, you must assign a SAP Container Control [Ext.] to it. Container controls are instances of special global classes from the SAP Control Framework. The global class for custom controls is called CL_GUI_CUSTOM_CONTAINER. To link a custom control to a container control, pass the custom control name to the CONTAINER_NAME parameter of the container control constructor when you instantiate it.

As well as using custom containers, you can link controls to a screen using a SAP Docking Container. This is encapsulated in the global class CL_GUI_DOCKING_CONTAINER. The SAP Docking Container does not place the control within a screen. Instead, it attaches it to one of the four edges. You can nest containers. For example, you can use the SAP Splitter Container (classes CL_GUI_EASY_SPLITTER_CONTAINER or CL_GUI_SPLITTER_CONTAINER) within other containers. This allows you to split a custom control or docking control into more than one area, allowing you to embed more than one control.



Application Controls

You must also create instances for the application controls that you want to place within your container - for example, a SAP Textedit Control (class CL_GUI_TEXTEDIT) or a SAP Tree Control (for which there is more than one global class - an example is CL_GUI_SIMPLE_TREE). When you instantiate the control, you pass a reference to the container in which you want to place it to the PARENT parameter of its constructor method. The container may be an instance of the class CL_GUI_CUSTOM_CONTAINER, but can also be an instance of one of the other SAP Container controls.

Control Methods

For information about control methods and their documentation, refer to the class definitions in the Class Builder or the SAP Library documentation. To minimize the network load between the application and presentation servers, method calls are buffered in the automation queue [Ext.] before being sent to the presentation server at defined synchronization points. One of the automatic synchronization points is the end of PBO processing. You can force a synchronization point in your program by calling a method that is not buffered, or by calling the static method FLUSH.

Control Events

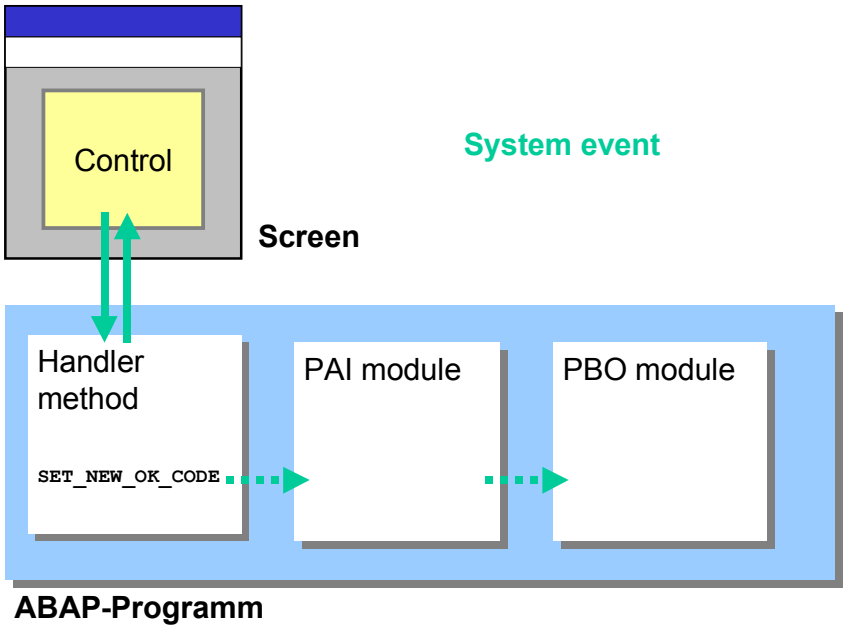
Unlike screens, on which user interaction triggers the PAI event and control returns to the application server, user interaction on controls is not automatically passed back to the application server. If you want an event to be passed back to the application server, you must register it in your program using the special method SET_REGISTERED_EVENTS. For a list of the events that you can register for each control, refer to its wrapper class in the Class Builder. You can register two kinds of event handling [Ext.] using SET_REGISTERED_EVENTS:

System Events (Default)

The event is passed to the application server, but **does not** trigger the PAI. If you have registered an **event handler method** in your ABAP program for the event (using the SET HANDLER statement), this method is executed on the application server.

Within the event handler method, you can use the static method SET_NEW_OK_CODE of the global class CL_GUI_CFW to set a function code and trigger the PAI event **yourself**. After the PAI has been processed, the PBO event of the next screen is triggered.

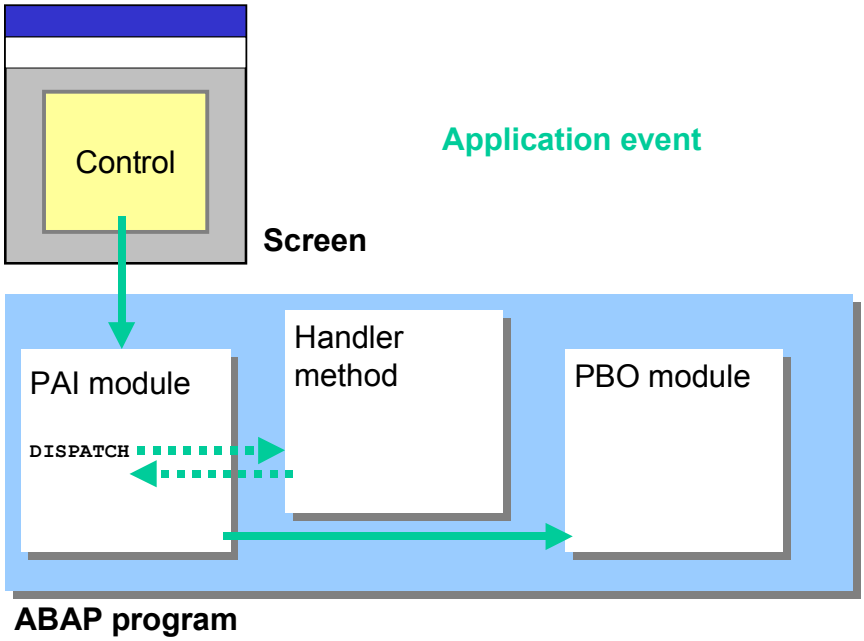
The advantage of using this technique is that the event handler method is executed automatically and there are no conflicts with the automatic input checks associated with the screen. The disadvantage is that the contents of the screen fields are not transported to the program, which means that obsolete values could appear on the next screen. You can work around this by using the SET_NEW_OK_CODE method to trigger field transport and the PAI event **after** the event handler has finished.



Application Events

The event is passed to the application server, and triggers the PAI. The function code that you pass contains an internal identifier. You **do not have to evaluate this** in your ABAP program. Instead, if you want to handle the event, you must include a method call in a PAI dialog module for the static method DISPATCH of the global class CL_GUI_CFW. If you have defined an **event handler method** in your ABAP program for the event (using the SET HANDLER statement), the DISPATCH method calls it. After the event handler has been processed, control returns to the PAI event after the DISPATCH statement and PAI processing continues.

The advantage of this is that you can specify yourself the point at which the event is handled, and the contents of the screen fields are transported to the application server beforehand. The disadvantage is that this kind of event handling can lead to conflicts with the automatic input checks on the screen, causing events to be lost.



Further Information

For further information about controls, and in particular, help on troubleshooting and optimizing synchronization, refer to BC Controls Tutorial [Ext.] and BC SAP Control Framework [Ext.].

Example

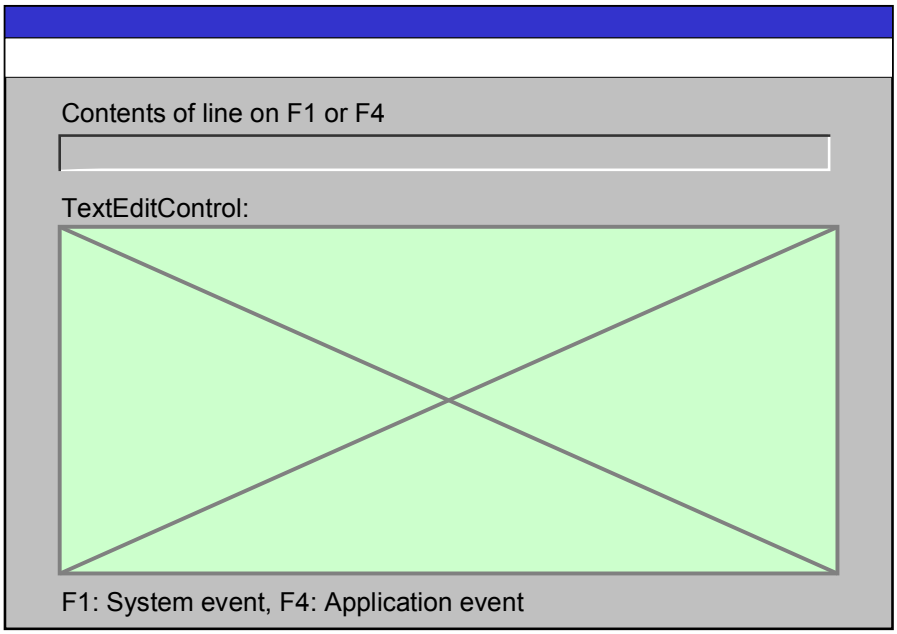
The following example shows the difference between system and application events.

```
REPORT demo_custom_control .

* Declarations
*****************************************************

CLASS event_handler DEFINITION.
  PUBLIC SECTION.
    METHODS: handle_f1 FOR EVENT f1 OF cl_gui_textedit
                       IMPORTING sender,
             handle_f4 FOR EVENT f4 OF cl_gui_textedit
                       IMPORTING sender.
ENDCLASS.

DATA: ok_code LIKE sy-ucomm,
      save_ok LIKE sy-ucomm.

DATA: init,
      container TYPE REF TO cl_gui_custom_container,
      editor    TYPE REF TO cl_gui_textedit.

DATA: event_tab TYPE cntl_simple_events,
      event     TYPE cntl_simple_event.
```

```
DATA: line(256),
      text_tab LIKE STANDARD TABLE OF line,
      field LIKE line.

DATA handle TYPE REF TO event_handler.

* Reporting Events
*****************************************************

START-OF-SELECTION.
  line = 'First line in TextEditControl'.
  APPEND line TO text_tab.
  line = '-------------------------------------------------'.
  APPEND line TO text_tab.
  line = '...'.
  APPEND line TO text_tab.
  CALL SCREEN 100.

* Dialog Modules
*****************************************************

MODULE status_0100 OUTPUT.
  SET PF-STATUS 'SCREEN_100'.
  IF init is initial.
    init = 'X'.
    CREATE OBJECT:
           container EXPORTING container_name = 'TEXTEDIT',
           editor    EXPORTING parent = container,
           handle.
    event-eventid = cl_gui_textedit=>event_f1.
    event-appl_event = ' '.                      "system event
    APPEND event TO event_tab.
    event-eventid = cl_gui_textedit=>event_f4.
    event-appl_event = 'X'.                      "application
event
    APPEND event TO event_tab.
    CALL METHOD: editor->set_registered_events
                 EXPORTING events = event_tab.
    SET HANDLER handle->handle_f1
                handle->handle_f4 FOR editor.
  ENDIF.
  CALL METHOD editor->set_text_as_stream
              EXPORTING text = text_tab.
ENDMODULE.

MODULE cancel INPUT.
  LEAVE PROGRAM.
ENDMODULE.

MODULE user_command_0100 INPUT.
  save_ok = ok_code.
  CLEAR ok_code.
  CASE save_ok.
    WHEN 'INSERT'.
      CALL METHOD editor->get_text_as_stream
                  IMPORTING text = text_tab.
    WHEN 'F1'.
```

```
      MESSAGE i888(sabapdocu) WITH text-001.
    WHEN OTHERS.
      MESSAGE i888(sabapdocu) WITH text-002.
      CALL METHOD cl_gui_cfw=>dispatch.
  ENDCASE.
  SET SCREEN 100.
ENDMODULE.

* Class Implementations
*****************************************

CLASS event_handler IMPLEMENTATION.
  METHOD handle_f1.
    DATA row TYPE i.
    MESSAGE i888(sabapdocu) WITH text-003.
    CALL METHOD sender->get_selection_pos
         IMPORTING from_line = row.
    CALL METHOD sender->get_line_text
         EXPORTING line_number = row
         IMPORTING text = field.
    CALL METHOD cl_gui_cfw=>set_new_ok_code
         EXPORTING new_code = 'F1'.
    CALL METHOD cl_gui_cfw=>flush.
  ENDMETHOD.

  METHOD handle_f4.
    DATA row TYPE i.
    MESSAGE i888(sabapdocu) WITH text-004.
    CALL METHOD sender->get_selection_pos
         IMPORTING from_line = row.
    CALL METHOD sender->get_line_text
         EXPORTING line_number = row
         IMPORTING text = field.
    CALL METHOD cl_gui_cfw=>flush.
  ENDMETHOD.
ENDCLASS.
```

The layout of screen 100 is:



The screen contains an output field FIELD and a custom control called TEXTEDIT.

It has the following flow logic:

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0100.

PROCESS AFTER INPUT.
  MODULE CANCEL AT EXIT-COMMAND.
  MODULE USER_COMMAND_0100.
```

The GUI status SCREEN_100 has the functions BACK, EXIT, and CANCEL (all with type E) and the function INSERT (normal function).

There is a local class EVENT_HANDLER defined in the program. It contains event handler methods for the F1 and F4 events of global class CL_GUI_TEXTEDIT. When you run the program, the classes CL_GUI_CUSTOM_CONTROL, CL_GUI_TEXTEDIT, and EVENT_HANDLER are instantiated in the PBO of screen 100.

The container control is linked to the custom control on the screen, and the instance of the SAP Textedit is linked to the container. The F1 and F4 events of the SAP Textedit are registered using the SET_REGISTERED_EVENTS method to ensure that they are passed to the application server when they occur. F1 is defined as a system event, F4 as an application event. The event handler methods of the HANDLE instance of the class EVENT_HANDLER are registered as handlers for the events.

Before screen 100 is displayed, the program fills the SAP Textedit Control with the contents of table TEXT_TAB. The user can edit the text while the screen is displayed. If the user chooses INSERT, the PAI event is triggered and the current text from the SAP Textedit is copied into table TEXT_TAB.

If the user chooses F1 on the control, the HANDLE_F1 method is executed. This assigns the contents of the line to the field FIELD: The method SET_NEW_OK_CODE triggers the PAI event. It is this that ensures that the PBO is processed, and the contents of FIELD are sent to the screen.

If the user chooses F4 on the control, the PAI event is triggered. The DISPATCH method is called, and this triggers the method HANDLE_F4. This assigns the contents of the line to the field FIELD: Since the PAI processing continues after the event, the PBO event follows, and the field contents are transferred to the screen.

The contents of the control are not passed to the internal table TEXT_TAB either after F1 or after F4. The contents of the control are therefore overwritten in the PBO event with the previous contents of TEXT_TAB.

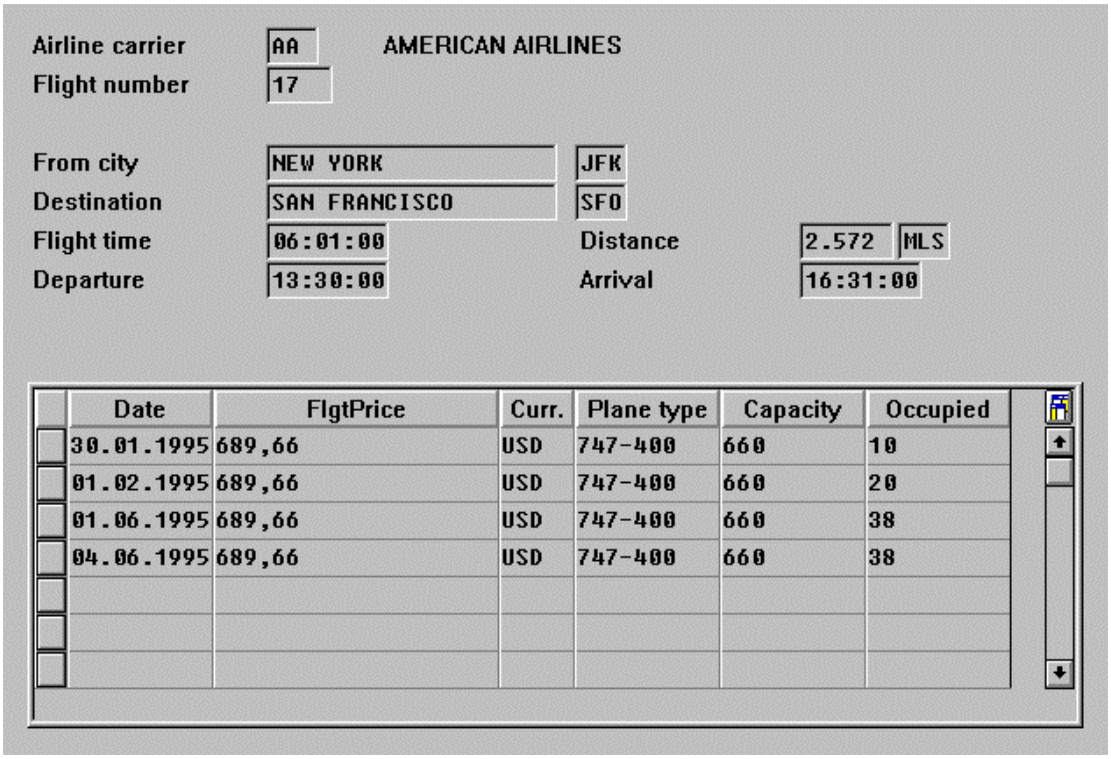
Table Controls

Table Controls in der Ablauflogik [Page 671]

Table Controls: Beispiel mit Blättern [Page 676]

Table Controls: Beispiel mit Modifikationen [Page 678]

ABAP offers two mechanisms for displaying and using table data in a screen. These mechanisms are *table controls* and *step loops*. Table controls and step loops are types of screen tables you can add to a screen in the Screen Painter. For example, the following screen contains a table control at the bottom:



This chapter describes how to program the screen flow logic and ABAP code that let you use screen tables. For information on using screen tables, see:

Introduction

Using the LOOP Statement [Page 670]

Using Table Controls

Using Step Loops [Page 679]

Example Transaction: Table Controls [Page 672]

Using the LOOP Statement

Screen Painter [Ext.]

The LOOP...ENDLOOP screen command lets you perform looping operations in the flow logic. You can use this statement to loop through both table controls and step loops. Between a LOOP and its ENDLOOP, you can use the FIELD, MODULE, SELECT, VALUES and CHAIN screen keywords. Most often, you use the MODULE statement to call an ABAP module.

You *must* code a LOOP statement in both the PBO and PAI events for each table in your screen. This is because the LOOP statement causes the screen fields to be copied back and forth between the ABAP program and the screen field. For this reason, at least an empty LOOP...ENDLOOP must be there.

There are two important forms of the LOOP statement:

- **LOOP.**

  This statement loops through screen table rows, transferring the data in each block to and from the corresponding ABAP fields in your program. The screen table fields may be declared in ABAP as anything (database table, structure or individual fields) except as internal table fields.

  With step loops, if you are implementing your own scrolling (for example, with `F21-F24`) you must use this statement.

- **LOOP AT <internal table>.**

  This statement loops through an internal table and the screen table rows in parallel. The screen table fields often are, but need not be, declared as internal table fields.

  With this LOOP, step loop displays appears with scroll bars. This scrolling is handled automatically by the system.

For more details on the different LOOP statements, see:

Looping Directly Through a Screen Table

Looping Through an Internal Table

Looping Through an Internal Table

Systemfelder [Page 1444]

Datentransports [Page 565]

If you write the statement

```
LOOP AT <internal table>.
```

loops through an internal table and a screen table in parallel. In particular, LOOP AT loops through the portion of the internal table that is currently visible in the screen. You can use this form of the LOOP statement for both table controls and step loops.

The complete syntax for this form of the LOOP statement is:

```
LOOP AT <internal table> CURSOR <scroll-var>
        [WITH CONTROL <table-control>]
        [FROM <line1>]  [TO <line2>].
...<actions>...
ENDLOOP.
```

This form of LOOP loops through the internal table, performing <actions> for each row. For each internal table row, the system transfers the relevant program fields to or from the corresponding screen table row.

When using step loops, omit the CURSOR parameter in the PAI event. The FROM and TO parameters are only possible with step loops. (For further information, refer to Using Step Loops [Page 679].) The WITH CONTROL parameter is only for use with table controls.

For further information, refer to the following sections:

How the System Transfers Data Values

Scrolling and the Scroll Variables

Example Transaction: Table Controls

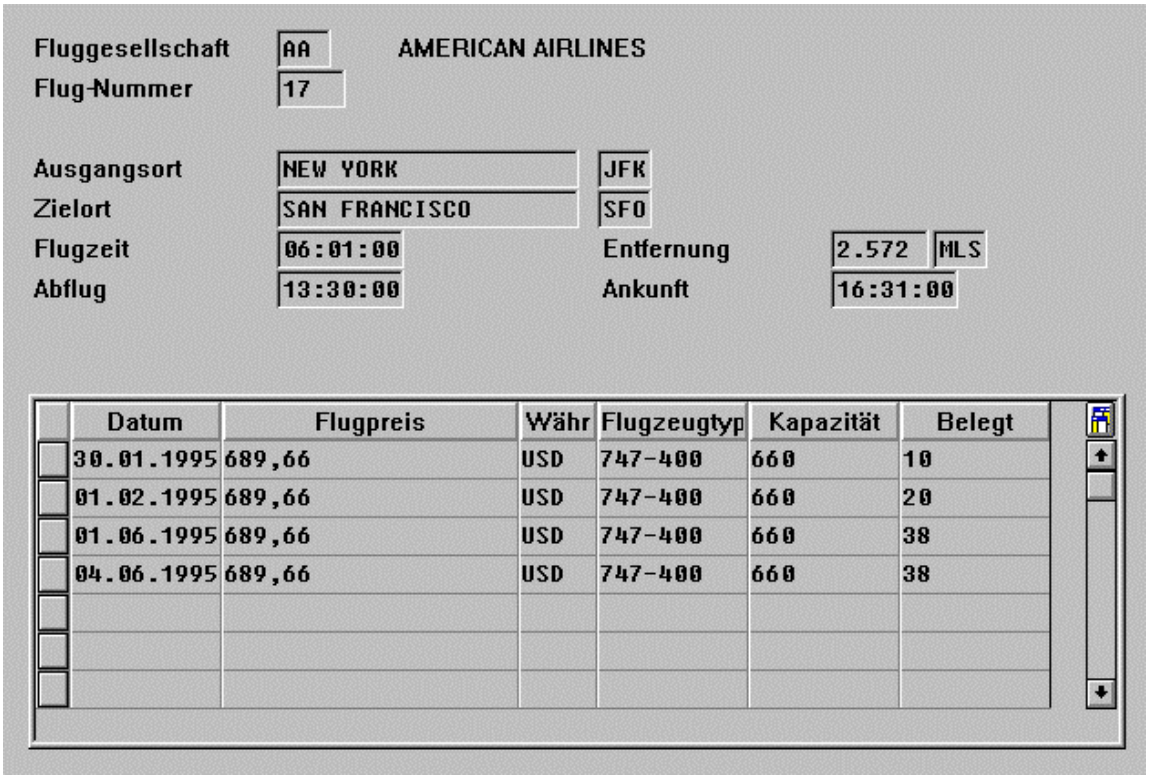
SCREEN [Page 612]

SCREEN [Page 612]

Cursorposition festlegen [Page 631]

Cursorposition bestimmen [Page 557]

Transaction TZ60 is an example of programming with table controls. It displays flight information on two screens. On the first, the user specifies a flight connection and requests the detailed information. On the second (screen 200), the transaction displays all flights scheduled for the connection:



There are two important things to notice in this program:

- how table data is fetched and passed around

  The fields in the table control are declared as database fields (table SFLIGHTS) in the ABAP program. To store several SFLIGHTS rows at a time, the program uses the internal table INT_FLIGHTS.

1. During PAI for screen 100, the program selects all flights for the given connection and stores them in internal table INT_FLIGHTS.
2. AT PBO for screen 200, the LOOP statement loops through INT_FLIGHTS, calling the module DISPLAY_FLIGHTS for each row. DISPLAY_FLIGHTS transfers an INT_FLIGHTS row to the SFLIGHTS work area. The SFLIGHTS fields match the screen table names, so at the end of each loop pass, the system transfers the ABAP values to the screen table fields.

- use of table control fields to control scrolling

  Table controls contain information for managing scrolling. This includes fields telling how many table rows are filled and which table row should be first in the screen display into the table. For both step loops and table controls, the system manages scrolling with the scroll bars automatically. This includes scrolling the actual screen display, as well as resetting scrolling variables. However, the example program offers scrolling with the `F21-F24` keys. In this case, the program must implement these functions explicitly.

1. During PAI for screen 100, the program initializes the “first-table-row” variable (field TOP_LINE) to 1.
2. At PAI for screen 200, the program resets scrolling variables, especially the TOP_LINE field, if the user has scrolled with the `F21-F24` function keys.

Screen Flow Logic

The screen flow logic (PAI only) for screen 200 in transaction TZ60 looks as follows.

```
*---------------------------------------------------------------------*
*   Screen 200: Flow Logic                 *
*&---------------------------------------------------------------------*
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0200.
    LOOP AT INT_FLIGHTS WITH CONTROL FLIGHTS
                    CURSOR FLIGHTS-CURRENT_LINE.
      MODULE DISPLAY_FLIGHTS.
  ENDLOOP.
*
*
PROCESS AFTER INPUT.
  LOOP AT INT_FLIGHTS.
    MODULE SET_LINE_COUNT.
  ENDLOOP.
  MODULE USER_COMMAND_0200.
```

ABAP Code

The following section of the TOP-module code shows the definitions relevant to the following code.

```
*---------------------------------------------------------------------*
*   INCLUDE MTD40TOP                                                  *
*---------------------------------------------------------------------*
PROGRAM  SAPMTZ60 MESSAGE-ID A&.
  TABLES: SFLIGHT.
  <....Other declarations....>
  DATA INT_FLIGHTS LIKE SFLIGHT OCCURS 1 WITH HEADER LINE.

  DATA: LINE_COUNT TYPE I.

  CONTROLS: FLIGHTS TYPE TABLEVIEW USING SCREEN 200.
```

Example Transaction: Table Controls

The following PBO module transfers internal table fields to the proper screen table fields at PBO.

```
*&---------------------------------------------------------------------*
*&      Module  DISPLAY_FLIGHTS  OUTPUT
*&---------------------------------------------------------------------*
MODULE DISPLAY_FLIGHTS OUTPUT.
  SFLIGHT-FLDATE    = INT_FLIGHTS-FLDATE.
  SFLIGHT-PRICE     = INT_FLIGHTS-PRICE.
  SFLIGHT-CURRENCY  = INT_FLIGHTS-CURRENCY.
  SFLIGHT-PLANETYPE = INT_FLIGHTS-PLANETYPE.
  SFLIGHT-SEATSMAX  = INT_FLIGHTS-SEATSMAX.
  SFLIGHT-SEATSOCC  = INT_FLIGHTS-SEATSOCC.
ENDMODULE.
```

At PAI, the program must loop again, to transfer screen table fields back to the program. During this looping, the program can use SY_LOOPC to find out how many rows were transferred. SY_LOOPC is a system variable telling the total number of rows currently showing in the display.

```
*&---------------------------------------------------------------------*
*&      Module  SET_LINE_COUNT  INPUT
*&---------------------------------------------------------------------*
MODULE SET_LINE_COUNT INPUT.
  LINE-COUNT = SY-LOOPC.
ENDMODULE
```

Transaction TZ60 lets the user press function keys (`F21-F24`) to scroll the display. The system handles scrolling with the scroll bars automatically, but not scrolling with function keys. So PAI for screen 200 must implement function-key scrolling explicitly:

```
MODULE USER_COMMAND_0200 INPUT.
  CASE OK_CODE.
WHEN 'CANC'...
WHEN 'EXIT'...
WHEN 'BACK'...
WHEN 'NEW'...
    WHEN 'P--'.
      CLEAR OK_CODE.
      PERFORM PAGING USING 'P--'.
    WHEN 'P-'.
      CLEAR OK_CODE.
      PERFORM PAGING USING 'P-'.
    WHEN 'P+'.
      CLEAR OK_CODE.
      PERFORM PAGING USING 'P+'.
    WHEN 'P++'.
      CLEAR OK_CODE.
      PERFORM PAGING USING 'P++'.
  ENDCASE.
ENDMODULE.
```

The PAGING routine causes the system to scroll the display by re-setting the table control field TOP_LINE to a new value. The TOP_LINE field tells the LOOP statement where to start looping at PBO.

```
*&---------------------------------------------------------------------*
*&      Form  PAGING
*&---------------------------------------------------------------------*
FORM PAGING USING CODE.
  DATA: I TYPE I.
                        J TYPE I.

          CASE CODE.
    WHEN 'P--'. FLIGHTS-TOP_LINE = 1.
    WHEN 'P-'.  FLIGHTS-TOP_LINE = FLIGHTS-TOP_LINE - LINE_COUNT.
             IF FLIGHTS-TOP_LINE LE 0.
               FLIGHTS-TOP_LINE = 1. ENDIF.
    WHEN 'P+'.  I = FLIGHTS-TOP_LINE + LINE_COUNT.
             J = FLIGHTS-LINES - LINE_COUNT + 1.
             IF J LE 0.
               J = 1. ENDIF.
             IF I LE J.
                                       FLIGHTS-TOP_LINE = I.
                                     ELSE. FLIGHTS-TOP_LINE = J.
                                     ENDIF.
    WHEN 'P++'. FLIGHTS-TOP_LINE = FLIGHTS-LINES - LINE_COUNT + 1.
             IF FLIGHTS-TOP_LINE LE 0.
               FLIGHTS-TOP_LINE = 1. ENDIF.
  ENDCASE.
ENDFORM.
```

Looping Directly Through a Screen Table

Use the simple form of the LOOP statement

```
LOOP.
...<actions>...
ENDLOOP.
```

to loop through the currently displayed rows of a screen table. If you are using a table control, include the additional WITH CONTROL parameter:

```
LOOP WITH CONTROL <table control>.
...<actions>...
ENDLOOP.
```

This simple LOOP is the most general form of the LOOP statement. If you use this LOOP, you can declare the screen table fields as any type (internal table, database table, structure or individual fields). The simple LOOP copies the screen table fields back and forth to the relevant ABAP fields. If you want to manipulate the screen values in a different structure, you must explicitly move them to where you want them.

Each pass through the loop places the next table row in the ABAP fields, and executes the LOOP <actions> (usually ABAP module calls) for it. In a PBO event, the LOOP statement causes loop fields in the program to be copied row by row to the screen. In a PAI event, the fields are copied row by row to the relevant program fields.

In an ABAP module, use the system variable SY-STEPL to find out the index of the screen table row that is currently being processed. The system sets this variable each time through the loop. SY-STEPL always has values from 1 to the number of rows currently displayed. This is useful when you are transferring field values back and forth between a screen table and an internal table. You can declare a table-offset variable in your program (often called BASE, and usually initialized with SY-LOOPC) and use it with SY-STEPL to get the internal table row that corresponds to the current screen table row.

(In the example below, the screen fields are declared as an internal table. The program reads or modifies the internal table to get table fields passed back and forth to the screen.)

```
***SCREEN FLOW LOGIC***
PROCESS BEFORE OUTPUT.
  LOOP.
    MODULE READ_INTTAB.
  ENDLOOP.

PROCESS AFTER INPUT.
  LOOP.
    MODULE MODIFY_INTTAB.
  ENDLOOP.


***ABAP MODULES***
MODULE READ_INTTAB.
  IND = BASE + SY-STEPL - 1.
  READ TABLE INTAB INDEX IND.
ENDMODULE.
```

```
MODULE MODIFY_INTTAB.
  IND = BASE + SY-STEPL - 1.
  MODIFY INTTAB INDEX IND.
ENDMODULE.
```

Remember that the system variable SY-STEPL is only meaningful within LOOP...ENDLOOP processing. Outside the loop, it has no valid value.

How the System Transfers Data Values

With the LOOP AT <internal table> statement, the screen table need not be declared as the internal table. Screen tables can also be database tables, structures or other program fields. If you don't define the screen table as an internal table, you must make sure that the correct fields are moved to and from the internal table header as needed during looping.

Note also that the fields you use in the screen table may be only a subset of the corresponding dictionary or internal table fields. The system transfers fields as needed in the screen table. Processing (in detail) is as follows:

- **PBO processing**

1. A row of the internal table is placed in the header area.
2. All loop statements are executed for that row.

   Loop statements are most often calls to ABAP modules. Where necessary, these modules should move the internal table fields to the relevant program fields (dictionary table or other fields).

   Example transaction TZ60 does this in the DISPLAY_FLIGHTS routine:

```
MODULE DISPLAY_FLIGHTS OUTPUT.
  SFLIGHT-FLDATE    = INT_FLIGHTS-FLDATE.
  SFLIGHT-PRICE     = INT_FLIGHTS-PRICE.
  SFLIGHT-CURRENCY  = INT_FLIGHTS-CURRENCY.
  SFLIGHT-PLANETYPE = INT_FLIGHTS-PLANETYPE.
  SFLIGHT-SEATSMAX  = INT_FLIGHTS-SEATSMAX.
  SFLIGHT-SEATSOCC  = INT_FLIGHTS-SEATSOCC.
ENDMODULE.
```

3. The system transfers values from the program fields to the screen fields with the same names.

- **PAI processing**

4. A row of the internal table is placed to the internal table header area.
5. All screen table fields are transferred to the ABAP program fields with the same names.
6. All loop <actions> are executed for the current internal table row.

   Again, <actions> is usually a call to an ABAP module. Where necessary, this module should first move the program field values to the header area for the internal table. This step overwrites data values from the internal table with those from the screen. Step 1 is necessary for cases where the screen table fields are only a subset of the fields defined for the internal table. If you want to update the internal table with new screen values, you must have the original table row as a basis in the header area.

   Remember: If you want to update the internal table with the contents of the header area, use the MODIFY statement in the ABAP module. The system does not automatically do it for you.

Note that there is a screen language MODIFY statement and an ABAP MODIFY statement. Do not use the screen language MODIFY to update internal tables.

Using Step Loops

Screen Painter [Ext.]

Screen Painter [Ext.]

Table Controls in der Ablauflogik [Page 671]

CXTAB_CONTROL [Page 672]

If you have a step loop in your screen, you can place the cursor on a particular element in the step loop block. Use the LINE parameter, entering the line of the loop block where you want to place the cursor:

SET CURSOR FIELD <fieldname> LINE <line>.

If you want, you can use the OFFSET and LINE parameters together.

A step loop is a repeated series of field-blocks in a screen. Each block can contain one or more fields, and can extend over more than one line on the screen.

Step loops as structures in a screen do not have individual names. The screen can contain more than one step loop, but if so, you must program the LOOP...ENDLOOPs in the flow logic accordingly. The ordering of the LOOP...ENDLOOPs must exactly parallel the order of the step loops in the screen. The ordering tells the system which loop processing to apply to which loop. Step loops in a screen are ordered primarily by screen row, and secondarily by screen column.

Transaction TZ61 (development class SDWA) implements a step loop version of the table you saw in transaction TZ60. Screen 200 for TZ61, shows the following step loop:

Using Step Loops

See the TZ61 code for a demonstration of how to use step loops.

Static and Dynamic Step Loops

Step loops fall into two classes: static and dynamic. Static step loops have a fixed size that cannot be changed at runtime. Dynamic step loops are variable in size. If the user re-sizes the window, the system automatically increases or decreases the number of step loop blocks displayed. In any given screen, you can define any number of static step loops, but only a single dynamic one.

You specify the class for a step loop in the Screen Painter. Each loop in a screen has the attributes *Looptype* (*fixed*=static, *variable*=dynamic) and *Loopcount*. If a loop is fixed, the *Loopcount* tells the number of loop-blocks displayed for the loop. This number can never change.

Programming with static and dynamic step loops is essentially the same. You can use both the LOOP and LOOP AT statements for both types.

Looping in a Step Loop

When you use LOOP AT <internal-table> with a step loop, the system automatically displays the step loop with vertical scroll bars. The scroll bars, and the updated (scrolled) table display, are managed by the system.

Use the following additional parameters if desired:

- FROM <line1> and TO <line2>

  These parameters limit the parts of the internal table that can be displayed or processed in the step loop. If you use one or both of these, declare <line1> and <line2> in ABAP with LIKE SY-TABIX. If you don’t use them, the system simply uses the beginning and/or end of the internal table as the limit.

- CURSOR <scroll-var>

  The CURSOR parameter tells which internal table row should be the first in the screen display. <Scroll-var> is a local program variable that can be set either by your program or automatically by the system. The value of <scroll-var> is absolute with respect to the internal table (that is, not relative to the FROM or TO values).

The CURSOR <scroll-var> parameter is required in the PBO event to tell the system where to start displaying.

Selection Screens

Subscreens [Page 754]

Subscreens und TabStrips für Selektionsbilder [Page 753]

Selection screens are one of the three types of screen in the R/3 System, along with dialog screens and lists. You use them whenever you want the user to enter either a single value for a field or fields, or to enter selection criteria.

Function

ABAP programs use screens to obtain input from users. The most general type of screen is a dialog screen, which you create using the ABAP Workbench [Ext.] tools Screen Painter and Menu Painter These tools allow you to create screens for data input and output. However, each of these screens requires its own flow logic.

You often use screens purely for **data input .** In these cases, you can use a selection screen. Selection screens provide a standardized user interface in the R/3 System. Users can enter both single values and complex selections. Input parameters are primarily used to control the program flow, while users can enter selection criteria to restrict the amount of data read from the database. You can create and save predefined sets of input values in the ABAP Editor for any selection screen. These are called variants [Ext.]. Texts on the selection screen are stored as language-specific selection texts [Ext.] in the program text elements. If you start an executable report using the SUBMIT [Page 1018] statement, the input fields of the selection screen also serve as a data interface.

Defining and Calling Selection Screens

You define selection screens using ABAP statements in a program. Simple statements allow you to create input fields, checkboxes, and radio buttons, and design the screen layout. If you want to create a screen exclusively for data input, you do not need to create it using the normal dialog programming tools. When you create a selection screen, the system automatically assumes the tasks of the Screen Painter and Menu Painter.

The rules for calling and defining selection screens in ABAP programs depend on the program type:

- Executable program (type 1) without logical database

  You can use a single **standard selection screen** and as many **user-defined selection screens** as you wish. The standard selection screen is called automatically when you start the program. User-defined selection screens, on the other hand, are called using the CALL SELECTION-SCREEN statement in a program. The standard selection screen always has the screen number 1000. User-defined selection screens can have any screen number except 1000.

- Executable program (type 1) with logical database

  The **standard selection screen** for an executable program linked to a logical database is made up of the logical database selections and the program selections.

- Module pools (type M) and function modules (type F)

  You can only use user-defined selection screens in module pools and function modules. These can have any number apart from 1000. You can only call a selection screen from

a function module using the CALL SELECTION-SCREEN statement. You can use a user-defined selection screen in a module pool as the initial screen of a transaction.

Selection Screens and Logical Databases [Page 683]

Defining Selection Screens [Page 686]

Calling Selection Screens [Page 724]

Processing Selection Screens [Page 739]

Using Selection Criteria in Programs [Page 764]

Selection Screens and Logical Databases

The selection parts of logical databases [Page 1163] contain statements for defining selection screens. The standard selection screen of an executable program (report) that is linked to a logical database automatically contains the corresponding input fields. A particular feature that you can find on the selection screen of logical databases are the dynamic selections. Dynamic selections allow users to enter values for fields that are not part of the main selection screen of the logical database.

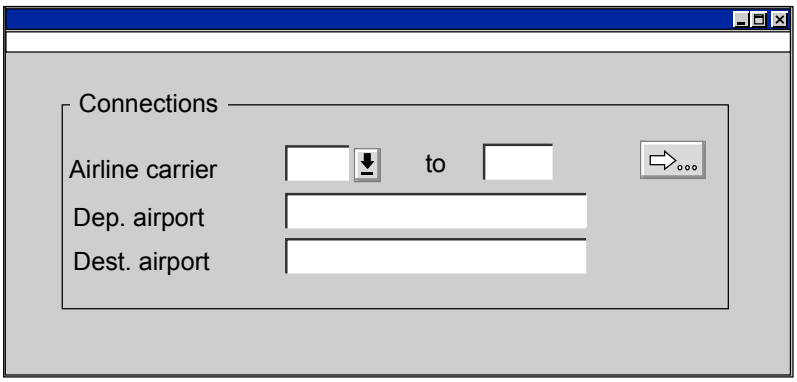
Static Selections in Logical Databases

The static selections of the logical database are defined in its selection part with the same ABAP statements as the program selections. The input fields for the logical database selections that actually appear on the screen depend on the nodes of the logical database that are specified in the program using the TABLES or the NODES statement.

The following program is linked to logical database F1S.

```
REPORT DEMO.
NODES SPFLI.
```

After DEMO has been started, the following selection screen is displayed:

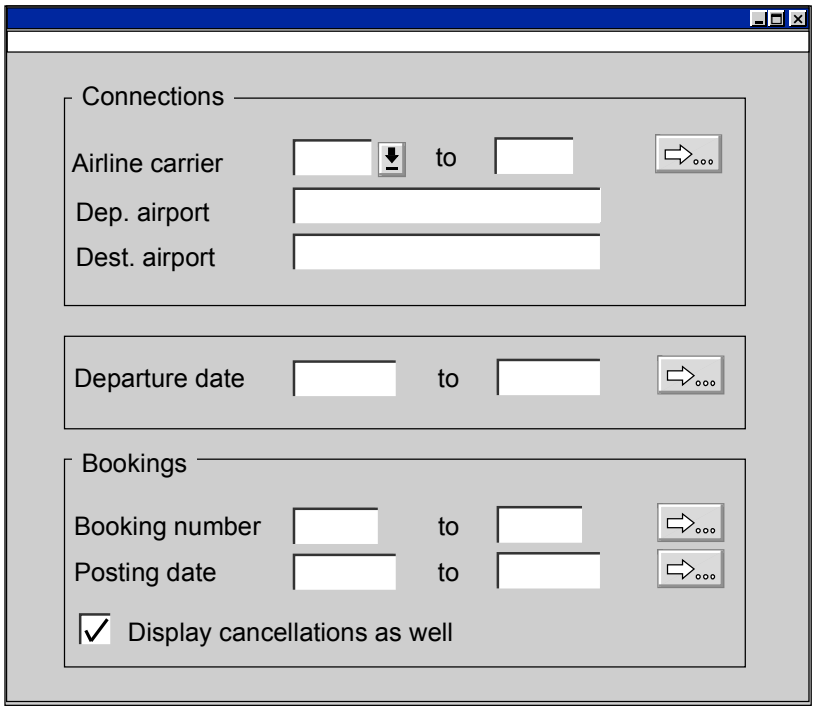


It contains input fields for selection criteria and parameters for the fields of database table SPFLI. The statements that define this screen (SELECT-OPTIONS and PARAMETERS) are contained in the selection part of the logical database.

Assume another program:

```
REPORT DEMO.
NODES SBOOK.
```

After DEMO has been started, the following selection screen is displayed:



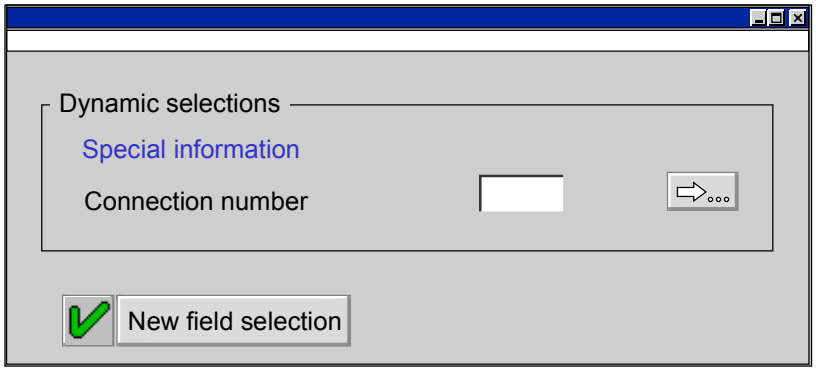
It not only contains the input fields of the selection criteria for database table SBOOK, but also the criteria for tables SPFLI and SFLIGHT.

The logical database uses its own selections to restrict the amount of data read from the database. You should make extensive use of the selection criteria provided by the logical database and only define program-specific selections if the selection criteria of the logical database are insufficient for your requirements. On the selection screen, the input fields for the program-specific selections appear below the selections for the logical database.

Dynamic Selections of Logical Databases

Logical databases allow the user to specify dynamic selections that are not defined in the selection part of the logical database. The user accesses these selections by clicking *Dynamic selections* in the application toolbar of the selection screen. The system displays a new selection screen or a screen where the user can select the database fields that he or she wants to specify selection criteria for.

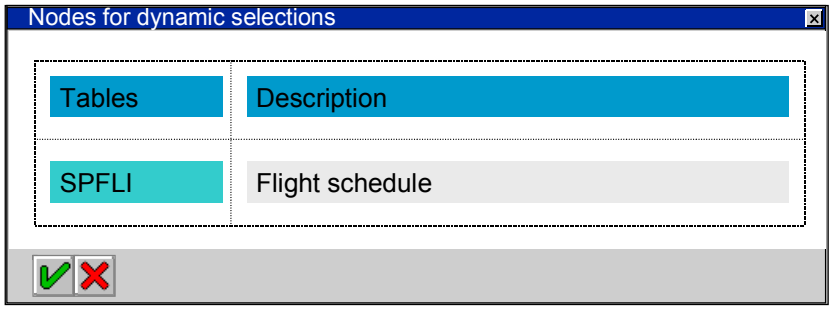
For logical database F1S, the screen for dynamic selections might look as follows:



Dynamic selections reduce the database accesses of logical database programs by means of dynamic statements. The possibility of dynamic selections must be coded in the logical database program (see Dynamic Selections in Database Programs [Page 1208]).

If a logical database allows dynamic selections (*Dynamic selections* appears in the application toolbar), you can use the ABAP Workbench to define for which fields the user can define dynamic selections (a selection view). To determine which database tables provide dynamic selections, choose *Tools* → *ABAP Workbench* → *Development* → *Programming environment*→ *Logical databases* → *Extras* → *Dynamic selections*. On the next screen, the system displays a list of the names of these database tables.

For logical database F1S, dynamic selections are only possible for database table SPFLI:



Defining Selection Screens

There are three ABAP statements for defining selection screens:

- PARAMETERS [Page 689] for single fields
- SELECT-OPTIONS [Page 703] for complex selections
- SELECTION-SCREEN [Page 718] for formatting the selection screen and defining user-specific selection screens

These ABAP statements are included in the declaration part of an ABAP program. When defining selection screens, you must distinguish between standard selection screens for executable programs (reports) and user-defined selection screens for all types of program.

Standard selection screens

The standard selection screen of executable programs is predefined and has screen number 1000. All PARAMETERS, SELECT-OPTIONS and SELECTION-SCREEN statements that do not occur in the definition of a user-defined selection screen in executable programs (reports), that is, that are **not** included between the statements

```
SELECTION-SCREEN BEGIN OF ...
...
SELECTION-SCREEN END OF ...
```

define the input fields and the formatting of the standard selection screen. For the sake of clarity, you should group together all statements that make up the standard selection screen before defining additional selection screens. You can define input fields of a standard selection screen only in executable programs (reports). In all other programs, the PARAMETERS, SELECT-OPTIONS and SELECTION-SCREEN statements must be included in the above statements.

User-defined selection screens

The two statements:

```
SELECTION-SCREEN BEGIN OF SCREEN <numb> [TITLE <title>] [AS WINDOW].
  ...
SELECTION-SCREEN END OF SCREEN <numb>.
```

define a user-defined selection screen with screen number <numb>. All PARAMETERS, SELECT-OPTIONS and SELECTION-SCREEN statements that occur **between** these two statements define the input fields and the formatting of this selection screen. Screen number <numb> can be any four-digit number apart from 1000 which is the number of the standard selection screen. You must also ensure that you do not accidentally assign a number to a selection screen which is already in use for another screen of the program.

The TITLE <title> addition allows you to define a title for a user-defined selection screen. <Title> can either be a static text symbol or a dynamic character field. If you use a character field, you must not define it using the DATA statement - the system generates it automatically. The character field can be filled during the INITIALIZATION [Page 954] event. The title of standard selection screens is always the name of the executable program.

You can use the AS WINDOW addition to call a user-defined selection screen as a modal dialog box. The window is defined when the screen is called. When you use the AS WINDOW addition, warnings and error messages associated with the selection screen are also displayed as modal dialog boxes, and not in the status bar of the selection screen.

If you define more than one selection screen in a program, you can re-use elements of one selection screen in another using the following statement:

```
SELECTION SCREEN INCLUDE          BLOCKS <block>
                        |      PARAMETERS <p>
                        | SELECT-OPTIONS <selcrit>
                        |         COMMENT <comm>
                        |    PUSH-BUTTON.<push>.
```

You can specify any of the following elements that have already been declared in another selection screen:

- Blocks [Page 723] with name <block>
- Parameters [Page 690] with name <p>
- Selection criteria [Page 707] with name <selcrit>
- Comments [Page 719] with name <comm>
- Pushbuttons [Page 733] with name <push>

GUI Status of Selection Screens

The GUI status of a selection screen is generated by the system. The SET PF-STATUS statement in the PBO event of the selection screen has no effect on the standard GUI status. If you want to use your own GUI status for a selection screen or deactivate functions in the standard GUI status, you can use one of the following function modules in the PBO event of the selection screen:

- RS_SET_SELSCREEN_STATUS

  Sets another GUI status defined in the same ABAP program, or deactivates functions of the standard GUI status.

- RS_EXTERNAL_SELSCREEN_STATUS

  Sets a GUI status defined in an external function group. You must use the SET PF-STATUS statement to set the status in a special function module in this function group. You must pass the name of the function module that sets the status as a parameter to the function module RS_EXTERNAL_SELSCREEN_STATUS.

Example

```
REPORT SELSCREENDEF.

...

PARAMETERS PAR1 ... .
SELECT-OPTIONS SEL1 FOR ... .
...

SELECTION-SCREEN BEGIN OF SCREEN 500 AS WINDOW.
  PARAMETERS PAR2 ... .
  SELECT-OPTIONS SEL2 FOR ... .
  ...
SELECTION-SCREEN END OF SCREEN 500.
```

Defining Selection Screens

```
SELECTION-SCREEN BEGIN OF SCREEN 600 TITLE TEXT-100.
  SELECTION-SCREEN INCLUDE: PARAMETERS PAR1,
                            SELECT-OPTIONS SEL1.
  PARAMETERS PAR3 ... .
  SELECT-OPTIONS SEL3 ... .
  ...
SELECTION-SCREEN END OF SCREEN 600.
```

Three selection screens - the standard selection screen and two user-defined selection screens - are defined. The program must have type 1 in order for a standard selection screen to be generated. Selection screen 500 is defined to be called as a modal dialog box. Selection screen 600 contains text symbol 100 as its title, and uses elements PAR1 and SEL1 from the standard selection screen.

Defining Input Fields for Single Values

Sichtbare Länge verkleinern [Page 695]

Suchhilfe für Parameter [Page 697]

If you want to enable the user to enter values for single fields on the selection screen, you must define specific variables [Page 123] (parameters) in the declaration part using the PARAMETERS statement. Each parameter declared with the PARAMETERS statement appears as an input field on the relevant selection screen.

Parameters are used for simple queries of single values. For example, you can use parameters to control the program flow. However, if you want to restrict database accesses or perform complex selections, you should use selection criteria [Page 703].

You can use the PARAMETERS statement for both standard and user-defined selection screens. The variants of the PARAMETERS statement which are important for program-specific selections are explained in the following sections:

There are special variants of the PARAMETERS statement that you can use to define database-specific selections in logical databases. For more information, see the keywords documentation and the section on Logical Databases [Page 1163].

Basic Form of Parameters [Page 690]

Dynamic Dictionary Reference [Page 691]

Default Values for Parameters [Page 692]

SPA/GPA Parameters as Default Values [Page 693]

Allowing Parameters to Accept Upper and Lower Case [Page 694]

Defining Required Fields [Page 696]

Checking Input Values [Page 698]

Defining Checkboxes [Page 699]

Defining Radio Buttons [Page 700]

Hiding Input Fields [Page 701]

Modifying Input Fields [Page 702]

Basic Form of Parameters

You use the PARAMETERS statement to declare variables, similarly to the DATA statement Variables declared with the PARAMETERS statement are called parameters. For each parameter declared, an input field appears on the corresponding selection screen. On the left hand side of the input field, the name of the parameter is displayed as text. You can modify this text as selection text [Ext.]. Values entered into the input field by the user are assigned to the corresponding variable while the system processes the selection screen. The position of the statement in the declaration part of the program determines the selection screen to which the input field belongs.

The basic form of the PARAMETERS statement is as follows:

PARAMETERS <p>[(<length>)] [TYPE <type>|LIKE <obj>] [DECIMALS <d>].

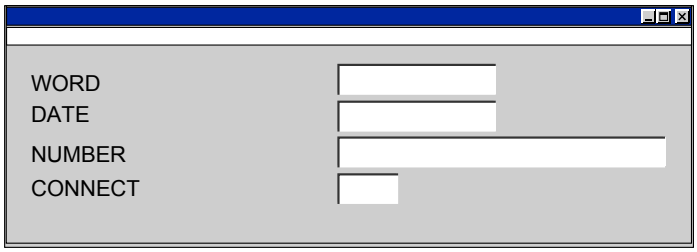
This statement creates parameter <p>. Currently, parameter names are limited to eight digits. The additions <length>, TYPE|LIKE and DECIMALS are the same as for the DATA [Page 123] statement. Note that the data types valid for parameters include all elementary ABAP types except data type F. You cannot use data type F, references and aggregate types.

If the parameter refers to data types from the Dictionary, it adopts all attributes of the Dictionary field. Currently, parameters can only refer to fields of database tables, views and structures. In particular, the field help (F1) and the possible entries help (F4) defined for these fields in the Dictionary are available to the user. To check a user entry against a value list in the Dictionary, you must use the special addition VALUE CHECK.

Parameters are used for simple queries of single values. For example, you can use parameters to control the program flow. However, if you want to restrict database accesses or perform complex selections, you should use selection criteria [Page 703].

```
REPORT DEMO.

PARAMETERS: WORD(10) TYPE C,
            DATE TYPE D,
            NUMBER TYPE P DECIMALS 2,
            CONNECT TYPE SPFLI-CONNID.
```

In this example, four parameters are created for the standard selection screen: a character field, WORD, of length 10; a date field, DATE, of length 8; a packed number field, NUMBER, with two decimals; and a field, CONNECT, that refers to the SPFLI-CONNID type in the ABAP Dictionary. When the user starts the program, the input fields appear as follows on the standard selection screen:



Dynamic Dictionary Reference

If you want the data type of a parameter to refer dynamically to a data type from the ABAP Dictionary, use the following syntax:

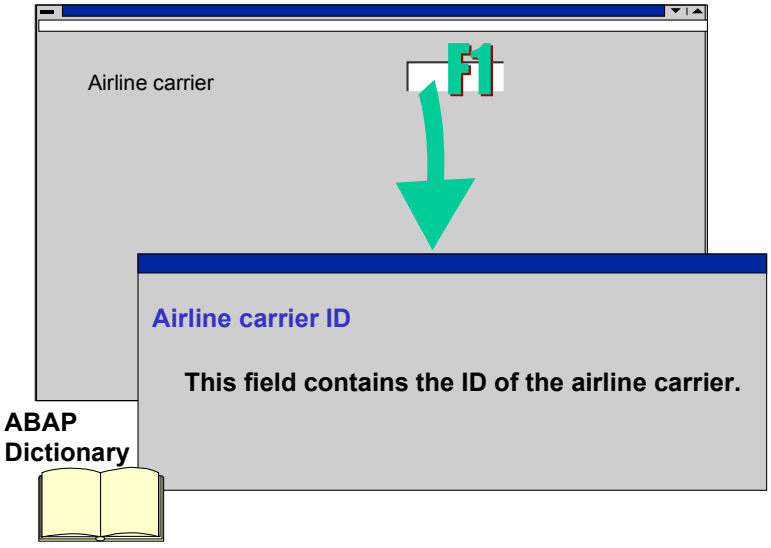
PARAMETERS <p> LIKE (<name>) ...

At the time the selection screen is called, field <name> must contain the name from a data type of the ABAP Dictionary. Currently, parameters can only refer to fields of database tables, views and structures. The parameter dynamically adopts the attributes of the Dictionary type, that is, technical properties and help texts. If you do not create a selection text for the parameter in the program, the field label from the Dictionary appears as the description on the selection screen. Otherwise, the selection text is displayed.

The contents of field <name> are taken from the program in which the selection screen is defined. If the selection screen is defined in the selection part of a logical database, then field <name> of the associated database program is used.

```
REPORT DEMO.

DATA NAME(20) TYPE C.

SELECTION-SCREEN BEGIN OF SCREEN 500.
  PARAMETERS P_CARR LIKE (NAME).
SELECTION-SCREEN END OF SCREEN 500.

NAME = 'SPFLI-CARRID'.

CALL SELECTION-SCREEN 500.
```

The selection screen called looks as follows:



The field length, the selection text and the field help are dynamically taken from field CARRID of database table SPFLI.

Default Values for Parameters

To assign a default value to a parameter, you use the following syntax:

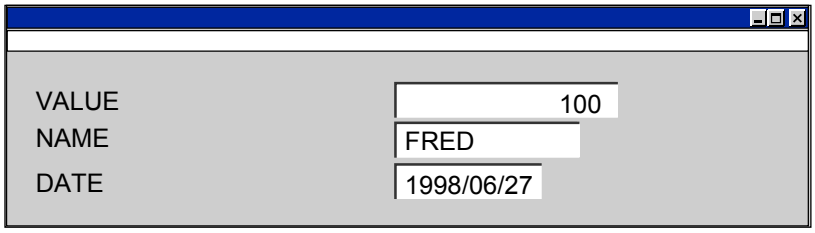
PARAMETERS <p> ..... DEFAULT <f> .....

Default value <f> can be either a literal or a field name. You can only use fields that contain a value when the program is started. These fields include several predefined data objects [Page 132].

The input field of the parameter on the selection screen is filled with the default value. The user can accept or change this value.

```
REPORT DEMO.

PARAMETERS: VALUE TYPE I DEFAULT 100,
            NAME LIKE SY-UNAME DEFAULT SY-UNAME,
            DATE LIKE SY-DATUM DEFAULT '19980627'.
```

If the name of the program user is FRED, the selection screen appears as follows:



Note that the default value of the field DATE appears formatted according to the user's master record on the selection screen.

SPA/GPA Parameters as Default Values

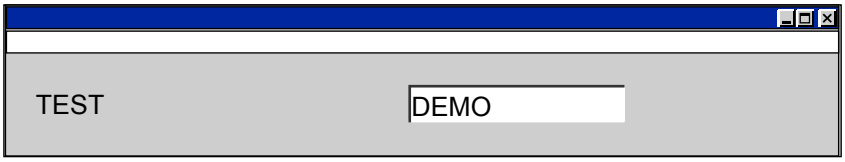
The SPA/GPA Parameter Technique [Page 1033] is a general procedure for filling the initial screen when a program is called. To use this technique for parameters on selection screens, you must link the parameter to an SPA/GPA parameter from the SAP memory as follows:

PARAMETERS <p> ..... MEMORY ID <pid>.....

If you use this addition, the current value of SPA/GPA parameter <pid> from the global user-related SAP memory is assigned to parameter <p> as a default value. Description <pid> can contain a maximum of twenty characters and must not be enclosed in quotation marks.

```
REPORT DEMO.

PARAMETERS TEST(16) MEMORY ID RID.
```

The selection screen looks as follows:



Parameter TEST is linked to SPA/GPA parameter RID which is used in the R/3 System to store the name of the program that was processed last. SPA/GPA parameter RID, for example, is also linked to the input fields for the program name on the screens of transactions SE38 and SA38.

Allowing Parameters to Accept Upper and Lower Case

To allow upper or lower case for character string parameters, you use the following syntax:

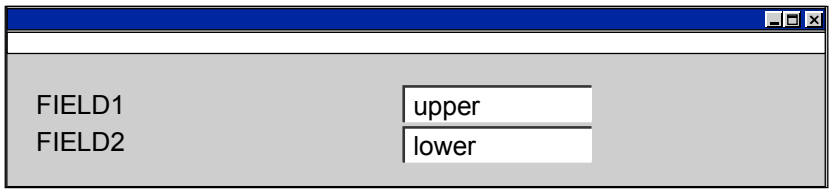
PARAMETERS <p> ..... LOWER CASE .....

If you do not use the LOWER CASE addition, all input values are automatically converted into upper case.

If you use the TYPE addition to refer to data types from the ABAP Dictionary, the parameter adopts all attributes of the Dictionary field. These attributes cannot be changed, and you cannot use the LOWER CASE addition. The possibility for entering either upper- or lower-case values must be defined in the ABAP Dictionary.

```
REPORT DEMO.

PARAMETERS: FIELD1(10),
            FIELD2(10) LOWER CASE.

WRITE: FIELD1, FIELD2.
```

Assume the following input values on the selection screen:



The output appears as follows:

```
UPPER      lower
```

The contents of FIELD1 are changed to upper case.

Reducing the Visible Length

By default, the length of an input field on the selection screen is the same as the length of the parameter in the ABAP program. However, you can define the visible length of a parameter as smaller than its actual length (as is also possible for input/output fields on screens):

PARAMETERS <p> ... VISIBLE LENGTH <len> ...

If <len> is smaller than the field length of <p>, the input field is displayed in the length <len>. In all other cases, the parameter is displayed in its full length. Part of the contents of parameters with a shorter visible length will be obscured, but the field is automatically set to scrollable.

```
REPORT demo_sel_screen_vis_len.

PARAMETERS: p1(10) TYPE c VISIBLE LENGTH 1,
            p2(10) TYPE c VISIBLE LENGTH 5,
            p3(10) TYPE c VISIBLE LENGTH 10.

START-OF-SELECTION.
  WRITE: / 'P1:', p1,
         / 'P2:', p2,
         / 'P3:', p3.
```

The three parameters P1, P2, and P3 have the same length (10), but are displayed in different lengths on the selection screen. However, you can enter up to ten characters in any of the fields.

Defining Required Fields

To define the input field of a parameter as a required field, you use the following syntax:

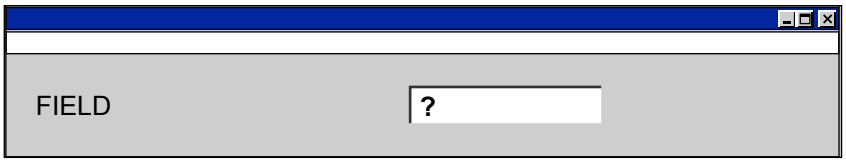
PARAMETERS <p> ..... OBLIGATORY .....

When you use this option, a question mark appears in the input field for parameter <p>.The user cannot continue with the program without entering a value in this field on the selection screen.

REPORT DEMO.

PARAMETERS FIELD(10) OBLIGATORY.

The selection screen looks as follows:



Search Helps for Parameters

A search help is a ABAP Dictionary object used to define possible values (F4) help (see Input Help in the ABAP Dictionary [Page 596]). You can link a search help to a parameter as follows:

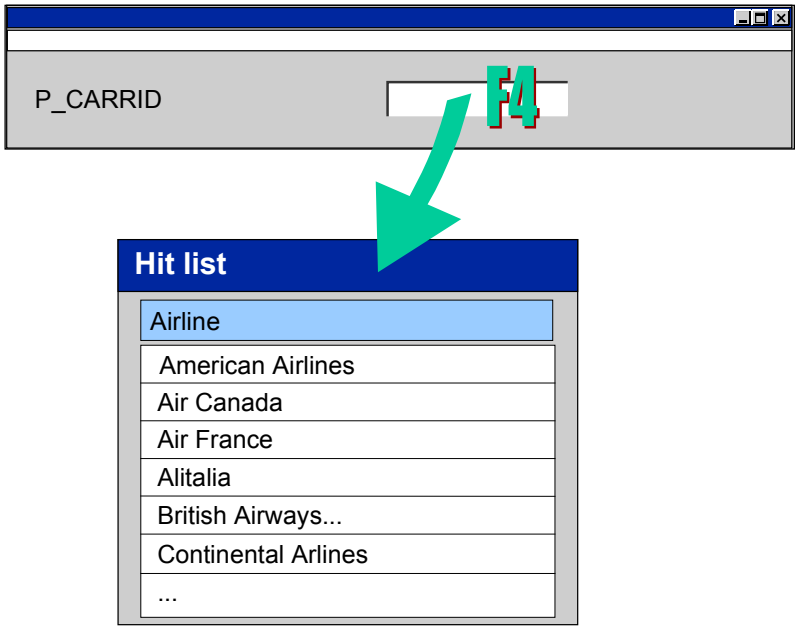
PARAMETERS <p> ... MATCHCODE OBJECT <search_help>.

The search help <search_help> must be defined in the ABAP Dictionary. The system now automatically displays the input help button for the field on the screen and activates the F4 key for it. When the user requests input help, the hit list of the search help appears, and when he or she selects an entry, the corresponding export parameter is placed in the input field.

The predecessors of search helps in the ABAP Dictionary were called matchcode objects, hence the name of the addition in the PARAMETERS statement. Existing matchcode objects are still supported.

```
REPORT demo_sel_screen_parameters_mco.

PARAMETERS p_carrid TYPE s_carr_id
           MATCHCODE OBJECT demo_f4_de.
```

The selection screen looks as follows:



The search help DEMO_F4_DE is defined in the ABAP Dictionary. The search help reads the columns CARRID and CARRNAME from the database table SCARR. Only CARRNAME is listed, but CARRID is flagged as an export parameter. When you choose a line, the airline code CARRID is placed in the input field.

Checking Input Values

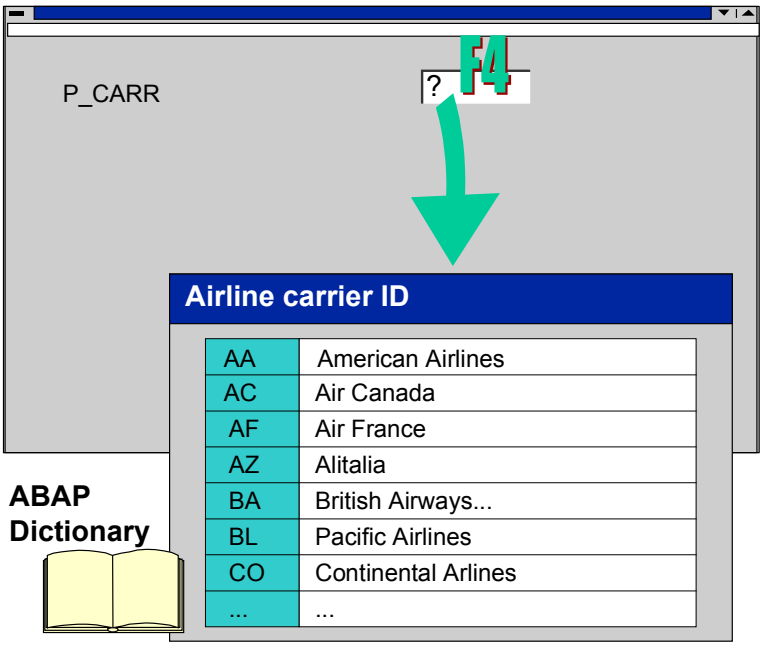
To check a user entry against a check table or against fixed values in the ABAP Dictionary, you use the following syntax:

PARAMETERS <p> TYPE <type> ... VALUE CHECK ...

In the TYPE addition, the data type of the parameter must refer to a data type from the ABAP Dictionary. Currently, parameters can only refer to fields of database tables, views and structures. If a check table or a list of fixed values is defined in the ABAP Dictionary for the relevant type, the user can only enter these valid values. Otherwise, the system issues an error message in the status bar of the selection screen when the selection screen is processed. Since this check is performed even if the input field is empty, you should use the VALUE CHECK addition only for required fields.

```
REPORT DEMO.

PARAMETERS P_CARR LIKE SPFLI-CARRID OBLIGATORY VALUE CHECK.
```

Parameter P_CARR is declared with reference to field CARRID of database table SPFLI. For this field, check table SCARR is specified in the ABAP Dictionary. The user can only enter carrier ID values that are contained in SCARR. The possible entries help of the input field for P_CARR displays the allowed values.



Defining Checkboxes

To define the input field of a parameter as a checkbox, you use the following syntax:

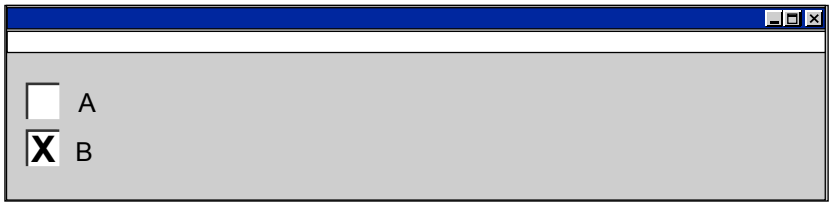
PARAMETERS <p> ..... AS CHECKBOX .....

Parameter <p> is created with type C and length 1. In this case, you may not use the additions TYPE and LIKE. Valid values for <p> are ' ' and 'X'. These values are assigned to the parameter when the user clicks the checkbox on the selection screen.

If you use the TYPE addition to refer to a data type in the ABAP Dictionary [Page 105] of type CHAR and length 1 for which 'X' and ' ' are defined as valid values in the domain, the parameter automatically appears as a checkbox on the selection screen.

```
REPORT DEMO.

PARAMETERS: A AS CHECKBOX,
            B AS CHECKBOX DEFAULT 'X'.
```

The selection screen looks as follows:



Two checkboxes appear on the left side of the selection screen with the selection text appearing on their right. Checkbox B has the default value of 'X'.

Defining Radio Buttons

To define the input field of a parameter as a radio button, you use the following syntax:

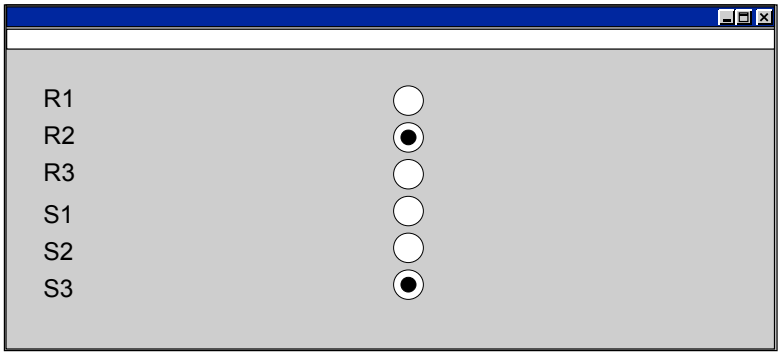
PARAMETERS <p> ..... RADIOBUTTON GROUP <radi>.....

Parameter <p> is created with type C and length 1, and is assigned to group <radi>. The maximum length of string <radi> is 4. You can use additions TYPE and LIKE, but you must refer to a field of type C with length 1. You must assign at least two parameters to each <radi> group. Only one parameter per group can have a default value assigned using the DEFAULT addition. The default value must be 'X'. If you do not use the DEFAULT addition, the first parameter of each group ist set to 'X'.

When the user clicks a radio button on the selection screen, the respective parameter is assigned the value 'X', while all other parameters of the same group are assigned the value ' '.

```
REPORT DEMO.

PARAMETERS: R1 RADIOBUTTON GROUP RAD1,
            R2 RADIOBUTTON GROUP RAD1 DEFAULT 'X',
            R3 RADIOBUTTON GROUP RAD1,

            S1 RADIOBUTTON GROUP RAD2,
            S2 RADIOBUTTON GROUP RAD2,
            S3 RADIOBUTTON GROUP RAD2 DEFAULT 'X'.
```

The selection screen looks as follows:



Radio buttons R1, R2 and R3 form group RAD1, while S2, S2 and S3 form group RAD2. On the selection screen, R2 and S3 are selected, while all others are not.

Hiding Input Fields

To suppress the display of the input field on the selection screen, you use the following syntax:

PARAMETERS <p> ..... NO-DISPLAY .....

Although parameter <p> is declared, it is not displayed on the selection screen.

If the parameter belongs to the standard selection screen, you can assign a value to it either by using the DEFAULT addition when you declare it, or during the INITIALIZATION [Page 954] event. If you call the executable program using the SUBMIT statement, the calling program can also pass the value.

When you use user-defined selection screens, you can assign a value to the parameter at any time before calling the selection screen.

If you want to display a parameter only in certain cases, for example, depending on the values entered by the user in other input fields of the selection screen, you cannot use the NO-DISPLAY addition. If you use NO-DISPLAY, the parameter actually is an element of the interface for program calls, but not an element of the selection screen. As a result, you cannot make it visible using the MODIFY SCREEN [Page 702] statement.

To hide a parameter that is an element of the selection screen, you must declare it without the NO-DISPLAY addition and suppress its display using the MODIFY SCREEN statement.

Modifying Input Fields

To modify the appearance of an input field on the selection screen, you must assign the parameter to a modification group as follows:

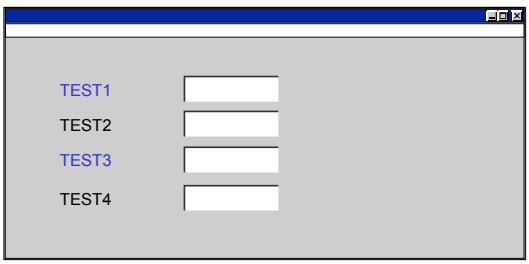
PARAMETERS <p> ..... MODIF ID <key> .....

The name of modification group <key> should be a three-character variable name without quotation marks. The MODIF ID addition always assigns <key> to the SCREEN-GROUP1 column of internal table SCREEN. Parameters assigned to a modification group can be processed as an entire group with the LOOP AT SCREEN and MODIFY SCREEN statements during the AT SELECTION-SCREEN OUTPUT [Page 743] event.

For more information on modification groups and internal table SCREEN, see Modifying the Screen [Page 611].

```
REPORT DEMO.

PARAMETERS: TEST1(10) MODIF ID SC1,
            TEST2(10) MODIF ID SC2,
            TEST3(10) MODIF ID SC1,
            TEST4(10) MODIF ID SC2.

AT SELECTION-SCREEN OUTPUT.

LOOP AT SCREEN.
  IF SCREEN-GROUP1 = 'SC1'.
    SCREEN-INTENSIFIED = '1'.
    MODIFY SCREEN.
    CONTINUE.
  ENDIF.
  IF SCREEN-GROUP1 = 'SC2'.
    SCREEN-INTENSIFIED = '0'.
    MODIFY SCREEN.
  ENDIF.
ENDLOOP.
```

The parameters TEST1 and TEST3 are assigned to group SC1, while TEST2 and TEST4 are assigned to group SC2. During the AT SELECTION-SCREEN OUTPUT event, the INTENSIFIED field of internal table SCREEN is set to 1 or 0, depending on the contents of the GROUP1 field. On the selection screen, the lines for TEST1 and TEST3 are highlighted while those for TEST2 and TEST4 are not, as shown below:



Defining Complex Selections

The by far most important function of selection screens is to allow users to minimize the data to be read before database tables are actually accessed. In this context, a user can also be a calling ABAP program. This important function of selection screens is also illustrated in Executing Reports Directly [Page 945].

However, the input fields that accept single values and are defined using PARAMETERS cannot fulfill this task. For this purpose, ABAP contains selection criteria. These selection criteria allow the user to easily handle complex selections. You can link selection criteria to the columns of database tables and to internal fields in a program. Each selection criterion may only apply to one column of a database table. However, a database table can have more than one selection criterion linked to it (for example, one criterion for each column). If selection criteria are used for complex selections, you do not have to write lengthy logical expressions, since this task is solved internally.

Unlike parameters that are declared as elementary variables in ABAP programs, selection criteria are based on special internal tables, called selection tables. To define a selection criterion, you must declare a selection table in the declaration part using the SELECT-OPTIONS statement. The relevant possible entries then appear on the selection screen.

You can use the SELECT-OPTIONS statement for both standard and user-defined selection screens. If the program is linked to a logical database, you must consider several special rules for handling selection criteria. The following sections explain the variants of the SELECT-OPTIONS statement which are important for program-specific selections.

There are special variants of the SELECT-OPTIONS statement that you can use to define database-specific selections in logical databases. For more information, see the keywords documentation and the section on Logical Databases [Page 1163].

Selection Tables

You use the statement

SELECT-OPTIONS <seltab> for <f>.

to declare a selection table in the program that is linked to the <f> column of a database table, or to an internal <f> field in the program. A selection table is an internal table object of the standard table type that has a standard key and a header line. Selection tables are used to store complex selections using a standardized procedure. They can be used in several ways. Their main purpose is to directly translate the selection criteria into database selections using the WHERE addition in Open SQL statements.

In addition to selection tables that you create using SELECT-OPTIONS, you can use the RANGES statement to create internal tables that have the structure of selection tables. You can use these tables with certain restrictions the same way you use actual selection tables.

Structure of Selection Tables

The row type of a selection table is a structure that consists of the following four components: SIGN, OPTION, LOW and HIGH. Each row of a selection table that contains values represents a sub-condition for the complete selection criterion. Description of the individual components:

- SIGN

  The data type of SIGN is C with length 1. The contents of SIGN determine for each row whether the result of the row condition is to be included in or excluded from the resulting set of all rows. Possible values are I and E.

– I stands for “inclusive” (inclusion criterion - operators are not inverted)

– E stands for “exclusive” (exclusion criterion - operators are inverted)

- OPTION

  The data type of OPTION is C with length 2. OPTION contains the selection operator. The following operators are available:

– If HIGH is empty, you can use EQ, NE, GT, LE, LT,CP, and NP. These operators are the same as those that are used for logical expressions [Page 225]. Yet operators CP and NP do not have the full functional scope they have in normal logical expressions. They are only allowed if wildcards ('*' or '+') are used in the input fields, and no escape character is defined.

– If HIGH is filled, you can use BT (BeTween) and NB (Not Between). These operators correspond to BETWEEN and NOT BETWEEN that you use when you check if a field belongs to a range [Page 235].

- LOW

  The data type of LOW is the same as the column type of the database table, to which the selection criterion is linked.

– If HIGH is empty, the contents of LOW define a single field comparison. In combination with the operator in OPTION, it specifies a condition for the database selection.

– If HIGH is filled, the contents of LOW and HIGH specify the upper and lower limits for a range. In combination with the operator in OPTION, the range specifies a condition for the database selection.

- HIGH

  The data type of HIGH is the same as the column type of the database table, to which the selection criterion is linked. The contents of HIGH specify the upper limit for a range selection.

If the selection table contains more than one row, the system applies the following rules when creating the complete selection criterion:

1. Form the union of sets defined on the rows that have SIGN field equal to I (inclusion).
2. Subtract the union of sets defined on the rows that have SIGN field equal to E (exclusion).
3. If the selection table consists only of rows in which the SIGN field equals E, the system selects all data outside the set specified in the rows.

RANGES Tables

You can use the RANGES statement to create internal tables of the same type as selection tables.

RANGES <rangetab> FOR <f>.

This statement is simply a shortened form of the following statements:

```
DATA: BEGIN OF <rangetab> OCCURS 0,
        SIGN(1),
        OPTION(2)
        LOW  LIKE <f>,
        HIGH LIKE <f>,
      END OF <rangetab>.
```

Internal tables created with RANGES have the same structure as selection tables, but they do not have the same functionality.

Selection tables created with RANGES are not components of the selection screen. As a result, no relevant input fields are generated. Also, you cannot use a RANGES table as a data interface in program <prog> called by the following statement:
SUBMIT <prog> WITH <rangetab> IN <table>.

However, you can use RANGES to create the table <table> in the calling program. The main function of RANGES tables is to pass data to the actual selection tables without displaying the selection screen when executable programs are called [Page 1018].

Although you can use RANGES tables like actual selection tables in the WHERE clause of Open SQL statements and in combination with the IN operator in logical expressions, they are not linked to a database table. This means that RANGES tables:

- are not passed like selection criteria to logical databases [Page 711]
- cannot be used with the shortened form of selection tables in logical expressions [Page 766]
- cannot be used like selection criteria in GET events [Page 769]

```
REPORT DEMO1.

RANGES S_CARRID FOR SPFLI-CARRID.
```

```
S_CARRID-SIGN   = 'I'.
S_CARRID-OPTION = 'EQ'.
S_CARRID-LOW    = 'LH'.

APPEND S_CARRID.

SUBMIT DEMO2 WITH CARRID IN S_CARRID.
```

In this example, RANGES table S_CARRID is created with reference to column CARRID of database table SPFLI. Fields S_CARRID-LOW and S_CARRID-HIGH have the same type as CARRID. The header line of internal table S_CARRID is filled and appended to the table. Program DEMO2 is called. If DEMO2 is linked to logical database F1S, its selections screen contains the fields of selection criterion CARRID from the logical database. These fields are filled with the contents of the RANGES table.

Basic Form of Selection Criteria

You use the SELECT-OPTIONS statement to declare selection tables [Page 704] and create corresponding input fields on the associated selection screen. You can modify the associated text as selection text [Ext.]. When the selection screen is processed, the values entered by the user into the input fields are assigned to the header line of the selection table and appended to the table. The position of the statement in the declaration part of the program determines the selection screen to which the input fields belong.

The basic form of the SELECT-OPTIONS statement is as follows:

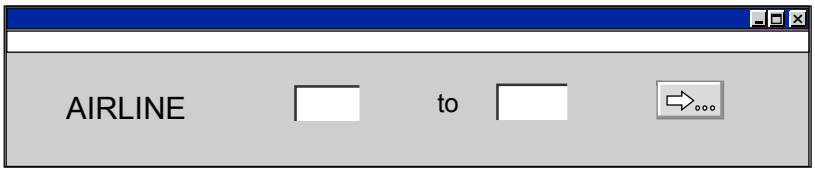
SELECT-OPTIONS <seltab> FOR <f>.

It declares a selection criterion <seltab> that is linked to a field <f> that has already been declared locally in the program. The names of selection criteria are currently restricted to eight characters. Valid data types of <f> include all elementary ABAP types except data type F. You cannot use data type F, references and aggregate types. If you want to use the selection criterion to restrict database selections, it is a good idea to declare <f> with reference to a column of the corresponding database table. It then inherits all of the attributes of the data type already defined in the ABAP Dictionary. In particular, the field help (F1) and the possible entries help (F4) defined for these fields in the Dictionary are available to the user on the selection screen.

The selection table, which has the same name as the selection criterion, <seltab>, is usually filled by the user on the selection screen or by calling programs. You can also process the selection table like any other internal table [Page 251] in the program.

```
REPORT DEMO.

DATA WA_CARRID TYPE SPFLI-CARRID.

SELECT-OPTIONS AIRLINE FOR WA_CARRID.

LOOP AT AIRLINE.
  WRITE: / 'SIGN:',   AIRLINE-SIGN,
           'OPTION:', AIRLINE-OPTION,
           'LOW:',    AIRLINE-LOW,
           'HIGH:',   AIRLINE-HIGH.
ENDLOOP.
```

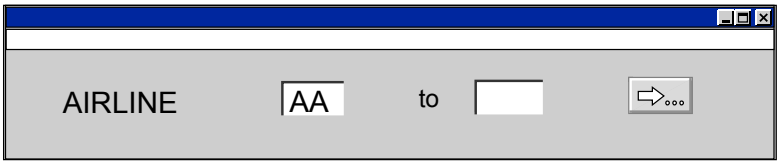
After the executable program DEMO has been started, the following standard selection screen appears:



Two input fields and a pushbutton to enter additional values for the selection criterion are displayed. The value that the user enters into the first input field is written into the AIRLINE-LOW component of the selection table. The value that the user enters into the second input field is written into the AIRLINE-HIGH component of the selection table.

Basic Form of Selection Criteria

If the user enters values as shown below:

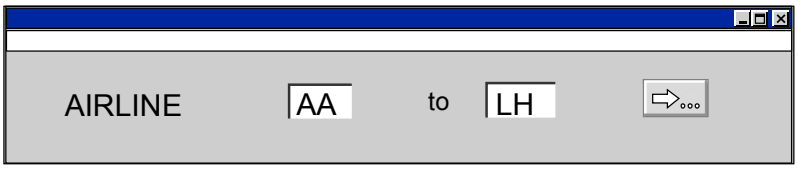


The output appears as follows:

```
SIGN: I OPTION: EQ LOW: AA HIGH:
```

If the user leaves the second field blank (single field comparison), the default settings for SIGN and OPTION are I and EQ.

If the user enters values as shown below:

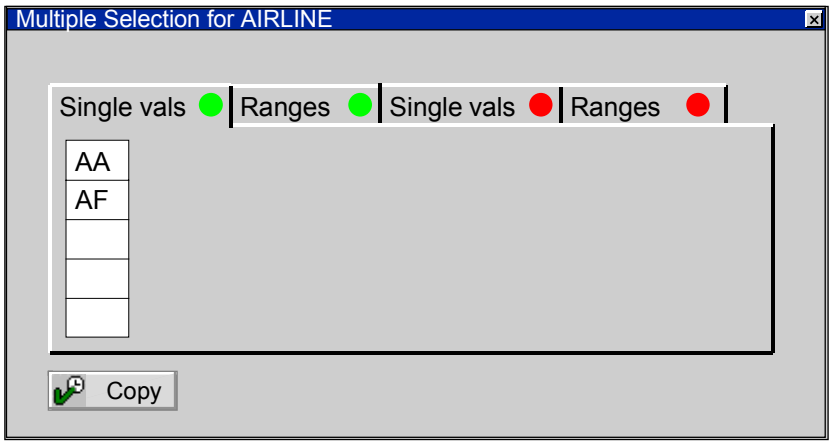


The output appears as follows:

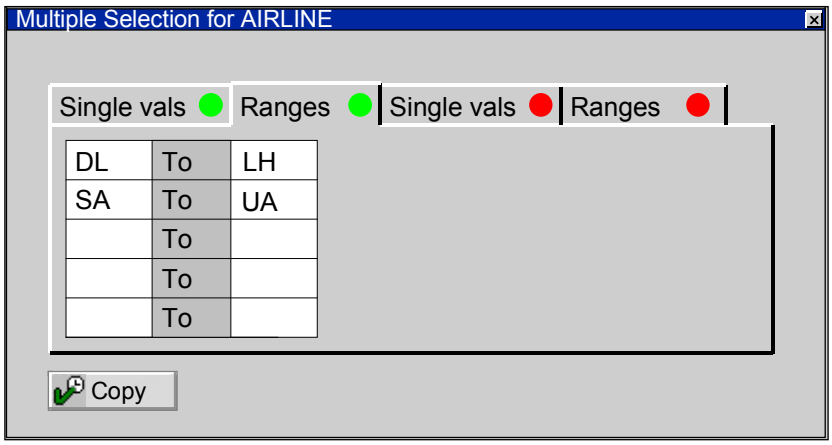
```
SIGN: I OPTION: BT LOW: AA  HIGH: LH
```

If the user enters a value in the second field (ranges selection), the default settings for SIGN and OPTION are I and BT.

To set up a more complex selection pattern, the user can choose the  pushbutton on the right side of the selection screen to display the *Multiple Selection* window and enter the following values:



and:



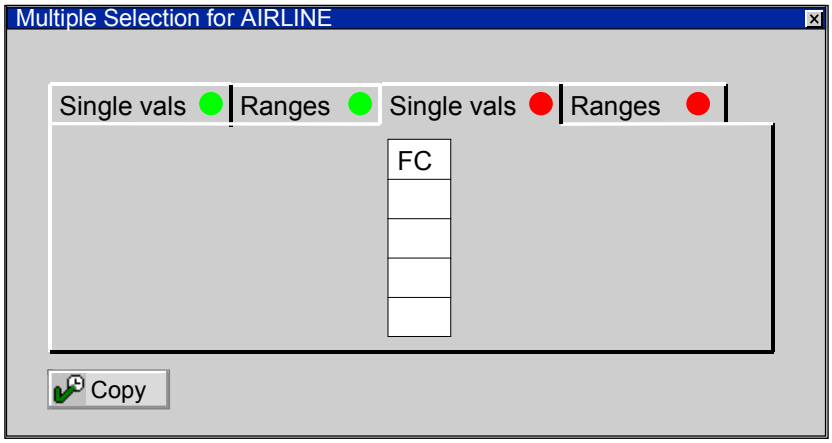
If the user adopts the multiple selection, the arrow on the selection screen turns green to indicate that multiple selections have been entered.

The output is:

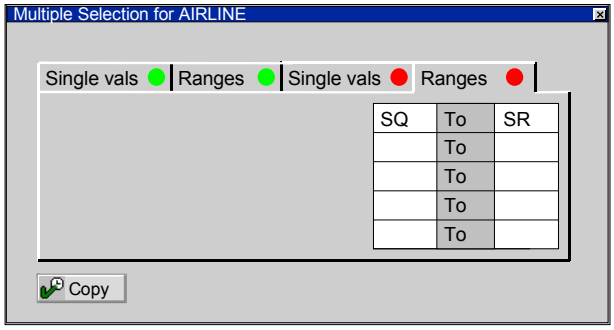
```
SIGN: I OPTION: EQ LOW: AA HIGH:
SIGN: I OPTION: EQ LOW: AF HIGH:
SIGN: I OPTION: BT LOW: DL HIGH: LH
SIGN: I OPTION: BT LOW: SA HIGH: UA
```

Using a multiple selection means to append several rows to a selection table.

The user can also specify exclusive criteria as multiple selections:



and:

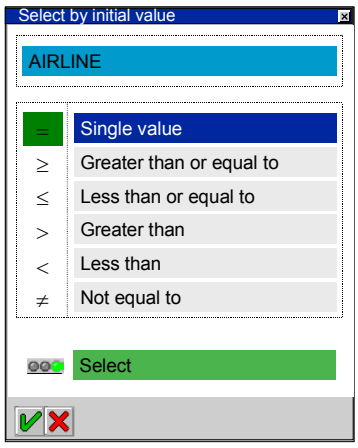


The output is:

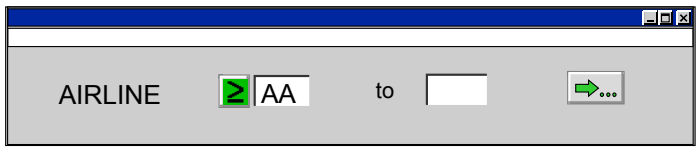
```
SIGN: E OPTION: EQ LOW: FC HIGH:
SIGN: E OPTION: BT LOW: SQ HIGH: SR
```

If exclusive criteria are used, the SIGN component of the selection table contains value E.

To explicitly set the SIGN and OPTION values, the user must select the input fields on the selection screen or the *Multiple Selection* screen either by double-clicking them or by choosing F2.



On the *Maintain Selection Options* screen that, for single field comparisons, looks like this, the user can select an operator for the OPTION field and switch between I and E for the SIGN field in the lowest line. On the selection screen, symbols on the left side of the input field are used to indicate the selection.



These symbols are green to show that the SIGN field has value I. If the SIGN field has value E, the symbols are red.

In this case, the list appears as follows:

```
SIGN: E OPTION: GE LOW: AA HIGH:
```

Selection Criteria and Logical Databases

If a program is linked to a logical database, and you define a selection criterion in the program that is assigned to a database table of this same logical database, you must distinguish between two cases, that is, whether or not dynamic selections are possible for the database table.

Dynamic Selections Not Possible

If you define the selection criterion for a column of a database table that does **not** support dynamic selections, this selection criterion does not affect the amount of data read by the logical database. You can perform corresponding checks during a GET [Page 958] event after a data record has been read.

Dynamic Selections Possible

If you define the selection criterion for a column of a database table that supports **dynamic selections**, the values entered on the selection screen are transferred to the logical database. There, they are treated as dynamic selections. The logical database does not read records from the database table that do not meet these selection criteria. This kind of selection is much more efficient than for database tables that are not designated for dynamic selections.

Besides, the input fields for the corresponding dynamic selection are displayed on the selection screen from the start. This spares the user from having to choose *Dynamic Selections* to display the corresponding screen.

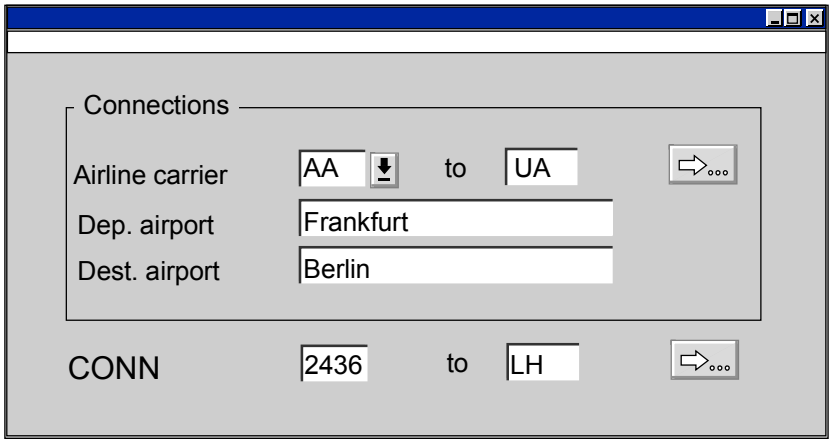
However, if the logical database is to read these rows anyway, since, for example, the selection criterion is not to be used for restricting database accesses, you must use the following special addition:

SELECT-OPTIONS <seltab> FOR <f> ..... NO DATABASE SELECTION .....

The following program is linked to logical database F1S.

```
REPORT DEMO.

NODES SPFLI.

SELECT-OPTIONS CONN FOR SPFLI-CONNID NO DATABASE SELECTION.

GET SPFLI.
   IF SPFLI-CONNID IN CONN.
     WRITE: SPFLI-CARRID, SPFLI-CONNID, 'meets criterion'.
   ELSE.
     WRITE: SPFLI-CARRID, SPFLI-CONNID,
                          'does not meet criterion'.
   ENDIF.
```

The following selection screen is displayed. The first part is defined in the logical database, and the last line (CONN) is defined in the program.



If the user fills the input fields as shown above, the output is as follows:

```
LH  2402 does not meet criterion
LH  2436 meets criterion
LH  2462 does not meet criterion
```

The following program does not use the NO DATABASE SELECTION option:

```
REPORT DEMO.

NODES SPFLI.

SELECT-OPTIONS CONN FOR SPFLI-CONNID.

GET SPFLI.
   IF SPFLI-CONNID IN CONN.
     WRITE: SPFLI-CARRID, SPFLI-CONNID, 'meets criterion'.
   ELSE.
WRITE: SPFLI-CARRID, SPFLI-CONNID,
                    'does not meet criterion'.
   ENDIF.
```

With the same entries as above, the output appears as follows:

```
LH  2436 meets criterion
```

The result is the same as if you did not use the SELECT-OPTIONS statement, and the user selected *Dynamic Selections* in the application toolbar of the selection screen and then entered 2436 there.

Default Values for Selection Criteria

To assign default values to a selection criterion, you use the following syntax:

SELECT-OPTIONS <seltab> FOR <f> DEFAULT <g> [TO <h>] ....

Default values <g> and <h> can be literals or field names. You can only use fields that contain a value when the program is started. These fields include several predefined data objects [Page 132].

For each SELECT-OPTIONS statement, you can specify only one DEFAULT addition. This means that you can fill only the **first row** of selection table <seltab> with default values. To fill more rows with default values, the selection table must be processed before the selection screen is called, for example, during the AT SELECTION-SCREEN -OUTPUT [Page 743] event.

You use the DEFAULT addition as follows to address the individual components of the first row:

- To fill only the LOW field (single field comparison), use:

  .....DEFAULT <g>.

- To fill the LOW and HIGH fields (range selection), use:

  .....DEFAULT <g> TO <h>.

- To fill the OPTION field, use:

  .....DEFAULT <g> [to <h>] OPTION <op>.

  For single field comparisons, <op> can be EQ, NE, GE, GT, LE, LT, CP, or NP. The default value is EQ. For range selections, <op> can be BT or NB. The default value is BT.

- To fill the SIGN field, use:

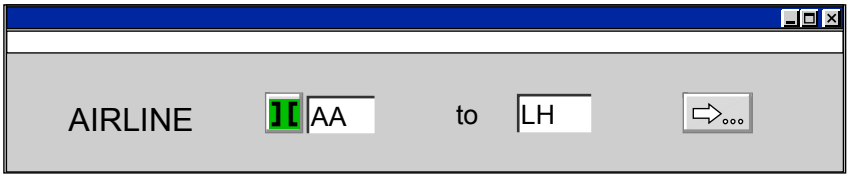
  .....DEFAULT <g> [to <h>] [OPTION <op>] SIGN <s>.

  The value of <s> can be I or E. The default value is I.

The input fields of the selection criterion are filled with the default values. The user can accept or change these values.

```
REPORT DEMO.

DATA WA_SPFLI TYPE SPFLI.

SELECT-OPTIONS AIRLINE FOR WA_SPFLI-CARRID
               DEFAULT 'AA'
                    TO 'LH'
                OPTION  NB
                  SIGN  I.
```

The following standard selection screen appears:



The symbol before the first field *FROM* shows that the AIRLINE-OPTION field contains operator NB. This symbol is green to show that the AIRLINE-SIGN field contains value I. The arrow on the right pushbutton is not green since only one row of the selection table is filled.

Restricting Entry to One Row

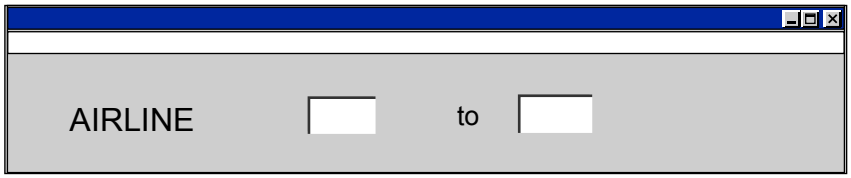
To allow the user to process only the first row of the selection table on the selection screen, you use the following syntax:

SELECT-OPTIONS <seltab> FOR <f> ..... NO-EXTENSION .....

As a result, the pushbutton for multiple selections does not appear on the selection screen, and multiple selections are not available to the user.

```
DATA WA_SPFLI TYPE SPFLI.

SELECT-OPTIONS AIRLINE FOR WA_SPFLI-CARRID NO-EXTENSION.
```

The following standard selection screen appears:



Restricting Entry to Single Fields

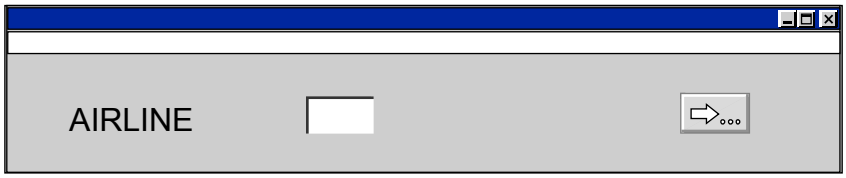
To allow the user to process only single fields on the selection screen, you use the following syntax:

SELECT-OPTIONS <seltab> FOR <f> ..... NO INTERVALS .....

As a result, the second input field does not appear on the selection screen. The user can only enter a single field comparison for the first row of the selection table. However, the user can call the *Multiple Selection* screen and enter range selections there.

```
DATA WA_SPFLI TYPE SPFLI.
SELECT-OPTIONS AIRLINE FOR WA_SPFLI-CARRID NO INTERVALS.
```

The following standard selection screen appears:

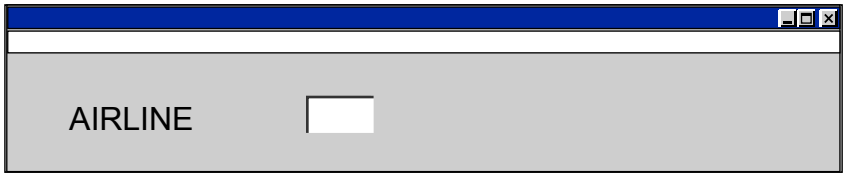


The user can enter only a single field directly. However, the user can enter multiple selections by clicking the pushbutton on the right side of the screen.

If you use the NO-EXTENSION addition as follows:

```
DATA WA_SPFLI TYPE SPFLI.
SELECT-OPTIONS AIRLINE FOR WA_SPFLI-CARRID NO INTERVALS
                           NO-EXTENSION.
```

the selection screen looks like this:



Now, the user can actually enter only a single field comparison.

Additional Options for Selection Criteria

VISIBLE LENGTH <len> [Page 695]

MATCHCODE OBJECT [Page 697]

Besides the special additions, there are a number of other additions that you can use with the SELECT-OPTIONS statement and that have the same syntax and the same function as for the PARAMETERS [Page 689] statement.

- SPA/GPA parameters as default values

  SELECT-OPTIONS <seltab> FOR <f> ... MEMORY ID <pid> [Page 693].....
- Upper and lower case for selection criteria:

  SELECT-OPTIONS <seltab> FOR <f> ... LOWER CASE [Page 694] .....
- To make the *From* field a required field on the selection screen, use:

  SELECT-OPTIONS <selcrit> FOR <f> ... OBLIGATORY [Page 696] .....
- To hide input fields on the selection screen, use:

  SELECT-OPTIONS <selcrit> FOR <f> ... NO DISPLAY [Page 701] .....
- To modify input fields on the selection screen, use:

  SELECT-OPTIONS <selcrit> FOR <f> ... MODIF ID <key> [Page 702] .....

Formatting Selection Screens

The selection screen that you define when you use the PARAMETERS or SELECT-OPTIONS statements on their own, has a standard layout in which all parameters appear line by line. This layout is not always sufficient. For example, when you define a group of radio buttons, you should set off these buttons against other input fields so that the user can identify them as a group.

The SELECTION-SCREEN statement has its own formatting options that you can use to define the layout for selection screens. You can define the layout of parameters and selection criteria and display comments and underlines on the selection screen. In addition, you can place pushbuttons in the application toolbar and on the screen itself.

You can only see the layout of a selection screen if you call that screen. However, standard selection screens in executable programs are only called if they contain at least one input field declared using the PARAMETERS or SELECT-OPTIONS statements. This means that it does not make any sense to place comments, underlines or pushbuttons on a standard selection screen without including at least one input field. User-defined selection screens, on the other hand, can be called even if they do not contain an input field. In this case, you can use all formatting options, in particular pushbuttons, on selection screens without input fields.

Specifying Blank Lines, Underlines, and Comments [Page 719]

Several Elements in a Single Line [Page 721]

Blocks of Elements [Page 723]

Pushbuttons in the Application Toolbar

Pushbuttons on the Selection Screen

Blank Lines, Underlines, and Comments

Blank Lines

To place blank lines on the selection screen, you use:

SELECTION-SCREEN SKIP [<n>].

This statement generates <n> blank lines, where <n> can have a value between 1 and 9. To produce a single blank line, you can omit <n>.

Underlines

To place underlines on the selection screen, you use:

SELECTION-SCREEN ULINE [[/]<pos(len)>] [MODIF ID <key>].

This statement generates an underline. If you do not use the <pos(len)> addition, a new line is generated for the underline below the current line, and the underline has the same length as the line. If you use the <pos(len)> addition, the underline begins at position <pos> in the current line and continues for a length of <len> characters. With several elements in one line, you can also specify (<len>) without <pos>. A slash (/) produces a line feed.

For <pos>, you can specify a number or either one of the expressions POS_LOW and POS_HIGH. POS_LOW and POS_HIGH mark the positions of the two input fields of a selection criterion.

The MODIF ID <key> [Page 702] addition has the same function as for the PARAMETERS statement. You can use it to modify the underline before the selection screen is called.

Comments

To place comments on the selection screen, you use:

```
SELECTION-SCREEN COMMENT [/]<pos(len)> <comm> [FOR FIELD <f>]
                                              [MODIF ID <key>].
```

This statement writes the <comm> comment on the selection screen. For <comm>, you can specify a text symbol [Page 121] or a field name with a maximum length of eight characters. This character field must not be declared with the DATA statement, but is generated automatically with length <len>. The field must be filled before the selection screen is called. You must always specify the <pos(len)> addition. Only if there are several elements in one line, can you omit <pos>.

The text <comm> will be displayed, starting in column <pos>, for a length of <len>. If you do not use a slash (/), the comment is written into the current line; otherwise a new line is created.

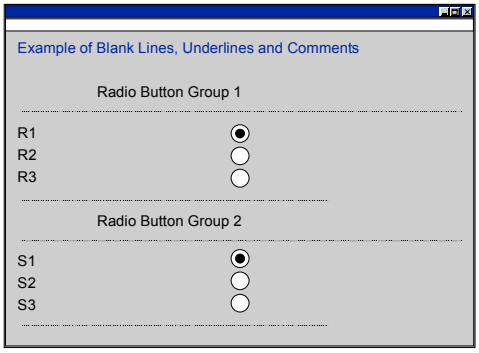
You use FOR FIELD <f> to assign a field label to the comment. <f> can be the name of a parameter or a selection criterion. As a result, if the user requests help on the comment on the selection screen, the help text for the assigned field <f> is displayed.

The MODIF ID <key> [Page 702] addition has the same function as for the PARAMETERS statement. You can use it to modify the comment before the selection screen is called.

Example

```
REPORT DEMO.

SELECTION-SCREEN COMMENT /2(50) TEXT-001 MODIF ID SC1.

SELECTION-SCREEN SKIP 2.
SELECTION-SCREEN COMMENT /10(30) COMM1.
SELECTION-SCREEN ULINE.

PARAMETERS: R1 RADIOBUTTON GROUP RAD1,
            R2 RADIOBUTTON GROUP RAD1,
            R3 RADIOBUTTON GROUP RAD1.

SELECTION-SCREEN ULINE /1(50).
SELECTION-SCREEN COMMENT /10(30) COMM2.
SELECTION-SCREEN ULINE.

PARAMETERS: S1 RADIOBUTTON GROUP RAD2,
            S2 RADIOBUTTON GROUP RAD2,
            S3 RADIOBUTTON GROUP RAD2.

SELECTION-SCREEN ULINE /1(50).

INITIALIZATION.

COMM1 ='Radio Button Group 1'.
COMM2 ='Radio Button Group 2'.

LOOP AT SCREEN.
    IF SCREEN-GROUP1 = 'SC1'.
            SCREEN-INTENSIFIED = '1'.
            MODIFY SCREEN.
    ENDIF.
ENDLOOP.
```

The following selection screen appears:



The text specified in text symbol 001, 'Example of Blank Lines, Underlines and Comments', appears highlighted. Two groups of radio buttons are displayed below the text, separated by underlines and described by comments. If there were no slashes (/) in the ULINE additions, the underlines would overwrite the last line of each radio button group.

Several Elements in a Single Line

To position a set of parameters or comments in a single line on the selection screen, you must declare the elements in a block enclosed by the following two statements:

SELECTION-SCREEN BEGIN OF LINE.
...
SELECTION-SCREEN END OF LINE.

Note that the selection text is **not** displayed when you use this option. To describe the elements, you must use the COMMENT addition of the SELECTION-SCREEN statement.

In the <pos(len)> formatting option of the SELECTION-SCREEN statement, you can omit <pos> between the statements specified above. The element is placed at the current position in the line. Between the statements specified above, you must not use a slash (/) in the <pos(len)> formatting option.

To determine the position of an element in a line, you use:

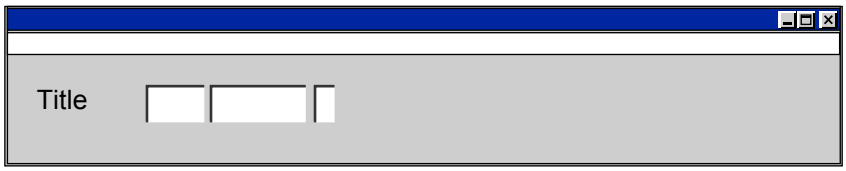
SELECTION-SCREEN POSITION <pos>.

For <pos>, you can specify a number or either one of the expressions POS_LOW and POS_HIGH. POS_LOW and POS_HIGH mark the positions of the input fields of a selection criterion. Use the POSITION addition only between BEGIN OF LINE and END OF LINE.

Examples

```
REPORT DEMO.

SELECTION-SCREEN BEGIN OF LINE.
  SELECTION-SCREEN COMMENT 1(10) TEXT-001.
  PARAMETERS: P1(3), P2(5), P3(1).
SELECTION-SCREEN END OF LINE.
```

The following selection screen appears:



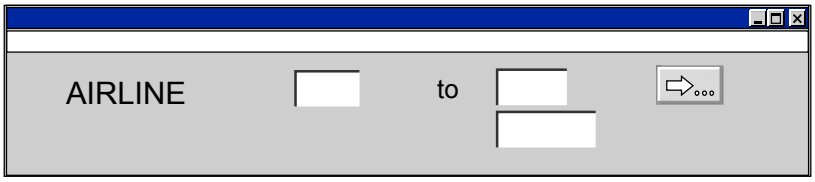
The line starts with the 'Title' contents of text symbol 001 and is followed by the input fields for parameters P1, P2, and P3.

```
REPORT DEMO.

DATA WA_SPFLI TYPE SPFLI.

SELECT-OPTIONS AIRLINE FOR WA_SPFLI-CARRID.
```

Several Elements in a Single Line

```
SELECTION-SCREEN BEGIN OF LINE.
   SELECTION-SCREEN POSITION POS_HIGH.
   PARAMETERS FIELD(5).
SELECTION-SCREEN END OF LINE.
```

The following selection screen appears:



The input field for the FIELD parameter appears below the second field of selection criterion AIRLINE. For FIELD, no selection text is displayed.

Blocks of Elements

To create a logical block of elements on the selection screen, you use:

```
SELECTION-SCREEN BEGIN OF BLOCK <block>
                 [WITH FRAME [TITLE <title>]]
                 [NO INTERVALS].
  ...
SELECTION-SCREEN END OF BLOCK <block>.
```

You must define a <block> name for each block. Blocks of elements can be nested.

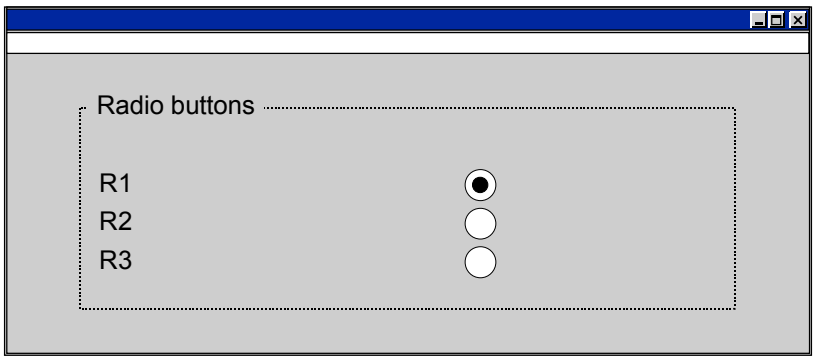
If you use the WITH FRAME addition, a frame is drawn around the block. You can nest up to five different blocks with frames.

If you use TITLE, you can assign a title to each block frame. <title> can be a text symbol or a field name with a maximum length of eight characters. This character field must not be declared with the DATA statement, but is generated automatically with the length of the frame width. The frame width is set automatically based on the nesting depth. The field must be filled before the selection screen is called.

If you use the NO INTERVALS addition, **all selection criteria** of the block are treated as if they had the appropriate addition for restricting entry to single fields [Page 716]. If the block has a frame, the width of the frame is smaller, and nested blocks automatically adopt the NO INTERVALS addition.

```
REPORT DEMO.

SELECTION-SCREEN BEGIN OF BLOCK RAD1
                 WITH FRAME TITLE TEXT-002.
  PARAMETERS R1 RADIOBUTTON GROUP GR1.
  PARAMETERS R2 RADIOBUTTON GROUP GR1.
  PARAMETERS R3 RADIOBUTTON GROUP GR1.
SELECTION-SCREEN END OF BLOCK RAD1.
```

The following selection screen appears:



The three radio buttons R1, R2, and R3 form a block that has a frame and the title 'Radio buttons' specified in text symbol 002.

Calling Selection Screens

How you call a selection screen depends on whether you are calling a standard selection screen for an executable program (report) or a user-defined selection screen in a program.

Standard Selection Screens [Page 725]

User-Defined Selection Screens [Page 726]

Calling Standard Selection Screens

PBO des Selektionsbilds [Page 743]

The standard selection screen of an executable program consists of

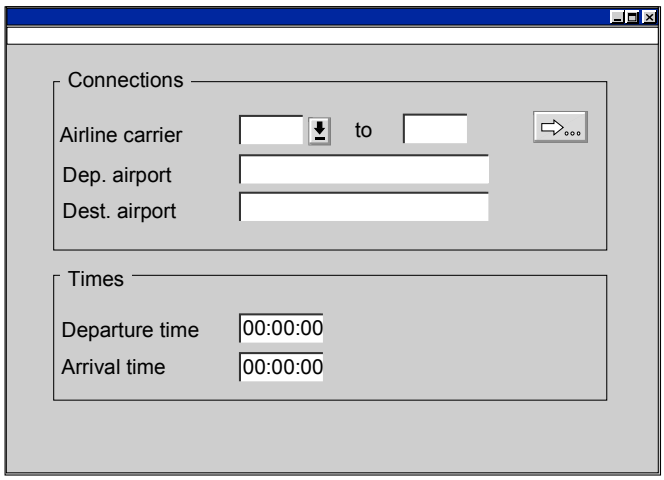
- the selections of a logical database that may be linked to the program
- all selection screen elements from the declaration part of a program that are not assigned to a user-defined selection screen.

As long as at least one input field is defined for the standard selection screen, it is called fully automatically between the INITIALIZATION [Page 954] and START-OF-SELECTION [Page 957] events. During selection screen processing [Page 739], the ABAP runtime environment generates special selection screen events. In the flow of an executable program, these events occur between the INTIALIZATION and the START-OF-SELECTION event. You can define event blocks for these events in the program.

```
REPORT DEMO.

NODES SPFLI.

SELECTION-SCREEN BEGIN OF BLOCK MYSEL WITH FRAME TITLE T01.
  PARAMETERS: DEPTIME LIKE SPFLI-DEPTIME,
              ARRTIME LIKE SPFLI-ARRTIME.
SELECTION-SCREEN END OF BLOCK MYSEL.

INITIALIZATION.
  T01 = 'Times'.
```

...

If the executable program is linked to logical database F1S, the following selection screen is displayed automatically when the program is started:



The top three blocks are defined in the logical database. The bottom block is defined in the program itself.

Calling User-Defined Selection Screens

You can create user-defined selection screens in executable programs (reports), function modules and module pools. There are three ways how user-defined selection screens can be called.

Call From a Program

From any program in which selection screens are defined, you can call these screens at any point of the program flow using the following statement:

CALL SELECTION-SCREEN <numb> [STARTING AT <x_1> <y_1>]
[ENDING AT <x_2> <y_2>].

This statement calls selection screen number <numb>. The selection screen called must be defined in the calling program either as the standard selection screen (screen number 1000) or as a user-defined selection screen (any screen number). You must always use CALL SELECTION-SCREEN to call selection screens, and not CALL SCREEN. If you use CALL SCREEN, the system will not be able to process the selection screen.

You can display a user-defined selection screen as a modal dialog box using the STARTING AT and ENDING AT additions. This is possible even if you have not used the AS WINDOW addition in the definition of the selection screen. However, you are recommended to do so, since warnings and error messages that occur during selection screen processing [Page 739] will then also be displayed as modal dialog boxes (see example below).

When it returns from the selection screen to the program, the CALL SELECTION-SCREEN statement sets the return value SY-SUBRC as follows:

- SY-SUBRC = 0 if the user has chosen *Execute* on the selection screen
- SY-SUBRC = 4 if the user has chosen *Cancel* on the selection screen

Any time a selection screen is processed [Page 739], the selection screen events are triggered. System field SY-DYNNR of the associated event blocks contains the number of the selection screen that is currently active.

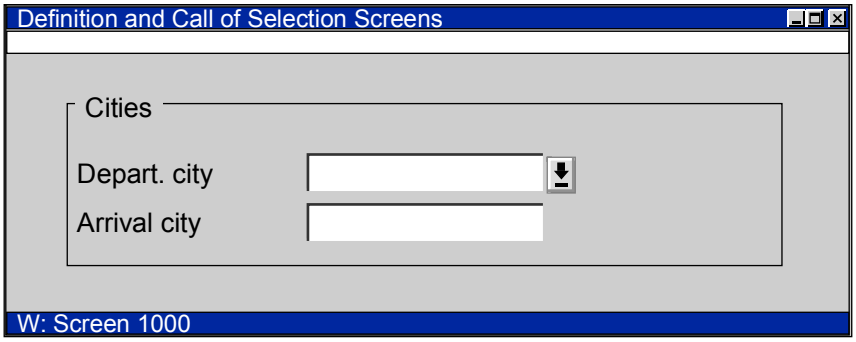
```
REPORT SELSCREENDEF.

SELECTION-SCREEN BEGIN OF BLOCK SEL1 WITH FRAME TITLE TIT1.
  PARAMETERS: CITYFR LIKE SPFLI-CITYFROM,
              CITYTO LIKE SPFLI-CITYTO.
SELECTION-SCREEN END OF BLOCK SEL1.

SELECTION-SCREEN BEGIN OF SCREEN 500 AS WINDOW.
  SELECTION-SCREEN INCLUDE BLOCKS SEL1.
  SELECTION-SCREEN BEGIN OF BLOCK SEL2
                   WITH FRAME TITLE TIT2.
    PARAMETERS: AIRPFR LIKE SPFLI-AIRPFROM,
                AIRPTO LIKE SPFLI-AIRPTO.
  SELECTION-SCREEN END OF BLOCK SEL2.
SELECTION-SCREEN END OF SCREEN 500.

INITIALIZATION.
  TIT1 = 'Cities'.
```

```
AT SELECTION-SCREEN.
  CASE SY-DYNNR.
    WHEN '0500'.
      MESSAGE W159(AT) WITH 'Screen 500'.
    WHEN '1000'.
      MESSAGE W159(AT) WITH 'Screen 1000'.
  ENDCASE.

START-OF-SELECTION.
  TIT1 = 'Cities for Airports'.
  TIT2 = 'Airports'.
  CALL SELECTION-SCREEN 500 STARTING AT 10 10.
  TIT1 = 'Cities again'.
  CALL SELECTION-SCREEN 1000 STARTING AT 10 10.
```

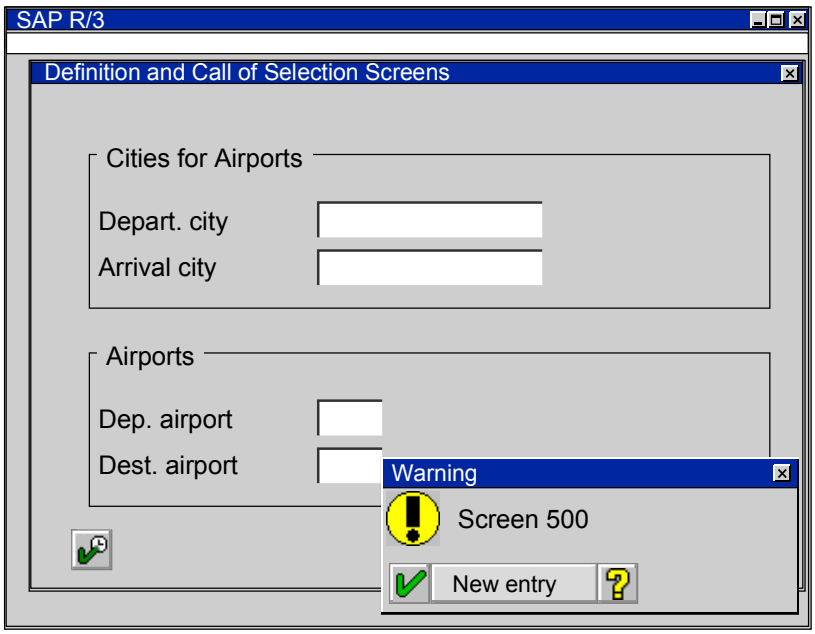
This executable program contains definitions for the standard selection screen and the user-defined screen number 500. Selection screen 500 is defined to be displayed as a modal dialog box and contains the SEL1 block of the standard selection screen. Note the phase in which the titles of the screens are defined. For the purpose of demonstration, the program calls warning messages with message class AT during the AT SELECTION-SCREEN event.

When you start the program, the following sequence of screens is displayed:

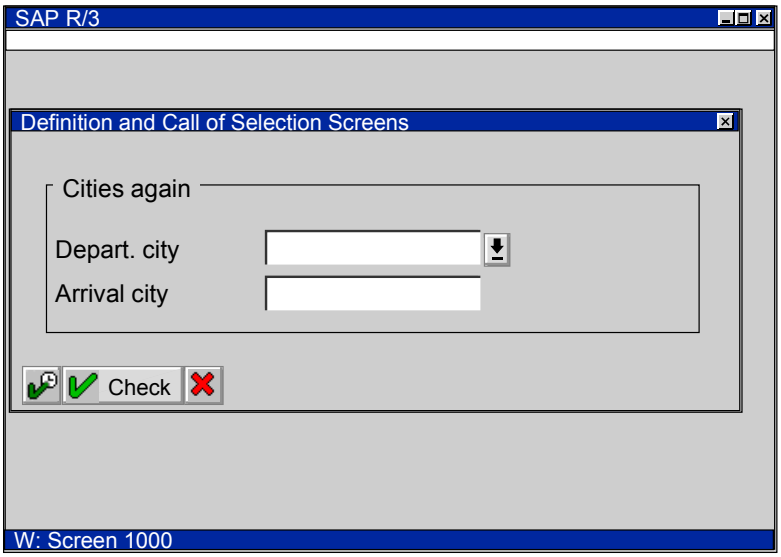
1. The standard selection screen. If the user chooses *Execute*, the system displays the warning SCREEN 1000 in the status bar.



2. Once the user has confirmed the warning by choosing *Enter*, selection screen 500 is called as a modal dialog box. When the user chooses *Execute*, the system displays the warning SCREEN 500, also in a dialog box.



3. When the user has confirmed this warning, the standard selection screen is called again, but this time as a modal dialog box. Since you cannot define the standard selection screen as a modal dialog box, the warning message displayed when the user chooses *Execute* appears in the status bar and not as a modal dialog box.

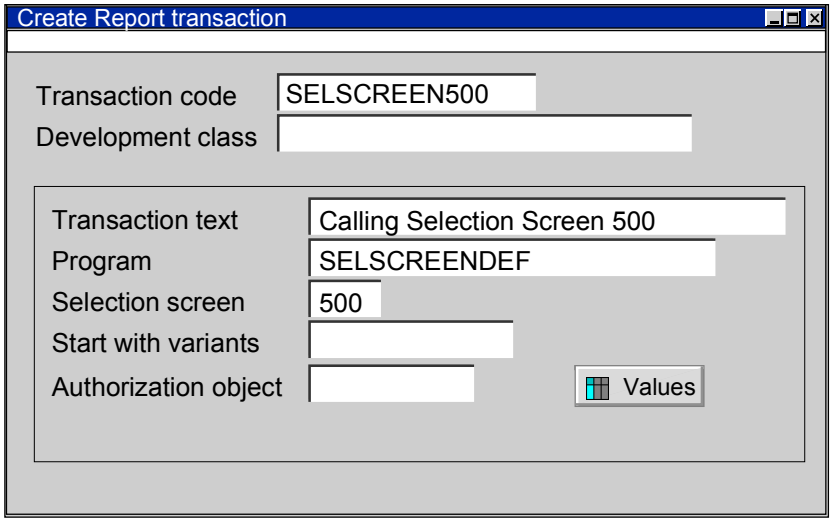


Consequently, you can only exit this selection screen using *Cancel*, since there is no *Enter* function in the dialog box to confirm the warning message.

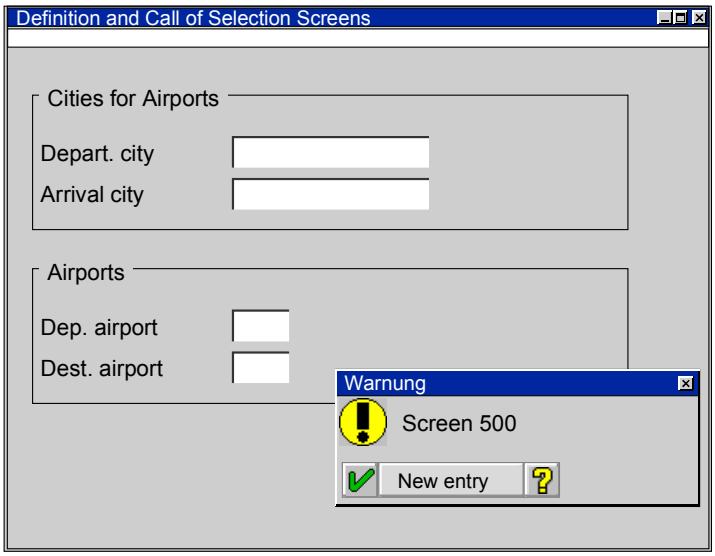
Call as a Report Transaction

When creating transaction codes for report transactions [Page 951], you can define any user-defined selection screen of the executable program for which you create the transaction code as the initial screen. When the program is then started using the transaction code, the user-defined selection screen and not the standard selection screen is called between the INITIALIZATION [Page 954] and the START-OF-SELECTION [Page 957] event.

You create a report transaction with transaction code SELSCREEN500 for the example program above:



When you call the transaction, the executable program is started directly with selection screen 500 as a full screen. When the user chooses *Execute*, the warning is displayed as a modal dialog box because selection screen 500 was originally defined as a modal dialog box.



The sequence of screens then continues in the same way as described from point 2 onwards in the above example.

Call as a Dialog Transaction

You can use any selection screen in executable programs and module pools that are called with transaction codes for dialog transactions [Page 982] as the initial screen. The selection screen is then the first screen of a screen sequence [Page 1000].

You must ensure that the program flow moves on to the correct next screen during selection screen processing [Page 739].

Let us consider the following module pool:

```
*&---------------------------------------------------------*
*& Modulpool         SAPMSSLS                            *
*&---------------------------------------------------------*

INCLUDE MSSLSTOP.

INCLUDE MSSLSEVT.
```

Include MSSLSTOP contains the following definition of selection screen 500:

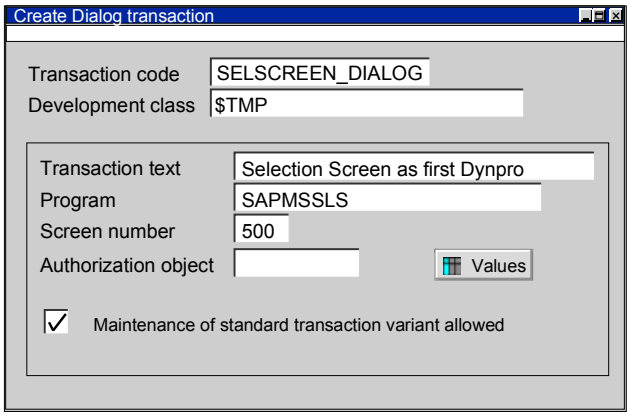
```
*&---------------------------------------------------------*
*& Include MSSLSTOP                                       *
*&---------------------------------------------------------*

PROGRAM  SAPMSSLS.

SELECTION-SCREEN BEGIN OF SCREEN 500 AS WINDOW.
  SELECTION-SCREEN BEGIN OF BLOCK SEL1 WITH FRAME.
    PARAMETERS: CITYFR LIKE SPFLI-CITYFROM,
                CITYTO LIKE SPFLI-CITYTO.
  SELECTION-SCREEN END OF BLOCK SEL1.
```

```
  SELECTION-SCREEN BEGIN OF BLOCK SEL2 WITH FRAME.
    PARAMETERS: AIRPFR LIKE SPFLI-AIRPFROM,
                AIRPTO LIKE SPFLI-AIRPTO.
  SELECTION-SCREEN END OF BLOCK SEL2.
SELECTION-SCREEN END OF SCREEN 500.
```

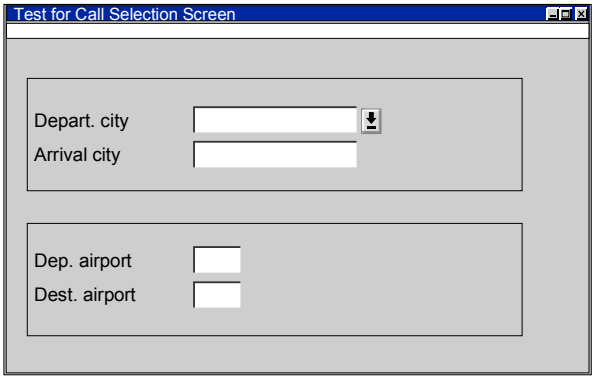
Include MSSLSEVT processes the AT SELECTION-SCREEN event:

```
*---------------------------------------------------------*
*   INCLUDE MSSLSEVT                                      *
*---------------------------------------------------------*

AT SELECTION-SCREEN.

  ...

  LEAVE TO SCREEN 100.
```

When we create a transaction code for program SAPMSSLS, we enter screen 500 as the initial screen.



Transaction SELSCREEN_DIALOG starts the program and displays the selection screen.



You can process the user entries from the selection screen either at the AT SELECTION-SCREEN event, or at a later point in the application logic. When the AT SELECTION-SCREEN event (PAI of the selection screen) has been processed, the program moves on to the next screen 100.

User Actions on Selection Screens

A selection screen is a standardized screen for entering data and selection ranges. It also has a standardized user interface, which you cannot affect in the same way that you can work with the user interfaces of screens and lists.

Selection screens are used primarily for entering data. When the user chooses functions, the corresponding function codes [Page 548] from the GUI status are not processed in the ABAP program in which the selection screen is defined. Instead, they are processed directly by the ABAP runtime environment. Some functions are executed by the ABAP runtime environment itself (for example, saving the selection screen contents as a variant, or diverting the list output from the screen to the spool system). Other functions, of which Enter and Execute (F8)are the most important, trigger the selection screen processing [Page 739] in the ABAP program. Here, special event blocks are called in the ABAP program. The triggering function from the ABAP program is not normally necessary.

The following sections describe the ways in which you can change the standard process:

Pushbuttons on the Selection Screen [Page 733]

Checkboxes and Radio Buttons with Function Codes [Page 735]

Pushbuttons in the Application Toolbar [Page 736]

Changing the Standard GUI Status [Page 738]

Pushbuttons on the Selection Screen

Drucktasten auf Dynpros [Page 542]

To create a pushbutton on the selection screen, you use:

```
SELECTION SCREEN PUSHBUTTON [/]<pos(len)> <push>
                  USER-COMMAND <ucom> [MODIF ID <key>].
```

The [/]<pos(len)> parameters and the MODIF IF addition have the same function as for the formatting options for underlines and comments [Page 719].

<push> determines the pushbutton text. For <push>, you can specify a text symbol [Page 121] or a field name with a maximum length of eight characters. This character field must not be declared with the DATA statement, but is generated automatically with length <len>. The field must be filled before the selection screen is called.

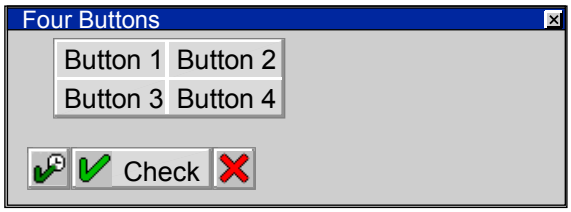
For <ucom>, you must specify a code of up to four characters. When the user clicks the pushbutton on the selection screen, <ucom> is entered in the UCOMM of the SSCRFIELDS interface work area [Page 130]. You must use the TABLES statement to declare the SSCRFIELDS structure. The contents of the SSCRFIELDS-UCOMM field can be processed during the AT SELECTION-SCREEN [Page 956] event.

```
REPORT DEMO.

TABLES SSCRFIELDS.

DATA FLAG.

SELECTION-SCREEN:
  BEGIN OF SCREEN 500 AS WINDOW TITLE TIT,
    BEGIN OF LINE,
      PUSHBUTTON 2(10) BUT1 USER-COMMAND CLI1,
      PUSHBUTTON 12(10) TEXT-020 USER-COMMAND CLI2,
    END OF LINE,
    BEGIN OF LINE,
      PUSHBUTTON 2(10) BUT3 USER-COMMAND CLI3,
      PUSHBUTTON 12(10) TEXT-040 USER-COMMAND CLI4,
    END OF LINE,
  END OF SCREEN 500.

AT SELECTION-SCREEN.

  CASE SSCRFIELDS.
    WHEN 'CLI1'.
      FLAG = '1'.
    WHEN 'CLI2'.
      FLAG = '2'.
    WHEN 'CLI3'.
      FLAG = '3'.
    WHEN 'CLI4'.
      FLAG = '4'.
  ENDCASE.

START-OF-SELECTION.
```

```
TIT  = 'Four Buttons'.
BUT1 = 'Button 1'.
BUT3 = 'Button 3'.

CALL SELECTION-SCREEN 500 STARTING AT 10 10.

CASE FLAG.
  WHEN '1'.
    WRITE / 'Button 1 was clicked'.
  WHEN '2'.
    WRITE / 'Button 2 was clicked'.
  WHEN '3'.
    WRITE / 'Button 3 was clicked'.
  WHEN '4'.
    WRITE / 'Button 4 was clicked'.
  WHEN OTHERS.
    WRITE / 'No Button was clicked'.
ENDCASE.
```

This example defines four pushbuttons on a selection screen that is displayed as a dialog box. The selection screen is defined in a statement chain for keyword SELECTION-SCREEN.

If the text symbols TEXT-020 and TEXT-040 are defined as 'Button 2' and 'Button 4', the four pushbuttons appear as follows on the selection screen displayed as a dialog box.



CLI1, CLI2, CLI3 and CLI4 are used for <ucom>. When the user clicks one of the pushbuttons, the AT SELECTION-SCREEN event is triggered, and the FLAG field is set. The FLAG field can be further processed during subsequent program flow after the user has chosen *Execute*.

Checkboxes and Radio Buttons with Function Codes

Similarly to on screens, where you can define checkboxes and radio buttons [Page 545] with function codes, you can do the same with checkboxes and radio buttons on selection screens. You do this using the USER-COMMAND addition when you declare the relevant parameters:

PARAMETERS ... AS CHECKBOX | RADIOBUTTON GROUP ... USER-COMMAND <ucom>.

You can assign a function code <ucom> to an individual checkbox. However, a radio button group must have one shared function code, since it is only possible to make the assignment for the first button in the group.

When you select a checkbox or radio button in a group, the runtime analysis triggers the AT SELECTION-SCREEN [Page 956] event and places the function code <ucom> into component UCOMM of the interface work area [Page 130] SSCRFIELDS. You must use the TABLES statement to declare the SSCRFIELDS structure.

After the AT SELECTION-SCREEN event has been processed, the system displays the selection screen again. The only way to exit the selection screen and carry on processing the program is to choose *Execute* (F8). Consequently, checkboxes and radio buttons with function codes are more suitable for controlling dynamic modifications on a selection screen than for controlling the program flow.

```
REPORT demo_sel_screen_user_command.

TABLES sscrfields.

PARAMETERS: rad1 RADIOBUTTON GROUP rad USER-COMMAND radio,
            rad2 RADIOBUTTON GROUP rad,
            rad3 RADIOBUTTON GROUP rad.

PARAMETERS  check AS CHECKBOX USER-COMMAND check.

AT SELECTION-SCREEN.
  MESSAGE i888(sabapdocu) WITH text-001 sscrfields-ucomm.

START-OF-SELECTION.
  WRITE text-002.
```

This program assigns function codes to a radio button group and a checkbox.

Pushbuttons in the Application Toolbar

You can create up to five pushbuttons in the application toolbar on the selection screen. Function keys are automatically assigned to these pushbuttons.

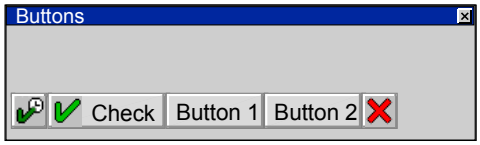
SELECTION-SCREEN FUNCTION KEY <i>.

<i> must be between 1 and 5. You must use the FUNCTXT_0<i> components of structure SSCRFIELDS for the individual pushbutton texts. You must declare this structure as an interface work area [Page 130] using the TABLES statement.

If the user clicks a pushbutton, function code FC0<i> of component UCOMM is assigned to structure SSCRFIELDS. The function code can be analyzed during the AT SELECTION-SCREEN [Page 956] event.

```
REPORT DEMO.

TABLES SSCRFIELDS.

DATA FLAG.

SELECTION-SCREEN BEGIN OF SCREEN 500 AS WINDOW TITLE TIT.
  SELECTION-SCREEN FUNCTION KEY 1.
  SELECTION-SCREEN FUNCTION KEY 2.
SELECTION-SCREEN END OF SCREEN 500.

AT SELECTION-SCREEN.
  CASE SY-DYNNR.
    WHEN '0500'.
      IF SSCRFIELDS-UCOMM = 'FC01'.
        FLAG = '1'.
      ELSEIF SSCRFIELDS-UCOMM = 'FC02'.
        FLAG = '2'.
      ENDIF.
  ENDCASE.

START-OF-SELECTION.

  SSCRFIELDS-FUNCTXT_01 = 'Button 1'.
  SSCRFIELDS-FUNCTXT_02 = 'Button 2'.
  TIT = 'Buttons'.

  CALL SELECTION-SCREEN 500 STARTING AT 10 10
                            ENDING   AT 10 11.

  IF FLAG = '1'.
    WRITE / 'Button 1 was clicked'.
  ELSEIF FLAG = '2'.
    WRITE / 'Button 2 was clicked'.
  ENDIF.
```

This example program defines a user-defined selection screen as a modal dialog box, containing two pushbuttons in the application toolbar with the texts ‘Button 1’ and ‘Button 2’.



When the user clicks one of the buttons, the AT SELECTION-SCREEN event is triggered, and the FLAG field is set there. The contents of the FLAG field can be further processed during subsequent program flow after the user has chosen *Execute*.

Changing the Standard GUI Status

The GUI status of a selection screen is generated by the system. The SET PF-STATUS statement in the PBO event of the selection screen has no effect on the standard GUI status. If you want to use your own GUI status for a selection screen or deactivate functions in the standard GUI status in exceptional cases, you can use one of the following function modules in the PBO event of the selection screen:

- RS_SET_SELSCREEN_STATUS

  Sets another GUI status defined in the same ABAP program, or deactivates functions of the standard GUI status.

- RS_EXTERNAL_SELSCREEN_STATUS

  Sets a GUI status defined in an external function group. You must use the SET PF-STATUS statement to set the status in a special function module in this function group. You must pass the name of the function module that sets the status as a parameter to the function module RS_EXTERNAL_SELSCREEN_STATUS.

For further information, refer to the function module documentation.

```
REPORT demo_sel_screen_status.

DATA itab TYPE TABLE OF sy-ucomm.

PARAMETERS test(10) TYPE c.

AT SELECTION-SCREEN OUTPUT.
  APPEND: 'PRIN' TO itab,
          'SPOS' TO itab.
  CALL FUNCTION 'RS_SET_SELSCREEN_STATUS'
       EXPORTING
            p_status  = sy-pfkey
       TABLES
            p_exclude = itab.
```

In this example, the *Print* and *Save as variant* functions are deactivated. To find out the function codes of the standard GUI status, choose *System* → *Status* and double-click the *GUI status* field.

Selection Screen Processing

Selection screens are special screens that are defined with the help of ABAP statements. As programmers do not have access to the flow logic of selection screens, they cannot define dialog modules for selection screens. The ABAP runtime environment fully controls the processing flow of selection screens. To allow programmers to modify the selection screen before it is called (PBO) and react to user actions on the selection screen (PAI), the ABAP runtime environment generates a number of special selection screen events before the selection screen is displayed and after the user has executed actions on the selection screen. Programmers can define event blocks [Page 941] in the program to react to these events.

The basic form of the selection screen events is the AT SELECTION-SCREEN event. This event occurs after the runtime environment has passed all input data from the selection screen to the ABAP program. The other selection screen events allow programmers to modify the selection screen before it is sent and specifically check user input.

Selection screen events occur both during standard and user-defined selection screen processing. The SY-DYNNR system field contains the number of the active selection screen and helps you to determine which selection screen is currently being processed in the event blocks.

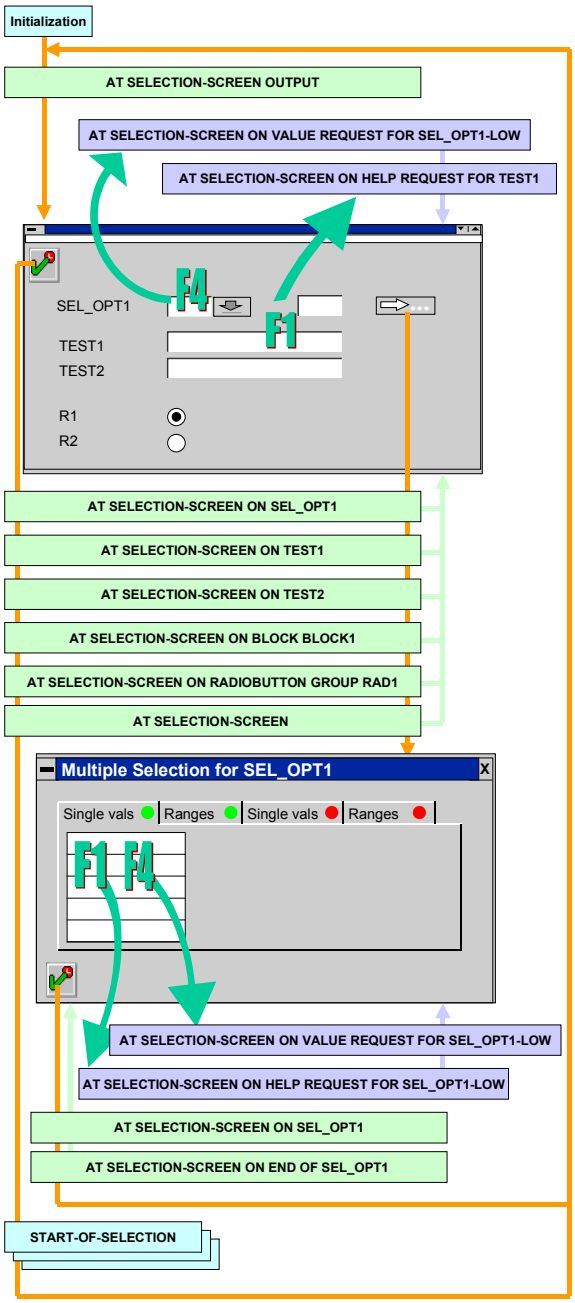
Overview of Selection Screen Events

For example, a standard selection screen is defined as follows in the declaration part of an executable program:

```
DATA FIELD1(10).

SELECT-OPTIONS SEL_OPT1 FOR FIELD1.

SELECTION-SCREEN BEGIN OF BLOCK BLOCK1.
  PARAMETERS:  TEST1(10),
               TEST2(10).
SELECTION-SCREEN END OF BLOCK BLOCK1.

PARAMETERS: R1 RADIOBUTTON GROUP RAD1 DEFAULT 'X',
            R2 RADIOBUTTON GROUP RAD1.
```

The following graphic shows the possible selection screen events and their chronological sequence between the INITIALIZATION [Page 954] and the START-OF-SELECTION [Page 957] event. If user-defined selection screens are called, the event sequence is imbedded accordingly in the current program flow.

Selection Screen Processing



Selection screen processing starts after the INITIALIZATION event with AT SELECTION SCREEN OUTPUT. The selection screen is then sent to the screen. User actions on the selection screen result in other events that are either used for field or possible entries help, or that trigger PAI processing of the selection screen. During PAI processing, error messages [Page 934] in the relevant event blocks allow users to return to the selection screen. Only if the AT SELECTION-SCREEN event is exited properly, that is not through an error message, are the other events of the executable program triggered, starting with START-OF-SELECTION.

Choosing multiple selection calls the relevant dialog box. Similarly, events are triggered during its PAI processing.

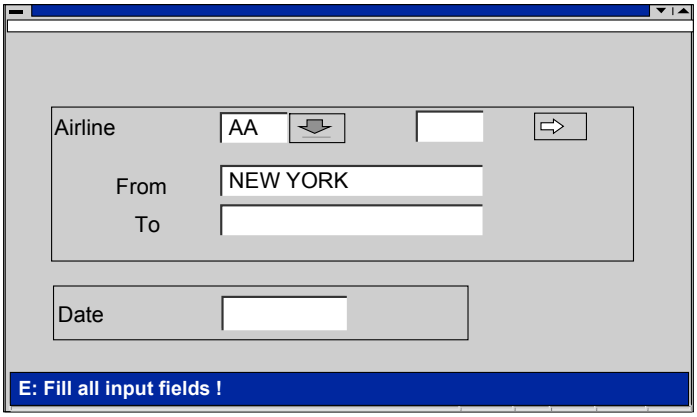
Basic Form

The AT SELECTION-SCREEN event is triggered in the PAI of the selection screen once the ABAP runtime environment has passed all of the input data from the selection screen to the ABAP program. If an error message occurs in this processing block, the selection screen is redisplayed with all of its fields ready for input. This allows you to check input values for consistency.

The program below is connected to the logical database F1S:

```
REPORT EVENT_DEMO.

NODES SPFLI.

AT SELECTION-SCREEN.
  IF CARRID-LOW IS INITIAL
    OR CITY_FR IS INITIAL
    OR CITY_TO IS INITIAL.
    MESSAGE E000(HB).
  ENDIF.
```

If the user does not enter values in all of the fields on the selection screen, an error message appears in the status line. This makes all of the input fields mandatory, even though they are not defined as such in the logical database.



PBO of the Selection Screen

INITIALIZATION [Page 954]

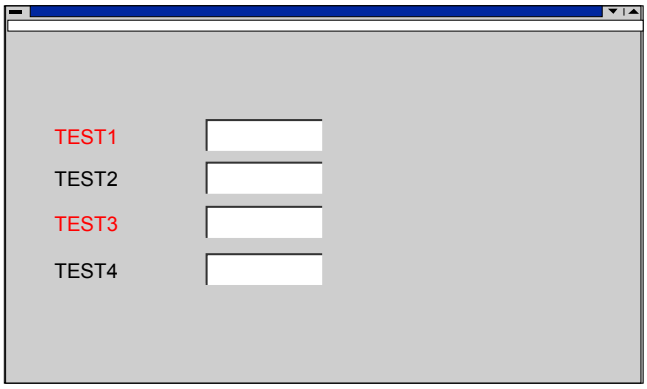
In the PBO of the selection screen, the

AT SELECTION-SCREEN OUTPUT

event is triggered. This event block allows you to modify the selection screen directly before it is displayed.

```
PARAMETERS: TEST1(10) MODIF ID SC1,
            TEST2(10) MODIF ID SC2,
            TEST3(10) MODIF ID SC1,
            TEST4(10) MODIF ID SC2.

AT SELECTION-SCREEN OUTPUT.

LOOP AT SCREEN.
  IF SCREEN-GROUP1 = 'SC1'.
    SCREEN-INTENSIFIED = '1'.
    MODIFY SCREEN.
    CONTINUE.
  ENDIF.
  IF SCREEN-GROUP1 = 'SC2'.
    SCREEN-INTENSIFIED = '0'.
    MODIFY SCREEN.
  ENDIF.
ENDLOOP.
```

The parameters TEST1 and TEST3 are assigned to the modification group SC1, while TEST2 and TEST4 are assigned to group SC2. During the AT SELECTION-SCREEN OUTPUT event, the INTENSIFIED field of internal table SCREEN is set to 1 or 0, depending on the contents of the GROUP1 field. On the selection screen, the lines for TEST1 and TEST3 are highlighted while those for TEST2 and TEST4 are not, as shown below:



Processing Single Fields

In the PAI event of the selection screen, the event

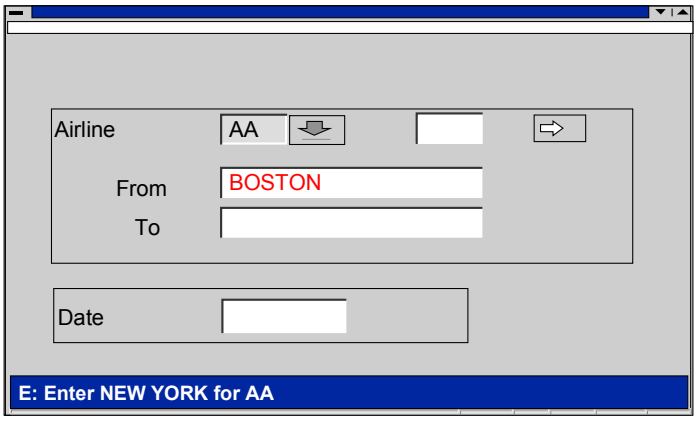
AT SELECTION-SCREEN ON <field>

is triggered when the contents of each individual input field are passed from the selection screen to the ABAP program. The input field <field> can be checked in the corresponding event block. If an error message occurs within this event block, the corresponding field is made ready for input again on the selection screen.

The program below is connected to the logical database F1S:

```
REPORT EVENT_DEMO.

NODES SPFLI.

AT SELECTION-SCREEN ON CITY_FR.
  IF CARRID-LOW EQ 'AA' AND CITY_FR NE 'NEW YORK'.
    MESSAGE E010(HB).
  ENDIF.
```

If the user enters “AA” in the first input field, but not NEW YORK for the departure city, an error message is displayed in the status line until the user enters the correct city.



Processing Blocks

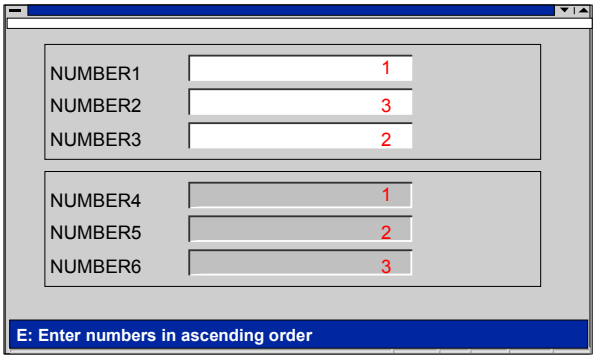
In the PAI event of the selection screen, the event

AT SELECTION-SCREEN ON BLOCK <block>

is triggered when the contents of all of the fields in a block are passed from the selection screen to the ABAP program. You define a block by enclosing the declarations of the elements in the block between the statements SELECTION-SCREEN BEGIN OF BLOCK <block> and SELECTION-SCREEN END OF BLOCK <block>. You can use this event block to check the consistency of the input fields in the block. If an error message occurs within this event block, the fields in the block are made ready for input again on the selection screen.

```
REPORT EVENT_DEMO.
SELECTION-SCREEN BEGIN OF BLOCK PART1 WITH FRAME.
  PARAMETERS: NUMBER1 TYPE I,
              NUMBER2 TYPE I,
              NUMBER3 TYPE I.
SELECTION-SCREEN END OF BLOCK PART1.
SELECTION-SCREEN BEGIN OF BLOCK PART2 WITH FRAME.
  PARAMETERS: NUMBER4 TYPE I,
              NUMBER5 TYPE I,
              NUMBER6 TYPE I.
SELECTION-SCREEN END OF BLOCK PART2.
AT SELECTION-SCREEN ON BLOCK PART1.
  IF NUMBER3 LT NUMBER2 OR
     NUMBER3 LT NUMBER1 OR
     NUMBER2 LT NUMBER1.
     MESSAGE E020(HB).
  ENDIF.
AT SELECTION-SCREEN ON BLOCK PART2.
  IF NUMBER6 LT NUMBER5 OR
     NUMBER6 LT NUMBER4 OR
     NUMBER5 LT NUMBER4.
     MESSAGE E030(HB).
  ENDIF.
```

If the user does not enter numbers in ascending order in one of the blocks, the whole of the corresponding block is made ready for input again.



Processing Radio Buttons

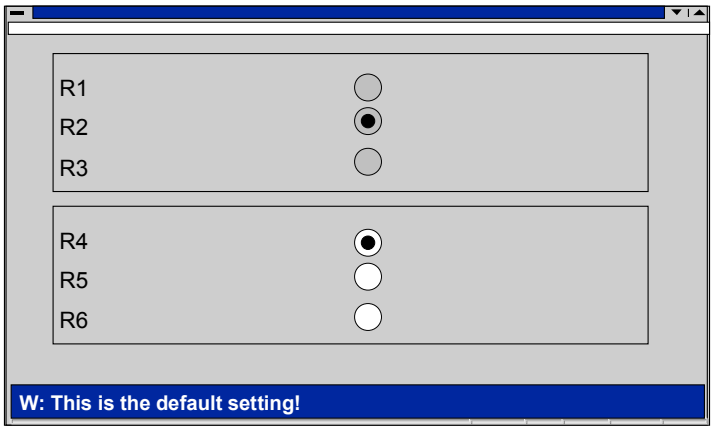
In the PAI event of the selection screen, the event

AT SELECTION-SCREEN ON RADIOBUTTON GROUP <radi>

is triggered when the contents of all of the fields in a radio button group are passed from the selection screen to the ABAP program. To define a radio button group <radi>, use the addition RADIOBUTTON GROUP <radi> in the corresponding PARAMETERS statements. This event block allows you to check the whole group. If an error message occurs within this event block, the radio button group is made ready for input again on the selection screen. The individual fields of radio button groups do not trigger the event AT SELECTION-SCREEN ON <field>.

```
REPORT EVENT_DEMO.
SELECTION-SCREEN BEGIN OF BLOCK B1 WITH FRAME.
  PARAMETERS: R1 RADIOBUTTON GROUP RAD1 DEFAULT 'X',
              R2 RADIOBUTTON GROUP RAD1,
              R3 RADIOBUTTON GROUP RAD1.
SELECTION-SCREEN END OF BLOCK B1.
SELECTION-SCREEN BEGIN OF BLOCK B2 WITH FRAME.
  PARAMETERS: R4 RADIOBUTTON GROUP RAD2 DEFAULT 'X',
              R5 RADIOBUTTON GROUP RAD2,
              R6 RADIOBUTTON GROUP RAD2.
SELECTION-SCREEN END OF BLOCK B2.
AT SELECTION-SCREEN ON RADIOBUTTON GROUP RAD1.
  IF R1 = 'X'.
    MESSAGE W040(HB).
  ENDIF.
AT SELECTION-SCREEN ON RADIOBUTTON GROUP RAD2.
  IF R4 = 'X'.
    MESSAGE W040(HB).
  ENDIF.
```

If the user does not change one of the radio button groups, a warning is displayed.



Processing Multiple Selections

If the user opens the Multiple selections dialog box for a selection option, the same events are triggered in the PAI of the selection screen as if the user had chosen *Execute*. The user can then enter the required multiple selections. In the *Multiple selections* dialog box, user actions either lead to input help or trigger the PAI event of the dialog box. At first, the

AT SELECTION-SCREEN ON <seltab>

event is triggered for the current line of the selection table. It can then be processed like a single field [Page 744].

Next, the

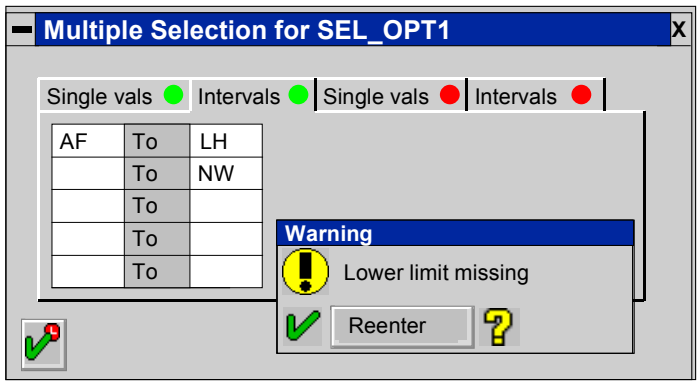
AT SELECTION-SCREEN ON END OF <seltab>

event is triggered. This event block allows you to check the whole selection table <seltab>. Warning messages are displayed as dialog boxes, not in the status line.

The program below is connected to the logical database F1S:

```
REPORT EVENT_DEMO.

NODES SPFLI.

AT SELECTION-SCREEN ON END OF CARRID.
  LOOP AT CARRID.
    IF CARRID-HIGH NE '    '.
      IF CARRID-LOW IS INITIAL.
        MESSAGE W050(HB).
      ENDIF.
    ENDIF.
  ENDLOOP.
```

If the user chooses the multiple selection button () on the selection screen, and then enters upper limits without lower limits in an interval field, a dialog box appears with a warning:



Defining Field Help

If the data type of an input field declared in an executable program is defined in the ABAP Dictionary, the documentation of the underlying data element is automatically displayed if the user positions the cursor in that field and presses F1. To create help for input fields that have no Dictionary reference, or to override the help normally linked to the field, you can create an event block for the event

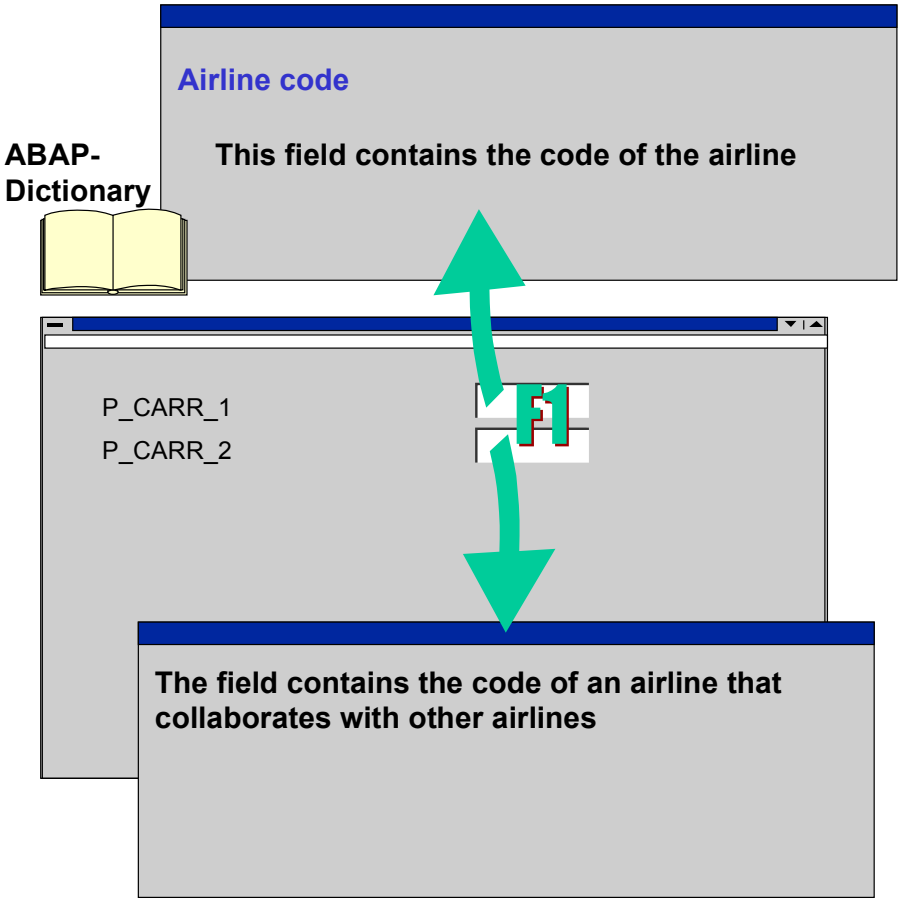
AT SELECTION-SCREEN ON HELP-REQUEST FOR <field>

The event is triggered when the user calls the F1 help for the field <field>. If no corresponding event block has been defined, the help from the ABAP Dictionary is displayed, or none at all if the field has no Dictionary reference. If a corresponding event block exists, it takes precedence over the default help mechanism. It is then up to the programmer to ensure that appropriate help is displayed.

You cannot declare the event block AT SELECTION-SCREEN ON HELP-REQUEST for input fields on the selection screen that are declared within a logical database. You cannot override the help mechanism of a logical database within the program. You can define separate help within the logical database program using the HELP-REQUEST option in the PARAMETERS and SELECT-OPTIONS statements.

```
REPORT SELECTION_SCREEN_F1_DEMO.

PARAMETERS: P_CARR_1 TYPE S_CARR_ID,
            P_CARR_2 TYPE S_CARR_ID.

AT SELECTION-SCREEN ON HELP-REQUEST FOR P_CARR_2.

  CALL SCREEN 100 STARTING AT 10 5
                  ENDING   AT 60 10.
```

This program declares a selection screen with two parameters that both refer to the data element S_CARR_ID in the ABAP Dictionary. The documentation from the ABAP Dictionary is used for P_CARR_1, and a help screen 100 is called for P_CARR_2. The help screen is defined in the Screen Painter as a modal dialog box with next screen 0. It contains the help text defined as help texts. The screen does not require any flow logic.


Airline code
This field contains the code of the airline
ABAP-
Dictionary
P_CARR_1
P_CARR_2
F1
The field contains the code of an airline that
collaborates with other airlines

Defining Input Help

Suchhilfe für Parameter [Page 697]

If a field in an executable program is defined with reference to an ABAP Dictionary field for which possible entries help is defined, the values from the ABAP Dictionary help are automatically displayed when the user calls the F4 help for that field. To create possible values help for input fields that have no Dictionary reference, or to override the help normally linked to the field, you can create an event block for the event

AT SELECTION-SCREEN ON VALUE-REQUEST FOR <field>

The event is triggered when the user calls the F4 help for the field <field>. If no corresponding event block has been defined, the possible values help from the ABAP Dictionary is displayed, or none at all if the field has no Dictionary reference. If a corresponding event block exists, it takes precedence over the default possible values help mechanism. It is then up to the programmer to ensure that an appropriate list of values is displayed, and that the user can choose a value from it.

You cannot declare the event block AT SELECTION-SCREEN ON VALUE-REQUEST for input fields on the selection screen that are declared within a logical database. You cannot override the possible values help mechanism of a logical database within the program. You can define separate help within the logical database program using the VALUE-REQUEST option in the PARAMETERS and SELECT-OPTIONS statements.

```
REPORT SELECTION_SCREEN_F4_DEMO.

PARAMETERS: P_CARR_1 TYPE SPFLI-CARRID,
            P_CARR_2 TYPE SPFLI-CARRID.

AT SELECTION-SCREEN ON VALUE-REQUEST FOR P_CARR_2.

  CALL SCREEN 100 STARTING AT 10 5
                  ENDING   AT 50 10.

MODULE VALUE_LIST OUTPUT.

  SUPPRESS DIALOG.
  LEAVE TO LIST-PROCESSING AND RETURN TO SCREEN 0.
  SET PF-STATUS SPACE.
  NEW-PAGE NO-TITLE.

  WRITE 'Star Alliance' COLOR COL_HEADING.
  ULINE.

  P_CARR_2 = 'AC '.
  WRITE: / P_CARR_2 COLOR COL_KEY, 'Air Canada'.
  HIDE P_CARR_2.

  P_CARR_2 = 'LH '.
  WRITE: / P_CARR_2 COLOR COL_KEY, 'Lufthansa'.
  HIDE P_CARR_2.

  P_CARR_2 = 'SAS'.
  WRITE: / P_CARR_2 COLOR COL_KEY, 'SAS'.
  HIDE P_CARR_2.
```

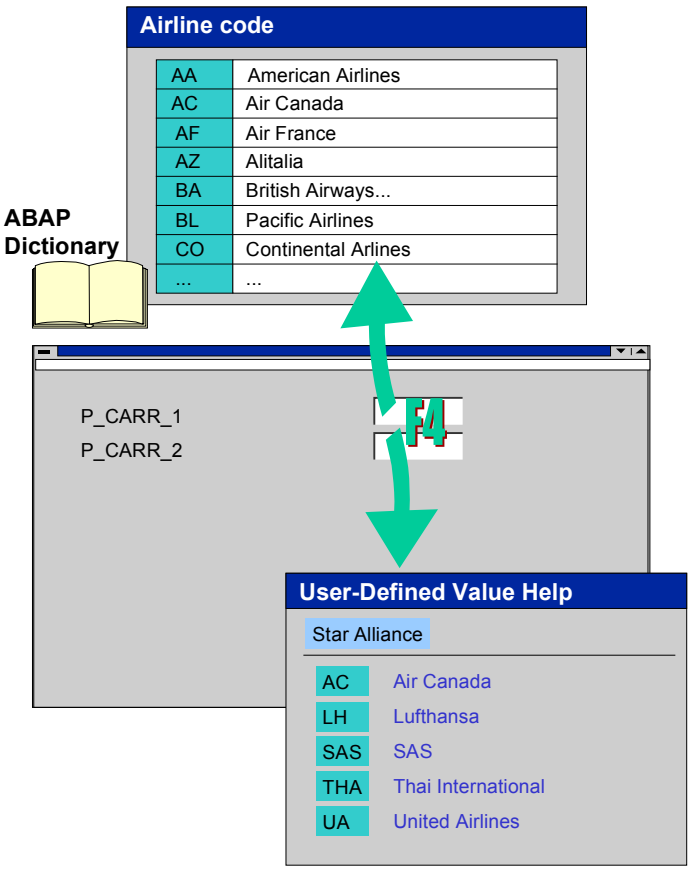
```
  P_CARR_2 = 'THA'.
4  WRITE: / P_CARR_2 COLOR COL_KEY, 'Thai International'.
  HIDE P_CARR_2.

  P_CARR_2 = 'UA '.
  WRITE: / P_CARR_2 COLOR COL_KEY, 'United Airlines'.
  HIDE P_CARR_2.

  CLEAR P_CARR_2.

ENDMODULE.

AT LINE-SELECTION.

  CHECK NOT P_CARR_2 IS INITIAL.
  LEAVE TO SCREEN 0.
```

This program defines a selection screen with two parameters, both of which refer to the column CARRID in the database table SPFLI. The possible entries help from the ABAP Dictionary is used for P_CARR_1, and a separate possible entries help is programmed for P_CARR_2. Screen 100 is used for the possible entries help. The dialog module VALUE_LIST is started in its PBO event. The actual screen mask is not used, and there are no dialog modules used in the PAI.

```
PROCESS BEFORE OUTPUT.
  MODULE VALUE_LIST.

PROCESS AFTER INPUT.
```

The dialog module VALUE_LIST suppresses the dialog of screen 100 and switches to list processing. The list contains values for the parameter P_CARR_2. These values are also placed in the HIDE area. When the user selects a line from the value list, the AT LINE-SELECTION event is triggered, and the selected value is transferred from the HIDE area into the field P_CARR_2. If the user selects a valid line, the system switches directly from the event block AT LINE-SELECTION back to the selection screen, and fills the corresponding input field.

Defining Input Help



Subscreens and Tabstrip Controls on Selection Screens

Some of the complex screen elements [Page 635] that you can create in the Screen Painter for screens can also be defined in ABAP programs for selection screens. For example, you can now use tabstrip controls on selection screens. Since you need subscreens to work with tabstrip controls, you can also now define selection screens as subscreens.

Selection Screens as Subscreens [Page 754]

Tabstrip Controls on Selection Screens [Page 758]

Subscreens on Selection Screens [Page 762]

Selection Screens as Subscreens

In the same way that you can define a screen as a subscreen [Page 647] in the Screen Painter, it is now possible to define selection screens as subscreens in an ABAP program:

```
SELECTION-SCREEN BEGIN OF SCREEN <scrn> AS SUBSCREEN
                 [NO INTERVALS]
                 [NESTING LEVEL <n>].
  ...
SELECTION-SCREEN END OF SCREEN <scrn>.
```

Selection screens that you define in this way can be included in:

- Subscreens [Page 647] on screens
- Tabstrip Controls [Page 653] on screens
- Tabstrip Controls on Selection Screens [Page 758]

You **cannot** call them using CALL SELECTION-SCREEN.

If you use the NO INTERVALS addition, the subscreen is made smaller and the selection criteria are all displayed with single input fields.

The NESTING LEVEL also reduces the size of the subscreen. You can use it to prevent scrollbars from appearing when you use the subscreen in a tabstrip control on the selection screen and the tabstrip already has a frame. If there is no frame around the tabstrip control, use NESTING LEVEL 0. For each frame around the tabstrip control, increase NESTING LEVEL by one.

When you include a selection screen as a subscreen on a screen or in a tabstrip control on a screen, you should remember that the CALL SUBSCREEN statement is executed in both the PBO and PAI events in the screen flow logic. Although you cannot program PAI modules for selection screens as subscreens, the CALL SUBSCREEN statement ensures that the input data is transferred to the ABAP program in the PAI event.

As on normal selection screens, the usual permitted user actions [Page 732] trigger the usual selection screen processing [Page 739]. This allows you to check user input or process function codes.

Examples

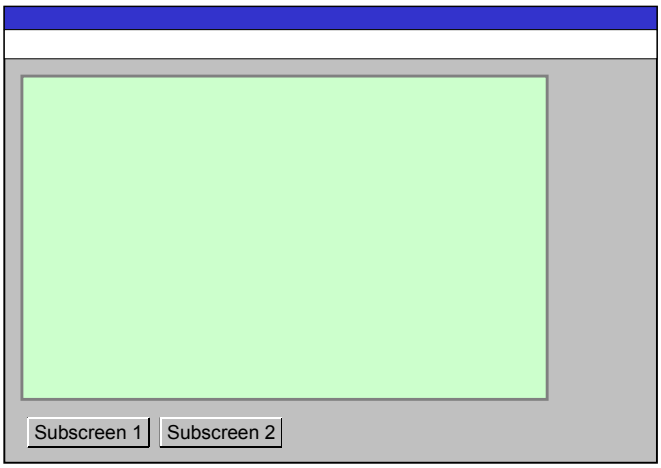
Selection screens as subscreens on screens.

```
REPORT demo_sel_screen_as_subscreen.

SELECTION-SCREEN BEGIN OF SCREEN 1100 AS SUBSCREEN.
SELECTION-SCREEN BEGIN OF BLOCK b1 WITH FRAME TITLE text-010.
PARAMETERS: p1(10) TYPE c,
            p2(10) TYPE c,
            p3(10) TYPE c.
SELECTION-SCREEN END OF BLOCK b1.
SELECTION-SCREEN END OF SCREEN 1100.

SELECTION-SCREEN BEGIN OF SCREEN 1200 AS SUBSCREEN.
SELECTION-SCREEN BEGIN OF BLOCK b2 WITH FRAME TITLE text-020.
```

```
PARAMETERS: q1(10) TYPE c OBLIGATORY,
            q2(10) TYPE c OBLIGATORY,
            q3(10) TYPE c OBLIGATORY.
SELECTION-SCREEN END OF BLOCK b2.
SELECTION-SCREEN END OF SCREEN 1200.

DATA: ok_code TYPE sy-ucomm,
      save_ok TYPE sy-ucomm.

DATA: number(4) TYPE n VALUE '1100'.

START-OF-SELECTION.
  CALL SCREEN 100.

MODULE status_0100 OUTPUT.
  SET PF-STATUS 'SCREEN_100'.
ENDMODULE.

MODULE cancel INPUT.
  LEAVE PROGRAM.
ENDMODULE.

MODULE user_command_0100 INPUT.
  save_ok = ok_code.
  CLEAR ok_code.
  CASE save_ok.
    WHEN 'BUTTON1'.
      number = 1100.
    WHEN 'BUTTON2'.
      number = 1200.
  ENDCASE.
ENDMODULE.

AT SELECTION-SCREEN.
  MESSAGE s888(sabapdocu) WITH text-030 sy-dynnr.
```

This defines two selection screens – 1100 and 1200 – as subscreens.

The next screen (statically defined) for screen 100 is itself. It has the following layout:



The screen contains a subscreen area AREA and two pushbuttons with the function codes BUTTON1 and BUTTON2.

The screen flow logic for screen 100 is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE status_0100.
  CALL SUBSCREEN area INCLUDING sy-repid number.

PROCESS AFTER INPUT.
  MODULE cancel AT EXIT-COMMAND.
  CALL SUBSCREEN area.
  MODULE user_command_0100.
```

When you run the program, a screen appears on which selection screen 1100 is displayed as a subscreen. You can display either selection screen in the subscreen area, using the pushbuttons to switch between them. Before you switch from selection screen 1200 to 1100, you must fill out the obligatory fields. The data you enter is available to the program in the parameters in the PAI event.

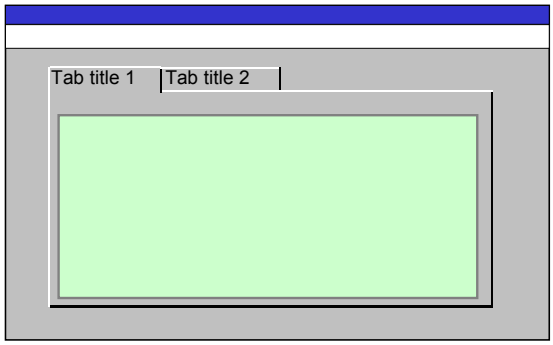
Selection screens as subscreens in a tabstrip control on a screen.

```
REPORT demo_sel_screen_in_tabstrip.

SELECTION-SCREEN BEGIN OF SCREEN 1100 AS SUBSCREEN
                                      NO INTERVALS.
SELECTION-SCREEN BEGIN OF BLOCK b1 WITH FRAME TITLE text-010.
PARAMETERS: p1(10) TYPE c,
            p2(10) TYPE c,
            p3(10) TYPE c.
SELECTION-SCREEN END OF BLOCK b1.
SELECTION-SCREEN END OF SCREEN 1100.

SELECTION-SCREEN BEGIN OF SCREEN 1200 AS SUBSCREEN
                                      NO INTERVALS.
SELECTION-SCREEN BEGIN OF BLOCK b2 WITH FRAME TITLE text-020.
PARAMETERS: q1(10) TYPE c OBLIGATORY,
            q2(10) TYPE c OBLIGATORY,
            q3(10) TYPE c OBLIGATORY.
SELECTION-SCREEN END OF BLOCK b2.
SELECTION-SCREEN END OF SCREEN 1200.

CONTROLS mytabstrip TYPE TABSTRIP.

DATA: ok_code TYPE sy-ucomm,
      save_ok TYPE sy-ucomm.

DATA: number(4) TYPE n VALUE '1100'.

START-OF-SELECTION.
  mytabstrip-activetab = 'BUTTON1'.
  CALL SCREEN 100.

MODULE status_0100 OUTPUT.
  SET PF-STATUS 'SCREEN_100'.
ENDMODULE.

MODULE cancel INPUT.
  LEAVE PROGRAM.
ENDMODULE.
```

```
MODULE user_command_0100 INPUT.
  save_ok = ok_code.
  CLEAR ok_code.
  CASE save_ok.
    WHEN 'BUTTON1'.
      mytabstrip-activetab = save_ok.
      number = 1100.
    WHEN 'BUTTON2'.
      mytabstrip-activetab = save_ok.
      number = 1200.
  ENDCASE.
ENDMODULE.

AT SELECTION-SCREEN.
  MESSAGE s888(sabapdocu) WITH text-030 sy-dynnr.
```

This defines two selection screens – 1100 and 1200 – as subscreens.

The next screen (statically defined) for screen 100 is itself. It has the following layout:



The screen contains a tabstrip area called MYTABSTRIP with three tab titles BUTTON1, BUTTON2 and BUTTON3. The function codes have the same name, and no special function type. All of the tab titles are assigned to a single subscreen area AREA.

The screen flow logic for screen 100 is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE status_0100.
  CALL SUBSCREEN area INCLUDING sy-repid number.

PROCESS AFTER INPUT.
  MODULE cancel AT EXIT-COMMAND.
  CALL SUBSCREEN area.
  MODULE user_command_0100.
```

This example is programmed in almost exactly the same way as the last one, and behaves in the same way. The only difference is that the pushbuttons have been replaced with tab titles, and the control MYTABSTRIP has been declared and filled. Scrolling between the tab pages is programmed in the ABAP program. Each time the user chooses a tab title, the function code from the OK_CODE field is assigned to the ACTIVETAB component of structure MYTABSTRIP. At the same time, the variable NUMBER is filled with the screen number of the subscreen that has to be displayed in the subscreen area AREA of the tabstrip control.

Tabstrip Controls on Selection Screens

As with screens, you can now use tabstrip controls [Page 653] on selection screens. To do this, you must define a tabstrip area and the associated tab pages, and assign a subscreen to the tab pages. You do not have to (indeed, cannot) declare the tabstrip control or program the screen flow logic in your ABAP program, since both are automatically generated.

To define a tabstrip area with tab pages, use the following statements in your selection screen definition:

```
SELECTION-SCREEN: BEGIN OF TABBED BLOCK <tab_area> FOR <n> LINES,
                  TAB (<len>) <tab1> USER-COMMAND <ucom1>
                        [DEFAULT [PROGRAM <prog>] SCREEN <scrn>],
                  TAB (<len>) <tab2> USER-COMMAND <ucom2>
                        [DEFAULT [PROGRAM <prog>] SCREEN <scrn>],
                  ...
                  END OF BLOCK <tab_area>.
```

This defines a tabstrip control <tab_area> with size <n>. The tab pages <tab1>, <tab2>… are assigned to the tab area. <len> defines the width of the tab title. You must assign a function code <ucom> area to each tab title. You can find out the function code from the field SY-UCOMM in the AT SELECTION-SCREEN event.

For each tab title, the system automatically creates a character field in the ABAP program with the same name. Before the selection screen is displayed, you can assign a text to the field. This then appears as the title of the corresponding tab page on the selection screen.

You must assign a subscreen to each tab title. This will be displayed in the tab area when the user chooses that title. You can assign one of the following as a subscreen:

- A subscreen screen [Page 647] defined using the Screen Painter.
- A selection screen subscreen [Page 754], defined in an ABAP program.

You can make the assignment either statically in the program or dynamically at runtime. If, at runtime, one of the tab titles has no subscreen assigned, a runtime error occurs.

- Static assignment

  Use the DEFAULT addition when you define the tab title. You can specify an ABAP program and one of its subscreens. If you do not specify a program, the system looks for the subscreen in the current program. When the user chooses the tab title, it is activated, and the subscreen is assigned to the tabstrip area. The static assignment is valid for the entire duration of the program, but can be overwritten dynamically before the selection screen is displayed.

- Dynamic assignment

  For each tab area, the system automatically creates a structure in the ABAP program with the same name. This structure has three components – PROG, DYNNR, and ACTIVETAB. When you assign the subscreens statically, the structure contains the name of the ABAP program containing the subscreen, the number of the subscreen, and the name of the tab title currently active on the selection screen (and to which these values are assigned). The default active tab page is the first page. You can assign values to the fields of the structure before the selection screen is displayed, and so set a subscreen dynamically.

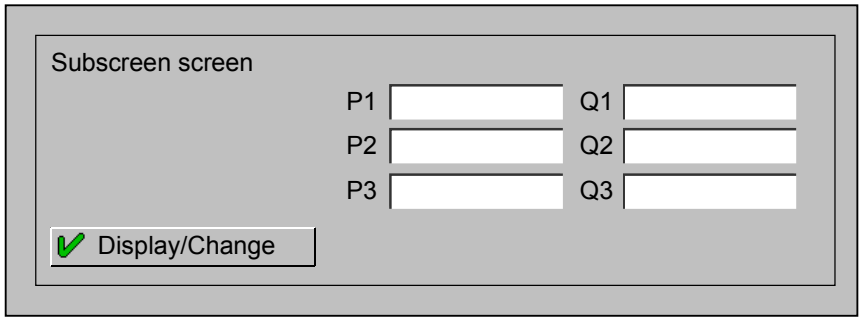
If you assign a normal subscreen screen to a tab title, the dialog modules containing its flow logic must be defined in the current ABAP program. If the subscreen is a selection screen, user actions will trigger the AT SELECTION-SCREEN event and its variants (see Selection Screen Processing [Page 739]). This includes when the user chooses a tab title. If one selection screen is included on another, AT SELECTION-SCREEN will be triggered at least twice – firstly for the "included" selection screen, then for the selection screen on which it appears.

```
REPORT demo_sel_screen_with_tabstrip.

DATA flag(1) TYPE c.

* SUBSCREEN 1

SELECTION-SCREEN BEGIN OF SCREEN 100 AS SUBSCREEN.
SELECTION-SCREEN BEGIN OF BLOCK b1 WITH FRAME.
PARAMETERS: p1(10) TYPE c,
            p2(10) TYPE c,
            p3(10) TYPE c.
SELECTION-SCREEN END OF BLOCK b1.
SELECTION-SCREEN END OF SCREEN 100.

* SUBSCREEN 2

SELECTION-SCREEN BEGIN OF SCREEN 200 AS SUBSCREEN.
SELECTION-SCREEN BEGIN OF BLOCK b2 WITH FRAME.
PARAMETERS: q1(10) TYPE c OBLIGATORY,
            q2(10) TYPE c OBLIGATORY,
            q3(10) TYPE c OBLIGATORY.
SELECTION-SCREEN END OF BLOCK b2.
SELECTION-SCREEN END OF SCREEN 200.

* STANDARD SELECTION SCREEN

SELECTION-SCREEN: BEGIN OF TABBED BLOCK mytab FOR 10 LINES,
                  TAB (20) button1 USER-COMMAND push1,
                  TAB (20) button2 USER-COMMAND push2,
                  TAB (20) button3 USER-COMMAND push3
                                   DEFAULT SCREEN 300,
                  END OF BLOCK mytab.

INITIALIZATION.
  button1 = text-010.
  button2 = text-020.
  button3 = text-030.
  mytab-prog = sy-repid.
  mytab-dynnr = 100.
  mytab-activetab = 'BUTTON1'.

AT SELECTION-SCREEN.
  CASE sy-dynnr.
    WHEN 1000.
      CASE sy-ucomm.
        WHEN 'PUSH1'.
          mytab-dynnr = 100.
          mytab-activetab = 'BUTTON1'.
        WHEN 'PUSH2'.
```

```
            mytab-dynnr = 200.
            mytab-activetab = 'BUTTON2'.
        ENDCASE.
      WHEN 100.
        MESSAGE s888(sabapdocu) WITH text-040 sy-dynnr.
      WHEN 200.
        MESSAGE s888(sabapdocu) WITH text-040 sy-dynnr.
    ENDCASE.

MODULE init_0100 OUTPUT.
  LOOP AT SCREEN.
    IF screen-group1 = 'MOD'.
      CASE flag.
        WHEN 'X'.
          screen-input = '1'.
        WHEN ' '.
          screen-input = '0'.
      ENDCASE.
      MODIFY SCREEN.
    ENDIF.
  ENDLOOP.
ENDMODULE.

MODULE user_command_0100 INPUT.
  MESSAGE s888(sabapdocu) WITH text-050 sy-dynnr.
  CASE sy-ucomm.
    WHEN 'TOGGLE'.
      IF flag = ' '.
        flag = 'X'.
      ELSEIF flag = 'X'.
        flag = ' '.
      ENDIF.
  ENDCASE.
ENDMODULE.

START-OF-SELECTION.
  WRITE: / 'P1:', p1,'Q1:', q1,
         / 'P2:', p2,'Q2:', q2,
         / 'P3:', p3,'Q3:', q3.
```

This program defines two selection screens – 100 and 200, as subscreens, and places a tabstrip control area with three tab pages on the standard selection screen. A subscreen screen 300 (from the same program) is assigned statically to the third tab page.

The layout of screen 300 is:



The input/output fields P1 to Q3 are defined by using [Ext.] the parameters from the ABAP program The pushbutton has the function code TOGGLE.

The screen flow logic for screen 300 is as follows:

```
PROCESS BEFORE OUTPUT.
  MODULE init_0100.

PROCESS AFTER INPUT.
  MODULE user_command_0100.
```

Both dialog modules are defined in the ABAP program.

When you run the program, the standard selection screen appears. In the INITIALIZATION event, the texts are defined on the tab titles, the subscreen selection screen 100 is assigned to the tab area, and the first tab title is activated.

User actions on the selection screen are processed in the AT SELECTION-SCREEN event block. In particular, it is here that the subscreens are assigned and tab titles activated when the user chooses one of the first two tab titles. This is not necessary for the third tab title, since the dynamic assignment (screen 300) is always placed in the structure MYTAB when the user chooses it.

Before the subscreen screen is displayed, the PBO module INIT_100 is executed. User actions on the subscreen screen trigger the PAI module. This includes when the user chooses a tab title. After that, the AT SELECTION-SCREEN event is triggered.

Messages in the status line show where an action has been processed.

Subscreens on Selection Screens

Displaying a subscreen in a subscreen area on a selection screen is a special case of a tabstrip control on a selection screen [Page 758]. To define a subscreen area on a selection screen, use these statements:

SELECTION-SCREEN: BEGIN OF TABBED BLOCK <sub_area> FOR <n> LINES,
END OF BLOCK <sub_area>.

Defining a subscreen area is the equivalent of defining a tabstrip area without tab titles. Before the selection screen is displayed, you **must** assign a subscreen to the subscreen area <sub_area>. To do this, use the components PROG and DYNNR of the structure <sub_area>, which is created automatically when you define the subscreen area. Assign the program name of the subscreen screen to the component PROG, and its screen number to DYNNR. You can use the following subscreens:

- A subscreen screen [Page 647] defined using the Screen Painter.
- A selection screen subscreen [Page 754], defined in an ABAP program.

If you have not assigned a subscreen when the selection screen is displayed, a runtime error occurs.

```
REPORT demo_sel_screen_with_subscreen.

TABLES sscrfields.

* SUBSCREEN 1

SELECTION-SCREEN BEGIN OF SCREEN 100 AS SUBSCREEN.
SELECTION-SCREEN BEGIN OF BLOCK b1 WITH FRAME TITLE text-010.
PARAMETERS: p1(10) TYPE c,
            p2(10) TYPE c,
            p3(10) TYPE c.
SELECTION-SCREEN END OF BLOCK b1.
SELECTION-SCREEN END OF SCREEN 100.

* SUBSCREEN 2

SELECTION-SCREEN BEGIN OF SCREEN 200 AS SUBSCREEN.
SELECTION-SCREEN BEGIN OF BLOCK b2 WITH FRAME TITLE text-020.
PARAMETERS: q1(10) TYPE c,
            q2(10) TYPE c,
            q3(10) TYPE c.
SELECTION-SCREEN END OF BLOCK b2.
SELECTION-SCREEN END OF SCREEN 200.

* SUBSCREEN 3

SELECTION-SCREEN BEGIN OF SCREEN 300 AS SUBSCREEN.
SELECTION-SCREEN BEGIN OF BLOCK b3 WITH FRAME TITLE text-030.
PARAMETERS: r1(10) TYPE c,
            r2(10) TYPE c,
            r3(10) TYPE c.
SELECTION-SCREEN END OF BLOCK b3.
SELECTION-SCREEN END OF SCREEN 300.
```

```
* STANDARD SELECTION SCREEN

SELECTION-SCREEN: FUNCTION KEY 1,
                  FUNCTION KEY 2.

SELECTION-SCREEN: BEGIN OF TABBED BLOCK sub FOR 10 LINES,
                  END OF BLOCK sub.

INITIALIZATION.
  sscrfields-functxt_01 = '@0D@'.
  sscrfields-functxt_02 = '@0E@'.
  sub-prog = sy-repid.
  sub-dynnr = 100.

AT SELECTION-SCREEN.
  CASE sy-dynnr.
    WHEN 100.
      IF sscrfields-ucomm = 'FC01'.
        sub-dynnr = 300.
      ELSEIF sscrfields-ucomm = 'FC02'.
        sub-dynnr = 200.
      ENDIF.
    WHEN 200.
      IF sscrfields-ucomm = 'FC01'.
        sub-dynnr = 100.
      ELSEIF sscrfields-ucomm = 'FC02'.
        sub-dynnr = 300.
      ENDIF.
    WHEN 300.
      IF sscrfields-ucomm = 'FC01'.
        sub-dynnr = 200.
      ELSEIF sscrfields-ucomm = 'FC02'.
        sub-dynnr = 100.
      ENDIF.
  ENDCASE.

START-OF-SELECTION.
  WRITE: / 'P1:', p1,'Q1:', q1, 'R1:', r1,
         / 'P2:', p2,'Q2:', q2, 'R2:', r2,
         / 'P3:', p3,'Q3:', q3, 'R3:', r3.
```

This program defines three subscreen selection screens, 100, 200, and 300. It also defines a subscreen area SUB on the standard selection screen. There are two pushbuttons in the application toolbar.

In the INITIALIZATION event, the subscreen selection screen 100 is assigned to the subscreen area. In the AT SELECTION-SCREEN event, the function keys are processed, and one of the other subscreens is assigned according to the user's choice.

Using Selection Criteria

When you define selection criteria, selection tables of the same name are declared in the program. The selections that the user specifies on the selection screen are stored in the rows of these tables in a standardized format. So that the program must not explicitly process the structure of the selection tables [Page 704] and their contents, selection tables can be addressed as a whole in certain ABAP statements. In these statements, complex selections are analyzed in a selection table. The programmer does not have to deal with their logic.

Selection Tables in the WHERE Clause [Page 765]

Selection Tables in Logical Expressions [Page 766]

Selection Tables in GET Events [Page 769]

Selection Tables in the WHERE Clause

The WHERE clause is an addition to Open SQL statements SELECT, UPDATE, and DELETE, and is used to restrict the values read during database accesses [Page 1037].

To use a selection table in the WHERE clause, you write:

..... WHERE <f> IN <seltab>.

<f> is the name of a database column, and <seltab> is the selection table that is assigned to this field. The relevant Open SQL statement accesses only those rows of the database table, where the contents of field <f> meet the selection criteria stored in <seltab>.

```
REPORT DEMO.

DATA WA_CARRID TYPE SPFLI-CARRID.

SELECT-OPTIONS AIRLINE FOR WA_CARRID.

SELECT CARRID FROM SPFLI INTO WA_CARRID WHERE CARRID IN
AIRLINE.
  WRITE WA_CARRID.
ENDSELECT.
```

Selection table AIRLINE is linked to the CARRID column of database table SPFLI. The WHERE clause of the SELECT statement checks if the contents of the CARRID column meet the selection criteria stored in AIRLINE.

If the selection table is filled as follows:

SIGN	OPTION	LOW	HIGH
I	BT	DL	UA
E	EQ	LH	

Then the database rows for all airlines between DL and UA except LH are read.

Selection Tables in Logical Expressions

To check if the value of a field meets the conditions in a selection table, you can use the following special logical expression [Page 225]:

... <f> IN <seltab> ....

This logical expression is true if the contents of field <f> meet the conditions stored in selection table <seltab>. <f> can be any elementary field.

If field <f> is identical to the field for which selection table <seltab> has been declared using SELECT-OPTIONS, you can use the following shortened form for the logical expression:

... <seltab> ....

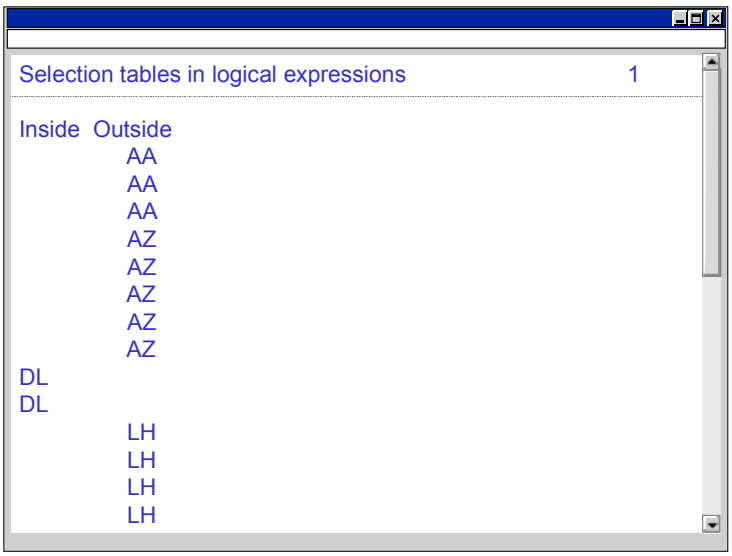
This shortened form is not possible for selection tables defined using the RANGES statement.

```
REPORT DEMO.

DATA WA_CARRID TYPE SPFLI-CARRID.

SELECT-OPTIONS AIRLINE FOR WA_CARRID.

WRITE: 'Inside', 'Outside'.

SELECT CARRID FROM SPFLI INTO WA_CARRID.
  IF WA_CARRID IN AIRLINE.
    WRITE: / WA_CARRID UNDER 'Inside'.
  ELSE.
    WRITE: / WA_CARRID UNDER Outside'.
  ENDIF.
ENDSELECT.
```

If the selection table is filled as follows:

SIGN	OPTION	LOW	HIGH
I	BT	DL	UA
E	EQ	LH	

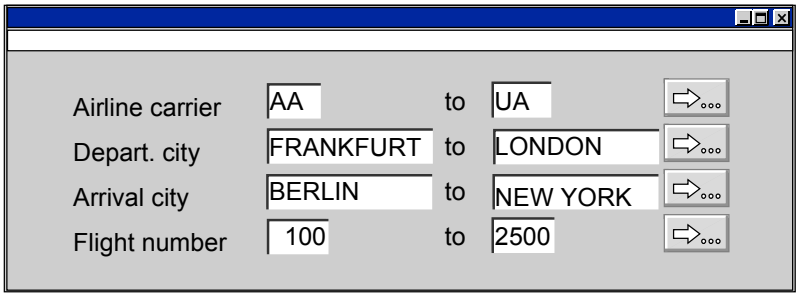
The list output appears as follows:



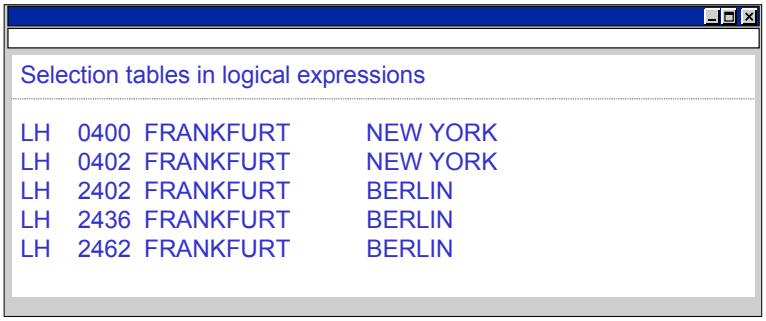
In the SELECT loop, all rows are read from database table SPFLI. If the IF statement is used, the program flow is branched into two statement blocks depending on the logical expression. The shortened form IF AIRLINE is also possible in this program.

```
REPORT DEMO.

DATA WA_SPFLI TYPE SPFLI.

SELECT-OPTIONS: S_CARRID FOR WA_SPFLI-CARRID,
                S_CITYFR FOR WA_SPFLI-CITYFROM,
                S_CITYTO FOR WA_SPFLI-CITYTO,
                S_CONNID FOR WA_SPFLI-CONNID.

SELECT * FROM SPFLI INTO WA_SPFLI.
  CHECK: S_CARRID,
         S_CITYFR,
         S_CITYTO,
         S_CONNID.
  WRITE: / WA_SPFLI-CARRID, WA_SPFLI-CONNID,
           WA_SPFLI-CITYFROM, WA_SPFLI-CITYTO.
ENDSELECT.
```

After starting the program, the selection screen appears, on which the user might fill the input fields as follows:



The list then appears as follows:



In the SELECT loop, all rows are read from database table SPFLI. The system lists only those rows that meet the conditions in the selection tables. The system leaves the loop pass after the CHECK statement. The CHECK statement uses the shortened forms of the logical expressions. The long forms are:

```
CHECK: WA_SPFLI-CARRID IN S_CARRID,
       WA_SPFLI-CITYFR IN S_CITYFR,
       WA_SPFLI-CITYTO IN S_CITYTO,
       SPFLI-CONNID IN S_CONNID.
```

Selection Tables in GET Events

When database tables are read using logical databases, you can use a special variant of the CHECK statement in the event blocks for GET [Page 958] events.

CHECK SELECT-OPTIONS.

The statement checks if the contents of the interface work area [Page 130] that has been filled by the logical database for the current GET event, meets the conditions in **all** selection tables that are linked to the database table read.

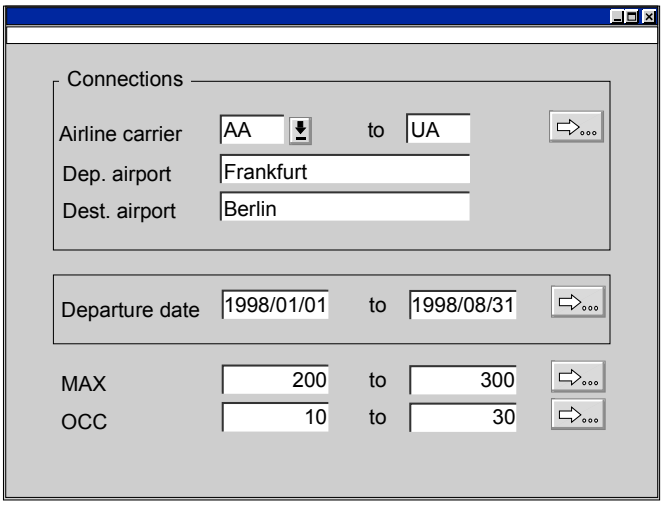
This variant of the CHECK statement only works in conjunction with selection criteria that are linked to database tables, and should only be used for GET events.

Since the CHECK statement cannot be used until after a row has been read by the logical database, you should use this variant only if the selections provided by the logical database are not sufficient to meet your requirements, and the relevant table is not designated for dynamic selections.

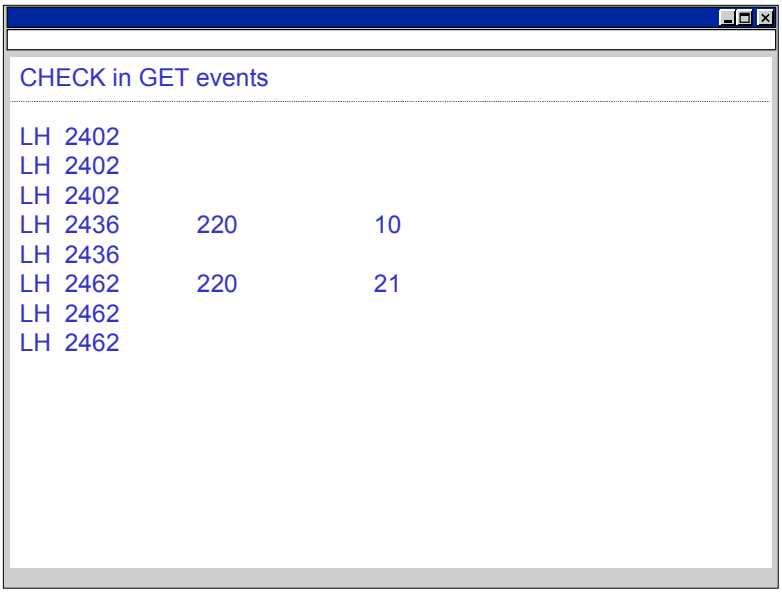
The following program is linked to logical database F1S.

```
REPORT DEMO.

NODES: SPFLI,SFLIGHT.

SELECT-OPTIONS: MAX     FOR SFLIGHT-SEATSMAX,
                OCC     FOR SFLIGHT-SEATSOCC.

GET SFLIGHT.
   WRITE: / SPFLI-CARRID, SPFLI-CONNID.
   CHECK SELECT-OPTIONS.
   WRITE: SFLIGHT-SEATSMAX, SFLIGHT-SEATSOCC.
```

After the program has been started, the selection screen appears, on which the user might fill the input fields as follows:



The list output appears as follows:



The system reads all rows from SFLIGHT that meet the selection criteria of the logical database. If these contents do not meet the selection criteria MAX and OCC defined in the program, the system leaves the GET event before writing the contents of SEATSMAX and SEATSOCC onto the screen.

Lists

Selection screens are one of the three types of screen in the R/3 System, along with dialog screens and lists. They are used to **display** data, and also allow user interaction. You create lists using ABAP statements. They can be output to the screen, but also to a printer.

Unlike screens, which contain defined elements like input/output fields and pushbuttons, each of which is identified by a name, and where data is exchanged with the ABAP program by means of identically-named fields, lists provide a freely-definable area that you fill using the WRITE, ULINE; and SKIP statements.

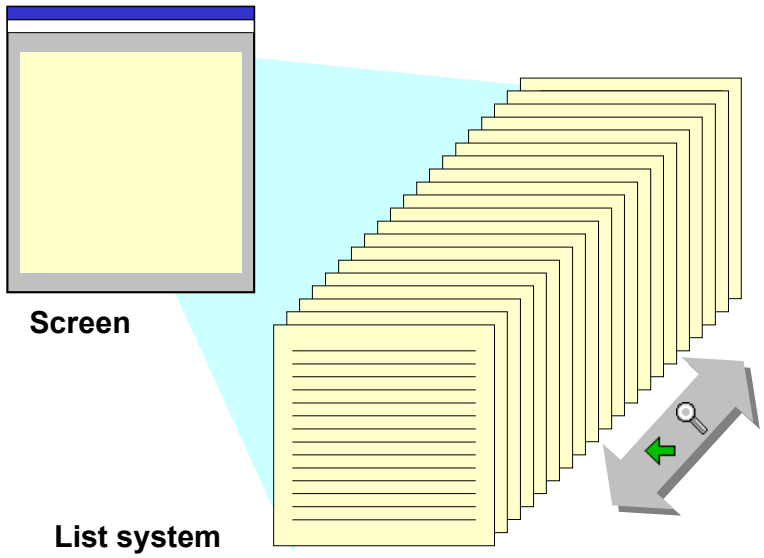
Creating and Displaying Lists

The ABAP statements that create lists actually create the list on the application server, where it is buffered. The list is then displayed either when the LEAVE TO LIST-PROCESSING statement occurs **in the program**, or, for executable programs, **automatically**. In executable programs, the list that you create is displayed (at the latest) after the last event block in the program.

When the list is displayed, the system calls the list processor, which displays the list on a special container screen (number 120). The container screen temporarily replaces the previous screen of the calling program. It inherits the same position, size, and GUI status. However, you can set a special GUI status for list processing before the list is displayed. In an executable program, the container screen replaces the standard selection screen (screen 1000), and automatically has the default list status.

User Actions and Detail Lists

When a list is displayed, the list processor has control of the program. Interactive user actions on a list trigger events in the ABAP program. Output statements in these event blocks create detail lists, which are then automatically displayed at the end of the event block. You can create up to 19 detail lists for a single basic list. Detail lists temporarily replace the previous list on the container screen. A basic list and its detail lists form a list system of up to twenty levels. Users can navigate between the different levels.



Lists and Screens

From a screen, you can call the list processor and display up to twenty lists in a list system using the LEAVE TO LIST-PROCESSING statement. When you start processing a screen, the list system is always initialized. This means that all list output statements apply to the basic list, and there are not yet any detail lists. If you start a new screen sequence [Page 1000] during list processing (CALL SCREEN statement), the list system of the original screen is retained. At the end of the screen sequence, the program returns to the last list level to have been displayed.

Creating Lists [Page 773]

User Actions and Detail Lists [Page 854]

Lists and Screens [Page 895]

Printing Lists [Page 904]

Creating Lists

The following sections describe how to create lists in ABAP. Lists are always displayed using the automatic list display functions in executable programs.

Creating Simple Lists with the WRITE Statement [Page 774]

Creating Complex Lists [Page 788]

Creating Simple Lists with the WRITE Statement

This section describes how to create simple output lists on the screen. To do this, you use ABAP statements WRITE, ULINE and SKIP.

This section includes the following topics:

When you run executable programs (reports), the system automatically displays the list generated by the program when program processing is completed. When running a dialog program (module pool), you can send the list to the screen using the LEAVE TO LIST-PROCESSING statement.

ABAP allows you to generate more complex and effective output lists, both on the screen and on paper, than that covered here. The following sections are based on this introduction.

The WRITE Statement

The basic ABAP statement for displaying data on the screen is WRITE.

Syntax

```
WRITE <f>.
```

This statement writes field <f> to the current list in its standard output format. By default, the list is displayed on the screen.

Field <f> can be

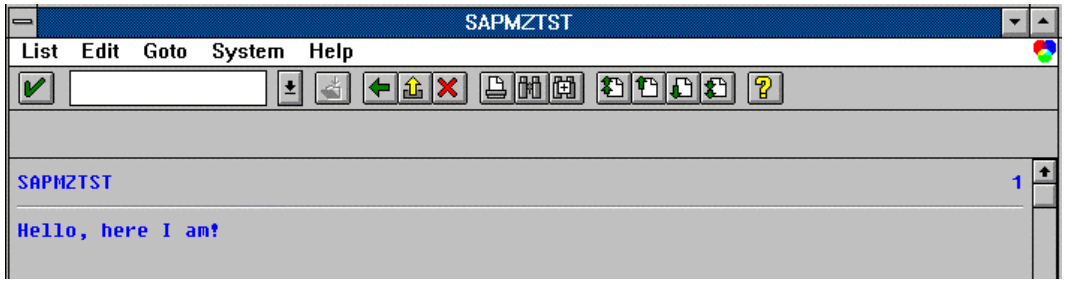
- any data object (see Data Objects [Page 118])
- a field symbol or formal parameter (see Working with Field Symbols [Page 201]).
- a text symbol (see Maintaining Text Elements [Ext.])

You can print the current output list directly from the output screen by choosing *Print*.

If a selection screen is defined for the program (see Selection Screens [Page 681]), you can choose *Execute and print* on the selection screen. Then, the list is not displayed on the screen, but sent directly to a printer.

```
PROGRAM sapmztst.

WRITE 'Hello, here I am!'.
```

When you start this program, the system leaves the current screen (this may be the *ABAP Editor:Initial Screen*) and branches to the output screen:



The name of the output screen is identical to the title of the program specified in the program attributes (see Maintaining Program Attributes [Page 75]).

The first line on the screen contains the list header. By default, the list header is identical to the title of the program. However, you can maintain the list header alone outside the actual program without affecting the program title. For more information on this topic, see Maintaining Text Elements [Ext.]. The current page number (1) appears on the right.

A horizontal line is displayed, then the actual list begins.

You can choose *Search* to search for specific patterns.

The WRITE Statement

On the screen, the output is normally left-justified. If you use several WRITE statements, the output fields are displayed one after the other, each separated by one column (that is, one blank). If there is not enough space for an output field in the current line, a new line is started.

```
PROGRAM sapmtest.

TABLES spfli.

.............

WRITE: 'COMPANY: ', spfli-carrid.
```

Note the use of the colon and the commas. The example contains **two** WRITE statements that are combined into a statement chain.

In the above example, two fields, literal 'COMPANY: ' and component CARRID of table work area SPFLI, are displayed on the screen.

```
COMPANY:  AA
```

The format of the data fields on the output screen depends on their data type (see Predefined Elementary Data Types [Page 97]).

Output format of predefined data types

Data Type	Output length	Positioning
C	field length	left-justified
D	8	left-justified
F	22	right-justified
I	11	right-justified
N	field length	left-justified
P	2 * field length (+1)	right-justified
T	6	left-justified
X	2 * field length	left-justified

The numeric data types F, I, and P are right-justified and padded with blanks on the left. If there is sufficient space, thousands separators are also displayed. If a type P field contains decimal places, the default output length is increased by one.

With data type D, the internal format of a date differs from its output format. When you use the WRITE statement for displaying data, the system automatically converts dates of type D based on the format specified in the user's master record (for example, DD/MM/YYYY or MM/DD/YYYY).

```
PROGRAM sapmtest.

DATA number TYPE p VALUE '-1234567.89' DECIMALS 2.

WRITE: 'Number', number, 'is packed'.
```

The output appears as follows:

```
Number     1,234,567.89- is packed
```

The field NUMBER has a total length of 13, made up of 9 digits, decimal point, minus sign, and two thousand separators. The output length of the NUMBER field is 2*8+1=17 because the field length of a type P field is 8. The remaining characters are filled up with spaces. This means that there are five blanks between the literal 'Number' and the number itself.

Positioning WRITE Output on the List

You can position the output of a WRITE statement on the list by making a format specification before the field name as follows:

Syntax

```
WRITE AT [/][<pos>][(<len>)] <f>.
```

where

- the slash '/' denotes a new line,
- <pos> is a number or variable up to three digits long denoting the position on the screen,
- <len> is a number or variable up to three digits long denoting the output length.

If the format specification contains only direct values (that is, no variables), you can omit the keyword AT.

```
WRITE 'First line.'.
WRITE 'Still first line.'
WRITE / 'Second line.'
WRITE /13 'Third line.'
```

This generates the following output on the screen:

```
First Line. Still first line.
Second line.
Third line.
```

If you specify a certain position <pos>, the field is always placed in that position regardless of whether or not there is enough space available or whether other fields are overwritten.

```
DATA: len TYPE i VALUE 10,
      pos TYPE i VALUE 11,
      text(10) TYPE c  VALUE '1234567890'
WRITE 'The text ------------ appears in the text.'.
WRITE AT pos(len) text.
```

This produces the following output:

```
The text -1234567890- appears in the text.
```

If the output length <len> is too short, fewer characters are displayed. Numeric fields are truncated on the left and prefixed with an asterisk (*). All other fields are truncated on the right, but no indication is given that the field is shorter.

```
DATA: number   TYPE i VALUE  1234567890,
      text(10) TYPE c VALUE 'abcdefghij'.
```

```
WRITE: (5) number, /(5) text.
```

This produces the following output:

```
*7890

abcde
```

In the default setting, you cannot create empty lines with the WRITE statement. You learn more about empty lines and how to change the default setting under Inserting Blank Lines [Page 799] in the section 'Creating Lists'.

```
WRITE:   'One',
       / '   ',
       / 'Two'.
```

This produces the following output:

```
One
Two
```

The system suppresses lines that contain nothing but blanks.

Formatting Options

You can use various formatting options with the WRITE statement.

Syntax

```
WRITE .... <f> <option>.
```

Formatting options for all data types

Option	Function
LEFT-JUSTIFIED	Output is left-justified.
CENTERED	Output is centered.
RIGHT-JUSTIFIED	Output is right-justified.
UNDER <g>	Output starts directly under field <g>.
NO-GAP	The blank after field <f> is omitted.
USING EDIT MASK <m>	Specifies format template <m>.
USING NO EDIT MASK	Deactivates a format template specified in the ABAP Dictionary.
NO-ZERO	If a field contains only zeros, these are replaced by blanks. For type C and N fields, leading zeros are replaced automatically.

Formatting options for numeric fields

Option	Function
NO-SIGN	The leading sign is not displayed on the screen.
DECIMALS <d>	<d> defines the number of digits after the decimal point.
EXPONENT <e>	In type F fields, the exponent is defined in <e>.
ROUND <r>	Type P fields are multiplied by 10**(-r) and then rounded.
CURRENCY <c>	Format according to currency <c> in table TCURX.
UNIT <u>	The number of decimal places is fixed according to unit <u> specified in table T006 for type P fields.

Formatting options for date fields

Option	Function
DD/MM/YY	Separators as defined in user's master record.
MM/DD/YY	Separators as defined in user's master record.
DD/MM/YYYY	Separators as defined in user's master record.
MM/DD/YYYY	Separators as defined in user's master record.

DDMMYY	No separators.
MMDDYY	No separators.
YYMMDD	No separators.

For more information on formatting options and the exclusion principles for some of these options, see the keyword documentation of the WRITE statement.

Below are some examples of formatting options. For more examples, see Creating Complex Lists [Page 788]. The decimal character and thousands separators (period or comma) of numeric fields are defined in the user's master record

ABAP Code	Screen output

`DATA: g(5) TYPE c VALUE 'Hello',`
`f(5) TYPE c VALUE 'Dolly'.`
`WRITE: g, f.`
`WRITE: /10 g,`
`/ f UNDER g.`
`WRITE: / g NO-GAP, f.`
	`Hello Dolly`
	`Hello`
	`Dolly`
	`HelloDolly`

`DATA time TYPE t VALUE '154633'.`
`WRITE: time,`
`/(8) time USING EDIT MASK '__:__:__'.`
	`154633`
	`15:46:33`

`WRITE: '000123',` `/ '000123' NO-ZERO.`	`000123` `123`
`DATA float TYPE f VALUE '123456789.0'.` `WRITE float EXPONENT 3.`	`123456,789E+03`

`DATA pack TYPE p VALUE '123.456'`
`DECIMALS 3.`
`WRITE pack DECIMALS 2.`
`WRITE: / pack ROUND -2,`
`/ pack ROUND -1,`
`/ pack ROUND 1,`
`/ pack ROUND 2.`
	`123,46`
	`12.345,600`
	`1.234,560`
	`12,346`
	`1,235`

`WRITE: sy-datum,` `/ sy-datum yymmdd.`	`27.06.1995` `950627`

Apart from the formatting options shown in the above tables, you can also use the formatting options of the FORMAT statement. These options allow you to specify the intensity and color of your output. For more information, see The FORMAT Statement [Page 833].

Displaying Symbols and Icons on the List

You can output symbols or R/3 icons on a list by using the following syntax:

Syntax

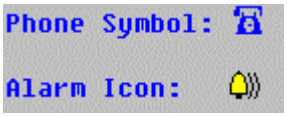
```
WRITE <symbol-name> AS SYMBOL.

WRITE <icon-name>   AS ICON.
```

The names of symbols and icons (<symbol-name> and <icon-name>) are system-defined constants that are specified in the include programs <SYMBOL> and <ICON> (the angle brackets are part of the name). The includes also contain a short description of the symbols and icons. The easiest way to output symbols and icons is to use a statement structure (see the example in Using WRITE via a Statement Structure [Page 785]).

To make symbols and icons available to your program, you must import the appropriate include or the more comprehensive include <LIST> in your program. For further information about importing include programs, see Using Include Programs [Page 449].

```
INCLUDE <symbol>.

INCLUDE <icon>.

/  'Phone Symbol:', SYM_PHONE AS SYMBOL.

SKIP.

WRITE: /  'Alarm Icon:   ', icon_alarm AS ICON.
```

This produces the following output:



You can replace both the above INCLUDE statements with one single INCLUDE statement:

```
INCLUDE <list>.
```

Blank Lines and Drawing Lines

Horizontal lines

You can generate horizontal lines on the output screen by using the following syntax:

Syntax

```
ULINE [AT [/][<pos>][(<len>)]].
```

This is equivalent to

```
WRITE [AT [/][<pos>][(<len>)]] SY-ULINE.
```

The format specifications after AT are exactly the same as the format specifications described for the WRITE statement in Positioning WRITE Output on the Screen [Page 778].

If there are no format specifications, the system starts a new line and fills it with a horizontal line. Otherwise, horizontal lines are output as specified.

Another way of generating horizontal lines is to type the appropriate number of hyphens in a WRITE statement as follows:

```
WRITE [AT [/][<pos>][(<len>)]] '-----...'.
```

Vertical lines

You generate vertical lines on the output screen by using the following syntax:

Syntax

```
WRITE [AT [/][<pos>]] SY-VLINE.
```

or

```
WRITE [AT [/][<pos>]] '|'.
```

Blank lines

You can generate blank lines on the screen by using the following syntax:

Syntax

```
SKIP [<n>].
```

Starting with the current line, this statement generates <n> blank lines on the output screen. If no value is specified for <n>, one blank line is output. In the standard setting, you cannot create empty lines with the WRITE statement alone.

To position the output data in a specific line on the screen, use:

Syntax

```
SKIP TO LINE <n>.
```

This statement allows you to move the output position upwards or downwards.

For more information and examples, see Creating Complex Lists [Page 788].

Displaying Field Contents as Checkboxes

You can output the first character of a field as a checkbox on the output screen by using the following syntax:

Syntax

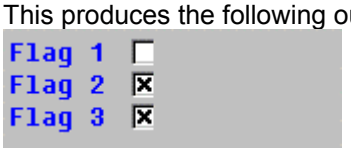
```
WRITE <f> AS CHECKBOX.
```

If the first character of field <f> is an "X", the checkbox is displayed filled. If the first character is SPACE, the checkbox is displayed blank.

In other words, the user can fill or clear them with a mouse click. For information on how you can check if output fields are ready for input or not, see Enabling Fields for Input [Page 839]. Fields that are ready for input are an essential component of interactive lists that allow a dialog with the user (see Interactive Lists [Page 854].

```
DATA: flag1(1) TYPE c    VALUE ' ',
      flag2(1) TYPE c    VALUE 'X',
      flag3(5) TYPE c    VALUE 'Xenon'.
WRITE: / 'Flag 1 ', flag1  AS CHECKBOX,
       / 'Flag 2 ', flag2  AS CHECKBOX,
       / 'Flag 3 ', flag3  AS CHECKBOX.
```

This produces the following output list:

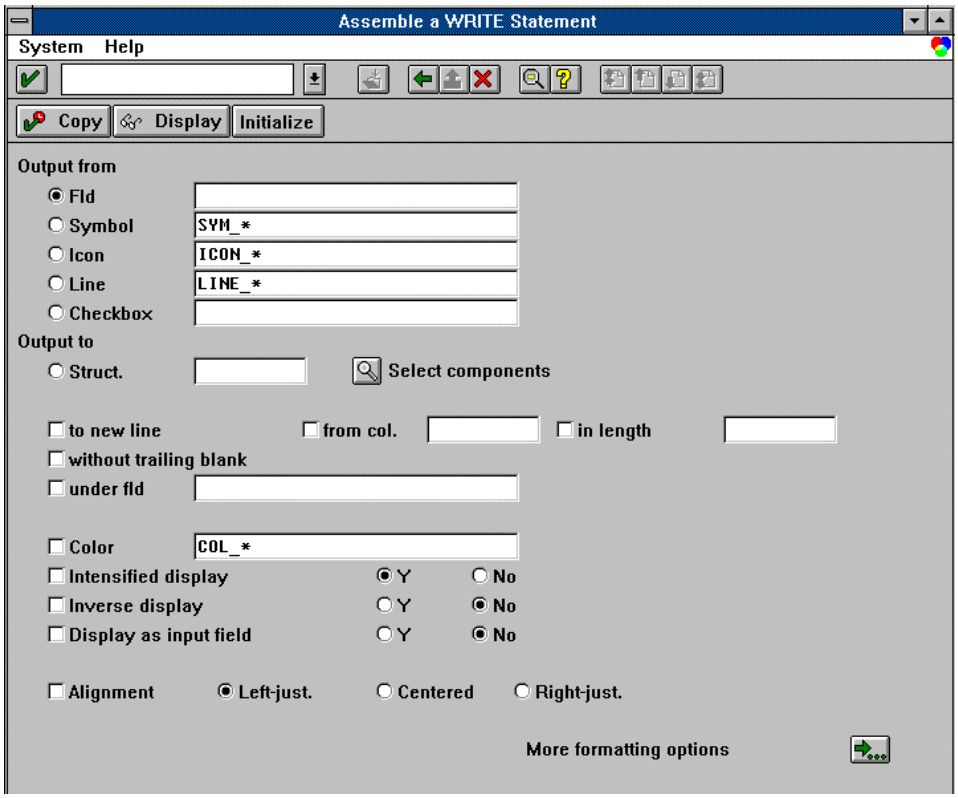


For FLAG2 and FLAG3, the checkboxes are filled because the first character of these fields is "X". The user can change the contents of the checkboxes with a mouse click.

Using WRITE via a Statement Structure

The R/3 System provides a useful facility for trying out all options and output formats of the WRITE statement and inserting them into your program. To do this, choose *Edit* → *Insert Statement...* in the ABAP Editor and then select WRITE in the relevant dialog box:

When you have confirmed your selection with *Enter*, you see the following screen:



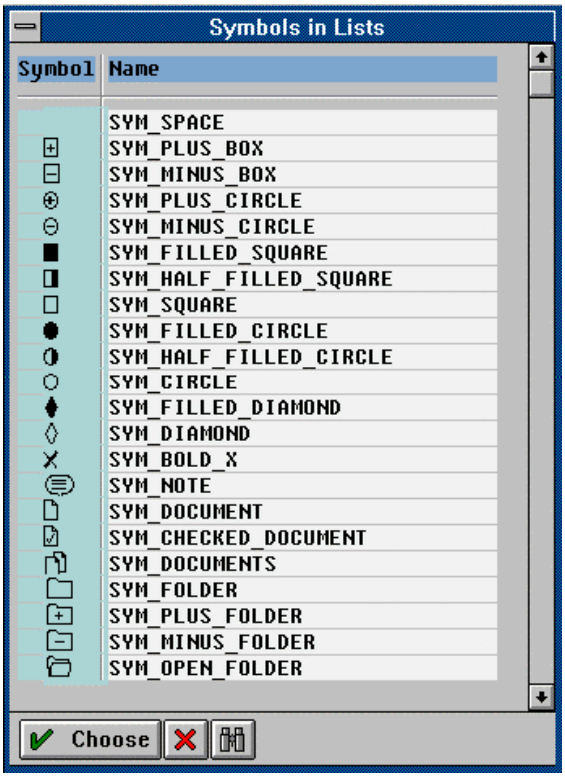
On this screen, you can

- determine the output format of an internal field by entering its name or a literal in field *Fld*. You then choose the formatting options on this screen or on another screen which you can access by selecting *More formatting options*  .
- generate the WRITE statements for symbols, icons, lines, and checkboxes simply by selecting the appropriate fields.
- generate the WRITE statements for components of structures defined in the ABAP Dictionary. This is useful, for example, after executing a SELECT statement (see Reading Data from Database Tables [Page 1043]).

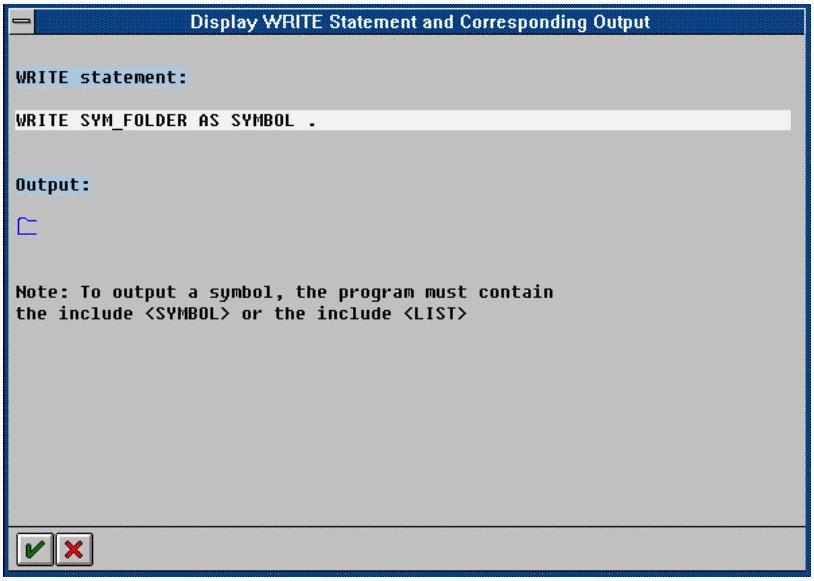
  

  On the screen *Assemble a WRITE-Statement*, select the radio button *Symbol* and then *Display*. The following dialog box appears:

Using WRITE via a Statement Structure



Here, you can choose a symbol, for example, SYM_FOLDER. The next dialog box displays the relevant WRITE statement and the resulting output on the output screen:



In addition, a note is displayed informing you that you need an include program in your program (see Displaying Symbols and Icons on the Screen [Page 782]).

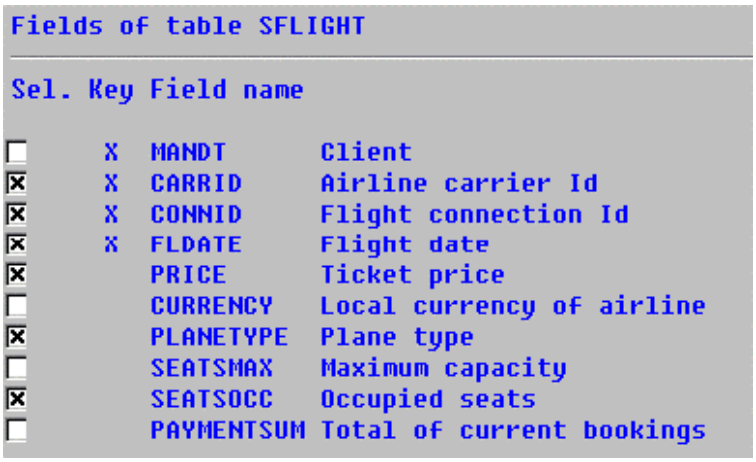
After choosing *Continue*, you see that the *Symbol* field on the *Assemble a WRITE Statement* screen now contains a value:

If you now choose *Execute*, the following text is inserted into your program:

```
WRITE sym_folder AS SYMBOL.
```

On the *Assemble a WRITE Statement* screen, select the radio button *Structure* and enter the following in the appropriate input field:

Then, choose *Select components*. On the next screen, you can select the components of the ABAP Dictionary structure SFLIGHT you want to output with WRITE, for example:



If you adopt this selection, the following WRITE statement is inserted into your program:

```
WRITE: sflight-carrid,
       sflight-connid,
       sflight-fldate,
       sflight-price,
       sflight-planetype,
       sflight-seatsocc.
```

Creating Complex Lists

Lists are the output medium for structured, formatted data from ABAP programs. Each program can produce up to 21 lists, one basic list and 20 secondary lists. The basic list is the standard screen of an executable program (report). You can display the basic list in a transaction using the LEAVE TO LIST-PROCESSING statement. This section deals with creating lists in general. That means, most of the statements described here apply to basic as well as to secondary lists. By default, the system transfers the output of a program to the basic list. In most cases, the basic list is the only list of a program. For this reason, the examples in this section mainly deal with the basic list. For information on how to program secondary lists, see Interactive Lists [Page 854].

From within your ABAP program, you can either output a list on the screen or send it to the SAP spool system. By default, the list is displayed on the screen. All examples in this section use the default. For information on how to print lists, see Printing Lists [Page 904].

The basic ABAP statement for writing data to lists is the WRITE statement. Other output statements are ULINE and SKIP. For details concerning these three statements, see Creating Simple Lists with the WRITE Statement [Page 774].

The following topics describe the structure of a list and the options you have to layout a list when creating it:

The Standard List [Page 789]

The Self-Defined List [Page 795]

Lists with Several Pages [Page 805]

Scrolling through Interactive Lists [Page 815]

Laying Out List Pages [Page 825]

The Standard List

If your ABAP program contains only the WRITE, SKIP, and ULINE output statements and not any of the editing statements described later on in this section, the system transfers the output to a standard list. In executable programs, the standard list is automatically displayed on the screen after the data is selected.

The topics below describe:

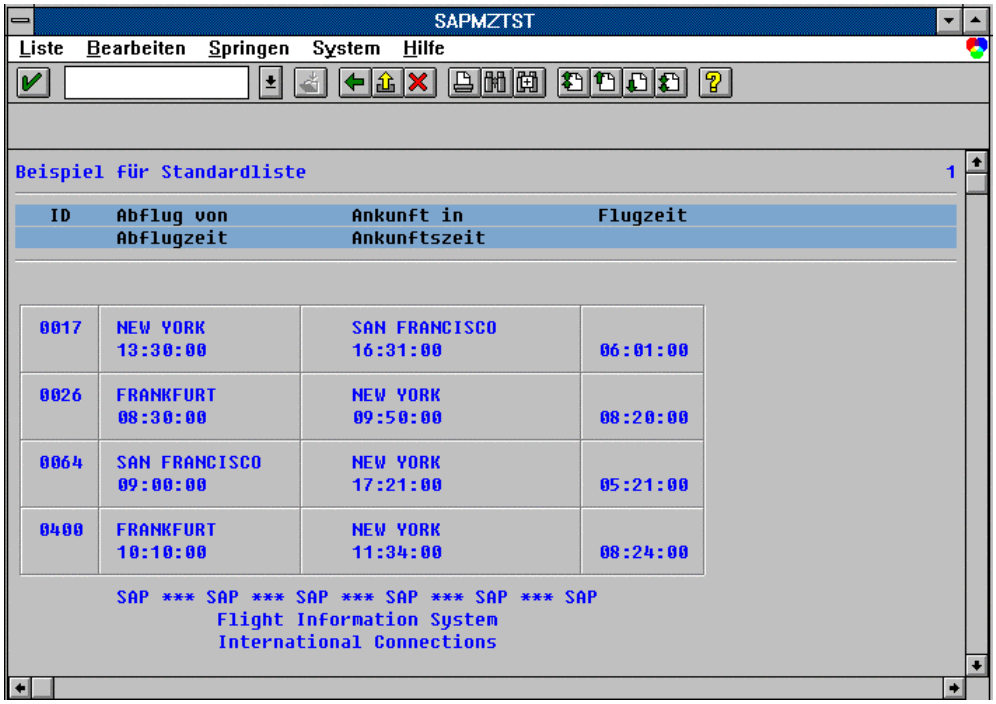
Structure of the Standard List [Page 790]

GUI Status for the Standard List [Page 792]

Structure of the Standard List

Overview

The output screen below shows a standard list:



To create this standard list, use the following sample program.

```
REPORT demo_list_standard.

TABLES spfli.

SKIP.
ULINE AT /(62).

SELECT * FROM spfli WHERE connid GE 0017
                       AND connid LE 0400.
  WRITE: / sy-vline, spfli-connid, sy-vline,
         (15) spfli-cityfrom, 26 sy-vline,
         31 spfli-cityto, 51 sy-vline, 62 sy-vline,
         / sy-vline, 8 sy-vline,
         spfli-deptime UNDER spfli-cityfrom, 26 sy-vline,
         spfli-arrtime UNDER spfli-cityto,   51 sy-vline,
         spfli-fltime, 62 sy-vline.
  ULINE AT /(62).
ENDSELECT.
```

```
WRITE: /10 'SAP *** SAP *** SAP *** SAP *** SAP *** SAP',
         /19(43) 'Flight Information System',
         /19(43) 'International Connections'.
```

The SELECT statement reads selected lines from the database table SPFLI. Within the SELECT loop, the WRITE, SKIP, and ULINE statements output fields of the table work area SPFLI as well as horizontal and vertical lines to the list.

Standard page header

The standard page header consists at least of a two-line standard header. The first line of the standard header contains the list header and the page number. The second line is made up of a horizontal line. When the program is executed, the list header is stored in the system field SY-TITLE. If necessary, you can add up to four lines of column headers and another horizontal line to the standard header.

GUI Status for the Standard List [Page 792] explains how to maintain list and column headers.

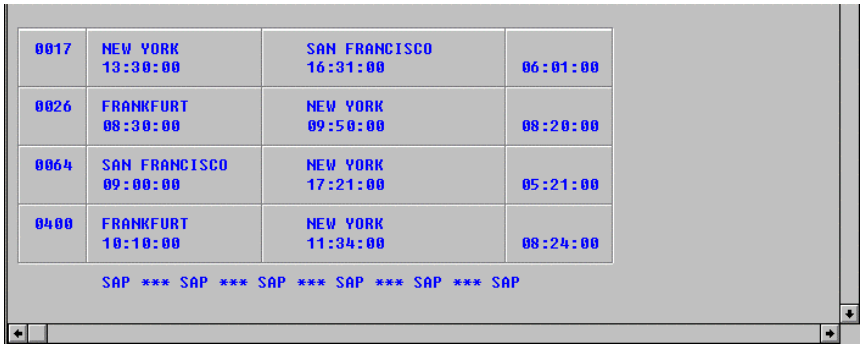
The width of the standard page header is automatically adapted to the window width.

If the user scrolls vertically through the list, the standard page header remains visible. Only the list beneath the header is scrolled.

If the user scrolls horizontally through the list, list header and page number remain visible.

Standard Page

Beneath the page header, the output data appears. The standard list consists of one single page of dynamic length (internal limit: 60,000 lines). The output length is determined by the current list size.



0017	NEW YORK 13:30:00	SAN FRANCISCO 16:31:00	06:01:00
0026	FRANKFURT 08:30:00	NEW YORK 09:50:00	08:20:00
0064	SAN FRANCISCO 09:00:00	NEW YORK 17:21:00	05:21:00
0400	FRANKFURT 10:10:00	NEW YORK 11:34:00	08:24:00

SAP *** SAP *** SAP *** SAP *** SAP *** SAP

The output screen includes a vertical scrollbar that allows the user to scroll through lists whose pages are longer than the window.

Width of the Standard List

The width of the standard list depends on the width of the window from which the program is started. If the user's window size is smaller than or equal to the standard window size, the width of the standard page conforms to the standard window width. The user may have to scroll to view all parts of the list. If the user's window size exceeds the standard window width, the width of the standard list conforms to the window width selected. In short, the standard list is at least as wide as the standard window. The width of the standard window depends on the operating system.

The output screen includes a horizontal scrollbar that allows the user to scroll through lists wider than the window.

GUI Status for the Standard List

The output screen of the standard list contains the standard menu bar and the standard toolbar of the R/3 system.

To allow the user to scroll through the standard list, the system offers the scrollbars and the functions *First page, Previous page, Next page,* and *Last page*. To find certain patterns in lists, the user can choose *Edit* → *Find...*.

The user can use the following list-specific functions.

Printing the Output List

To print the list displayed on the screen, the user chooses *List* → *Print.*

The printed standard page header differs from the displayed standard page header, as it additionally contains the current date.

Printing the standard list created in Structure of the Standard List [Page 790] results in:

```
12.01.1996       Example for Standard List            1
------------------------------------------------------------
-----------
  ID  Departure from    Arrival at      Time of
     Departure Time    Arrival Time     Flight
------------------------------------------------------------
-----------

+------+-----------------+-------------------------+----------+
| 0017 | NEW YORK        |    SAN FRANCISCO        |          |
|      | 13:30:00        |    16:31:00             | 06:01:00 |
+------+-----------------+-------------------------+----------+
| 0064 | SAN FRANCISCO   |    NEW YORK             |          |
|      | 09:00:00        |    17:21:00             | 05:21:00 |
+------+-----------------+-------------------------+----------+
| 0400 | FRANKFURT       |    NEW YORK             |          |
|      | 10:10:00        |    11:34:00             | 08:24:00 |
+------+-----------------+-------------------------+----------+
| 0026 | FRANKFURT       |    NEW YORK             |          |
|      | 08:30:00        |    09:50:00             | 08:20:00 |
+------+-----------------+-------------------------+----------+
         SAP *** SAP *** SAP *** SAP *** SAP *** SAP
                  Flight Information System
                  International Connections
```

Use this printing method only if you need a hardcopy of the screen list for testing purposes. For more information on how to print lists, see Printing Lists [Page 904].

Saving a List

To save the displayed list, the user chooses *List* → *Save.*

Saving the List in SAPoffice

When the user chooses *List* → *Save* → *Office,* a dialog box appears that allows the user to either store the displayed list in the user's Office folder or send it to another user.

Saving the List in the Reporting Tree

When the user chooses *List* → *Save* → *Reporting tree,* a dialog box appears that allows the user to save the displayed list in the appropriate branch of the reporting tree.

Saving the List as Local File on the Presentation Server

When the user chooses *List* → *Save* → *File,* a dialog box appears that allows the user to store the displayed list in different formats as a local file.

The format options are:

- *unconverted*: the system stores the file as text file.
- *Spreadsheet*: the system inserts tabs between the columns.
- *Rich Text Format*:the system stores the data formatted for word processing.

If the user stores the standard list created in Structure of the Standard List [Page 790] in *Rich text format* and re-displays it using a word-processing program that can read this format (for example, MS WORD), the list looks like this:

1996/03/05 Beispiel für Standardliste 1

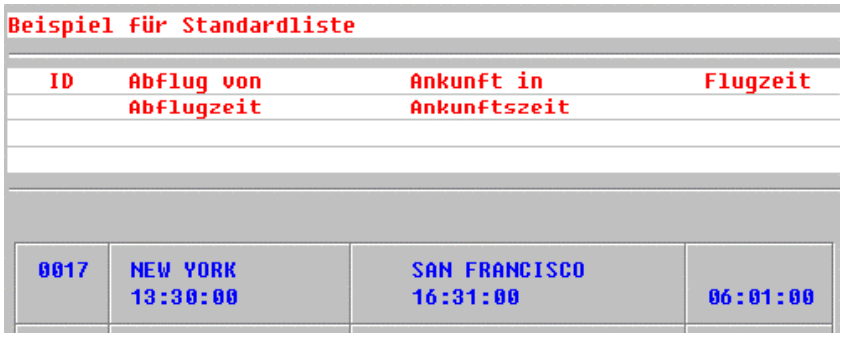
ID	Abflug von Abflugzeit	Ankunft in Ankunftszeit	Flugzeit
0017	NEW YORK 13:30:00	SAN FRANCISCO 16:31:00	06:01:00
0064	SAN FRANCISCO 09:00:00	NEW YORK 17:21:00	05:21:00
0400	FRANKFURT 10:10:00	NEW YORK 11:34:00	08:24:00
0026	FRANKFURT 08:30:00	NEW YORK 09:50:00	08:20:00

SAP *** SAP *** SAP *** SAP *** SAP *** SAP
Flight Information System
International Connections

Modifying List and Column Headers

Usually, you create and modify list and column headers as text elements (see Creating and Changing List and Column Headers [Ext.]). However, you can also modify these headers while displaying the list on the screen. To do so, choose *System* → *List* → *List header*. The lines of the page header now accept input:

GUI Status for the Standard List



Use this function, for example, to position the column headers exactly above the columns of the displayed list.

Save your changes. The system stores the modified headers as text elements of the program in the text pool of the current logon language. For more information on text elements, see Maintaining Text Elements [Ext.].

The Self-Defined List

You can modify the structure of the standard list and create lists of an individual structure. Use the options of the REPORT statement as well as the events TOP-OF-PAGE and END-OF-PAGE. The PROGRAM statement is equivalent to the REPORT statement and has the same options.

You have the following possibilities for modifications:

Individual Page Header [Page 796]

Determining the List Width [Page 798]

Creating Blank Lines [Page 799]

Determining the Page Length [Page 801]

Defining a Pager Footer [Page 803]

If your list consists of several pages, you can structure each page differently. For information on how to do this, see Lists with Several Pages [Page 805].

Individual Page Header

To layout a page header individually, you must define it in the processing block following the event keyword TOP-OF-PAGE:

Syntax

```
TOP-OF-PAGE.
  WRITE: ....
```

The TOP-OF-PAGE event occurs as soon as the system starts processing a new page of a list. The system processes the statements following TOP-OF-PAGE before outputting the first line on a new page. For information on events and processing blocks, see Controlling the Flow of ABAP Programs by Events [Page 952].

Remember to end the processing block following TOP-OF-PAGE by using an appropriate event keyword, such as START-OF-SELECTION, if you want to start processing the actual list afterwards (see Defining Processing Blocks [Page 941]).

The self-defined page header appears beneath the standard page header. If you want to suppress the standard page header, use the NO STANDARD PAGE HEADING option of the REPORT statement:

Syntax

```
REPORT <rep> NO STANDARD PAGE HEADING.
```

When you use this statement, the system does not display a standard page header on the pages of a list of program <rep>. If you have defined an individual page header using TOP-OF-PAGE, the system displays it.

During the event TOP-OF-PAGE, you can also fill the system fields SY-TVAR0 to SY-TVAR9 with values that should replace possible placeholders &0 to &9 in the standard page header.

When you scroll vertically, the self-defined page header remains visible as does the standard page header. However, the self-defined page header consists of normal list lines and therefore does not adapt automatically to the window width.

```
REPORT demo_list_page_heading NO STANDARD PAGE HEADING.
TOP-OF-PAGE.
  WRITE: sy-title, 40 'Page', sy-pagno.
  ULINE.
  WRITE: / 'SAP AG', 29 'Walldorf, ',sy-datum,
         / 'Neurottstr. 16', / '69190 Walldorf/Baden'.
  ULINE.
START-OF-SELECTION.
  DO 5 TIMES.
    WRITE / sy-index.
  ENDDO.
```

This program does not use the standard page header, but one that is self-defined following TOP-OF-PAGE. It is necessary to specify the event keyword START-

OF-SELECTION to implicitly end the TOP-OF-PAGE processing block. The output appears as follows:

```
SAPMZTST                                Page     1

SAP AG                      Walldorf,  16.01.1996
Neurottstr. 16
69190 Walldorf/Baden

        1
        2
        3
        4
        5
```

The self-defined page header consists of six lines. The program title comes from the SY-TITLE system field, the page number from SY-PAGNO. The self-defined page header is not as wide as the list.

Determining the List Width

To determine the width of the output list, use the LINE-SIZE option of the REPORT statement.

Syntax

```
REPORT <rep> LINE-SIZE <width>.
```

This statement determines the width of the output list of program <rep> as <width> characters. If you set <width> to zero, the system uses the width of the standard list [Page 790].

A line can be up to 255 characters long. However, if you intend to print the list, note that most printers cannot print lists wider than 132 characters. If you want to print the list directly while creating it, the page width must comply with one of the existing print formats. Otherwise, the system will not print the list (see Print Parameters [Page 908]). Make sure not to choose a list width exceeding 132 characters, unless you create the list for display only.

While creating the list, the system field SY-LINSZ contains the current line width. To adapt the list width to the current window width, see Lists with Several Pages [Page 805].

Horizontal lines that you create using the ULINE statement (without the AT option) automatically adapt to the self-defined list width.

```
REPORT demo_list_line_size LINE-SIZE 40.
WRITE:  'SY-LINSZ:', sy-linsz.
ULINE.
DO 20 TIMES.
  WRITE sy-index.
ENDDO.
```

This program creates the following output:

```
Defining the width                       1
----------------------------------------
SY-LINSZ:    40
----------------------------------------
         1          2          3
         4          5          6
         7          8          9
        10         11         12
        13         14         15
        16         17         18
        19         20
```

The example uses the standard page header. If you replace the LINE-SIZE value 40 with 60, the output looks as follows:

```
Defining the width                                           1
------------------------------------------------------------
SY-LINSZ:    60
------------------------------------------------------------
         1          2          3          4          5
         6          7          8          9         10
        11         12         13         14         15
        16         17         18         19         20
```

The standard page header and the underscores automatically conform to the list width.

Creating Blank Lines

To create blank lines, use the SKIP statement as follows:

Syntax

```
SKIP [<n>].
```

The system writes <n> blank lines into the current list, starting at the current line. If you omit the <n> option, the system creates one blank line.

The following restrictions apply for this operation:

- If the number of lines remaining on the current page is too small, the above SKIP statement produces a page break, displaying the page footer if any. The system positions the next output to the first line beneath the page header of the new page.
- At the beginning of a page, the system executes the above statement only, if this page is the first page of a list level or if the page was created using the NEW-PAGE statement. For all other pages, the system ignores this statement at the beginning of a page.
- If the above statement is the last output statement of the last list page (that is, there are no more WRITE or ULINE statements), the system ignores it.

In the default setting, the system does not output any blank lines created using the WRITE statement with the AT / option. A blank line is a line that contains character strings only and whose individual fields consist of nothing but blank characters. However, if you intend to output blank lines created by WRITE statements when outputting character strings, use this statement:

Syntax

```
SET BLANK LINES ON|OFF.
```

If you use the ON option, the system does not suppress the output of blank lines created using WRITE statements. To reset the default setting, use the OFF option.

You use this statement, for example, to represent empty table entries in the list. Note that the system displays a line containing nothing but, for example, empty input fields or empty checkboxes only if you specify SET BLANK LINES ON beforehand.

The following program creates five blank lines. The output '*****' appears in the sixth line.

```
REPORT sapmztst.
SKIP 5.
WRITE '*****'.
```

The following program does not create any blank lines. The output '*****' appears in the first line. The SET BLANK LINES OFF statement is used only to emphasize the default setting.

```
REPORT sapmztst.
SET BLANK LINES OFF.
DO 5 TIMES
  WRITE / ' '.
ENDDO:
WRITE '*****'.
```

Creating Blank Lines

The program below creates five blank lines, since the SET BLANK LINES ON statement is used. The output '*****' appears in the sixth line.

```
REPORT sapmztst.
SET BLANK LINES ON.
DO 5 TIMES
  WRITE / ' '.
ENDDO.
SET BLANK LINES OFF.
WRITE  / '*****'.
```

Determining the Page Length

To determine the page length of an output list, use the LINE-COUNT option of the REPORT statement.

Syntax

```
REPORT <rep> LINE-COUNT <length>[(<n>)].
```

This statement determines the page length of the output list of program <rep> as <length> lines. If you specify the optional number <n>, the system reserves <n> lines of the page length for the page footer. Those lines of the page footer that are not filled at the END-OF-PAGE event, appear as blank lines (see Defining a Page Footer [Page 803]).

If you set <length> to zero, the system uses the standard page length (see The Standard List [Page 789]). To adapt the page length to the current window size, see Lists with Several Pages [Page 805]. While the list is created, the system field SY-LINCT contains the current number of lines per page (that is <length>, or 0 for the standard page length).

Consider that the length of the page header is part of <length>. Thus, for the list itself you can use only <length> minus page header length minus <n> lines. If <length> is less than the page header length, a runtime error occurs.

If during list processing the system reaches the end of the area provided for the actual list, it outputs the page footer, if any, inserts some space, and starts a new page. The space inserted belongs to the list background and is not a line of the list. The SY-PAGNO system field always contains the current page number.

When determining the page length, keep in mind the following points:

- For screen output, use the standard page length to avoid page breaks in the middle of the screen.
- For printing lists, set the page length according to the printer requirements. Write your program in a way that it produces appropriate output for any page length. If you choose a page length that is not covered by one of the existing print formats, you may no longer be able to directly print a list while creating it. For more information, see Print Parameters [Page 908].
- Use fixed length specifications for form-like lists of a specified page layout only. Before coding a program for such a list, check whether you can use predefined SAP*script* forms. For information on forms, see the documentation BC - Style- and Layout Set Maintenance [Ext.].

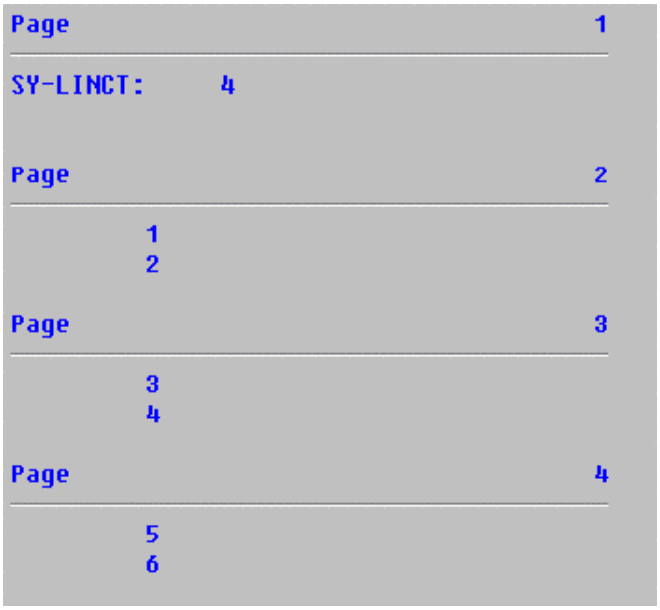
The following program is designed to demonstrate the usage of the LINE-COUNT option. Therefore, the different pages of the list appear on one screen page.

```
REPORT demo_list_line_count LINE-SIZE 40 LINE-COUNT 4.

WRITE:  'SY-LINCT:', sy-linct.
SKIP.
```

Determining the Page Length

```
DO 6 TIMES.
  WRITE / sy-index.
ENDDO.
```

This program sets the page length to four lines. It uses the standard page header. Suppose, the standard page header consists of the two-line list header. The output then may look as follows:



The list consists of four pages of four lines each. Each page is made up of the page header and two lines of the actual list. Note the space at the end of each page.

Defining a Page Footer

To define a page footer, use the END-OF-PAGE event. This event occurs if, while processing a list page, the system reaches the lines reserved for the page footer, or if the RESERVE statement triggers a page break. Fill the lines of the page footer in the processing block following the event keyword END-OF-PAGE:

Syntax

```
END-OF-PAGE.
  WRITE: ....
```

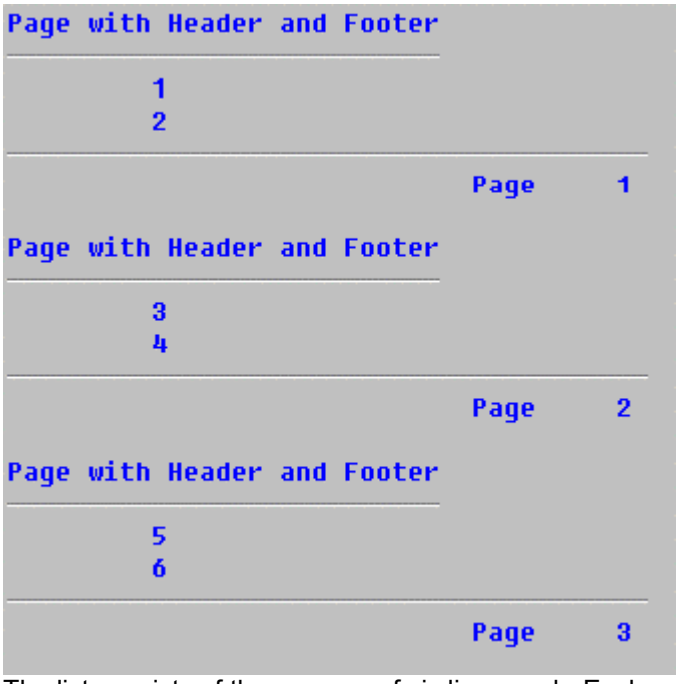
The system only processes the processing block following END-OF-PAGE if you reserve lines for the footer in the LINE-COUNT option of the REPORT statement (see Determining the Page Length [Page 801]).

Remember to end the processing block following END-OF-PAGE by using an appropriate event keyword, such as START-OF-SELECTION, if you want to start processing the actual list afterwards (see Defining Processing Blocks [Page 941]).

```
REPORT demo_list_end_of_page LINE-SIZE 40 LINE-COUNT 6(2)
                             NO STANDARD PAGE HEADING.
TOP-OF-PAGE.
  WRITE: 'Page with Header and Footer'.
  ULINE AT /(27).
END-OF-PAGE.
  ULINE.
  WRITE: /30 'Page', sy-pagno.
START-OF-SELECTION.
  DO 6 TIMES.
    WRITE / sy-index.
  ENDDO.
```

This program consists of three processing blocks. The standard page header is turned off. The page length is set to six lines, where two of them are reserved for the page footer.

Defining a Page Footer



The list consists of three pages of six lines each. Each page is made up of the self-defined two-line page header, of two lines of actual list, and of a two-line page footer. The current page number displayed in the page footer comes from the SY-PAGNO system field.

Lists with Several Pages

If in your program you write more lines to the output page of a list than are defined in the LINE-COUNT option of the REPORT statement, the system automatically creates a new page (see Determining the Page Length [Page 801].). Each new page contains the page header as well as the page footer defined for the program (if any).

Apart from automatic page breaks, you can use the NEW-PAGE and RESERVE statements to code page breaks explicitly. The options of the NEW-PAGE statement allow you to layout each page individually. You also need NEW-PAGE to print lists from within the program (see Printing from within the Program [Page 912]).

The following topics explain:

Programming Page Breaks [Page 806]

Standard Page Headers of Individual Pages [Page 809]

Page Length of Individual Pages [Page 811]

Page Width of List Levels [Page 814]

Programming Page Breaks

To program unconditional page breaks, use the NEW-PAGE statement.

To program page breaks depending on the number of empty lines left on a page, use the RESERVE statement.

Unconditional Page Break

To trigger a page break during list processing, use the basic form of the NEW-PAGE statement:

Syntax

```
NEW-PAGE.
```

This statement

- ends the current page. All other output appears on a new page.
- only starts a new page if output is written to the current page as well as to the new page after NEW-PAGE. The system then increases the SY-PAGNO system field by one. You cannot produce empty pages.
- does not trigger the END-OF-PAGE event. This means that the system does not output a page footer even if one is defined.

```
REPORT demo_list_new_page LINE-SIZE 40.

TOP-OF-PAGE.

  WRITE: 'TOP-OF-PAGE', sy-pagno.
  ULINE AT /(17).

START-OF-SELECTION.

  DO 2 TIMES.
    WRITE / 'Loop:'.
    DO 3 TIMES.
      WRITE / sy-index.
    ENDDO.
    NEW-PAGE.
  ENDDO.
```

This sample program uses both the standard page header whose list header 'Standard Page Header' is defined as text element, and a self-defined page header. Both page headers appear on each page.

```
Standard Page Header                    1
------------------------------------------
TOP-OF-PAGE     1
----------------
Loop:
         1
         2
         3

Standard Page Header                    2
------------------------------------------
TOP-OF-PAGE     2
----------------
Loop:
         1
         2
         3
```

The DO loop encounters the NEW-PAGE statement twice, but executes a page break only once. After the second NEW-PAGE statement, no output is available.

Conditional Page Break- Defining a Block of Lines

To execute a page break on the condition that less than a certain number of lines is left on a page, use the RESERVE statement:

Syntax

```
RESERVE <n> LINES.
```

This statement triggers a page break if less than <n> free lines are left on the current list page between the last output and the page footer. <n> can be a variable. Before starting a new page, the system processes the END-OF-PAGE event. RESERVE only takes effect if output is written to the subsequent page (the system will not generate an empty page).

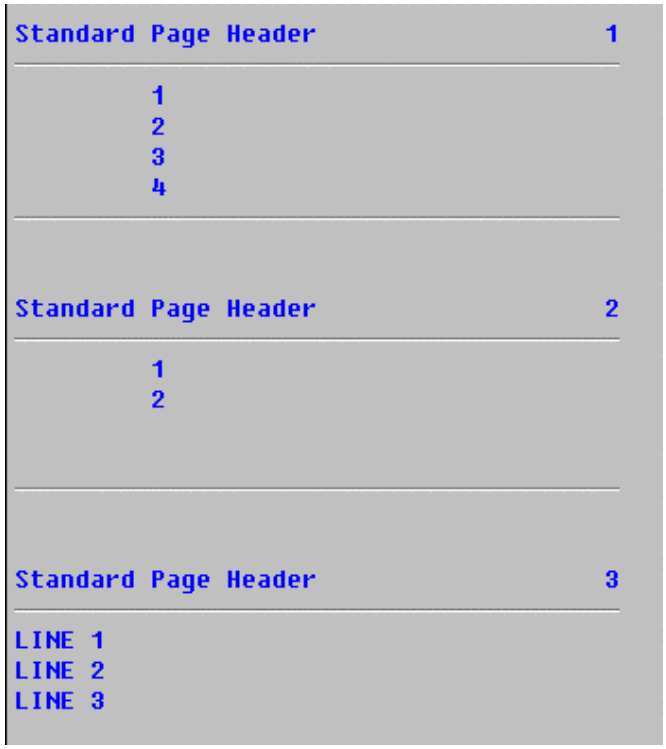
The RESERVE statement thus defines a block of lines that must be output as a whole. To find out which additional practical effects a block of lines may have, see Specifying a Relative Position [Page 829].

```
REPORT demo_list_reserve LINE-SIZE 40 LINE-COUNT 8(2).

END-OF-PAGE.

  ULINE.

START-OF-SELECTION.

  DO 4 TIMES.
    WRITE / sy-index.
  ENDDO.

  DO 2 TIMES.
    WRITE / sy-index.
  ENDDO.
```

Programming Page Breaks

```
RESERVE 3 LINES.
WRITE: / 'LINE 1',
       / 'LINE 2',
       / 'LINE 3'.
```

The list header of the standard page header of this sample program is defined as 'Standard Page Header'. The REPORT statement determines the page length to be eight lines. Two of them are used for the standard page header, another two are reserved for the page footer. The page footer consists of a horizontal line and a blank line. Thus, for outputting the actual list, four lines per page remain. The first DO loop fills these four lines. Then the END-OF-PAGE event occurs, after which the system automatically starts a new page. After the second DO loop, the RESERVE statement triggers the END-OF-PAGE event and a page break, since the number of free lines left on the page is less than three. The output appears as follows:



The three lines on page 3 form a block of lines.

Standard Page Headers of Individual Pages

The standard page header consists of list and column headers. To influence the representation of these individual components of the standard page header, use the following options of the NEW-PAGE statement:

Syntax

```
NEW-PAGE [NO-TITLE|WITH-TITLE] [NO-HEADING|WITH-HEADING].
```

You can switch the standard heading for all subsequent pages on or off using the WITH-TITLE or NO-TITLE additions respectively.

You can switch the column headings for all subsequent pages on or off using the WITH-HEADING or NO-HEADING additions respectively.

For information on basic and secondary lists, see Interactive Lists [Page 854].

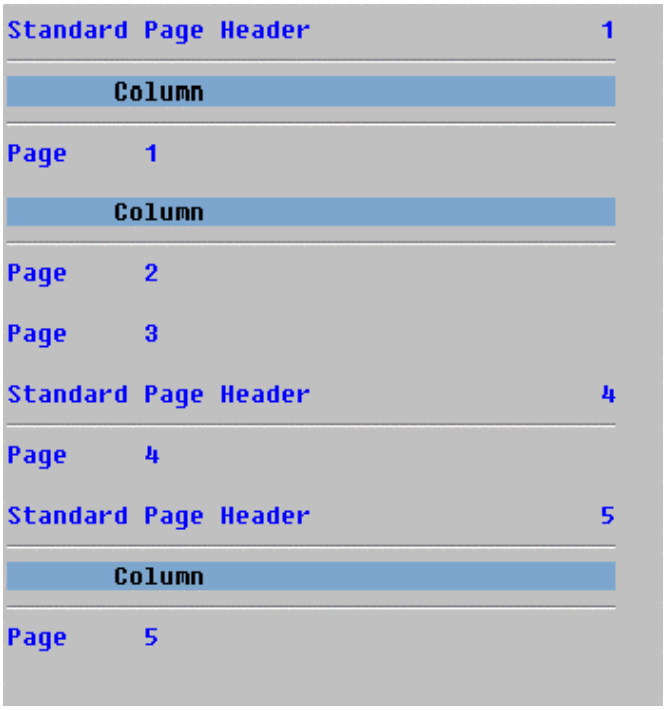
Even if you suppress the standard page header by using the NO STANDARD PAGE HEADING option of the REPORT statement, you can activate the display of the individual components using WITH-TITLE and WITH-HEADING.

The NEW-PAGE statement does not affect the display of a page header that you define in the event TOP-OF-PAGE, since this event is processed on the new page (see Individual Page Header [Page 796]).

```
REPORT demo_list_new_page_options LINE-SIZE 40.

WRITE: 'Page', sy-pagno.

NEW-PAGE NO-TITLE.
WRITE: 'Page', sy-pagno.

NEW-PAGE NO-HEADING.
WRITE: 'Page', sy-pagno.

NEW-PAGE WITH-TITLE.
WRITE: 'Page', sy-pagno.

NEW-PAGE WITH-HEADING.
WRITE: 'Page', sy-pagno.
```

This program creates five pages with different page headers. In the text elements, the list header is defined as 'Standard Page Header', and the column header as 'Column'.

Standard Page Headers of Individual Pages



Pages 1 and 5 contain the complete standard page header. On page 2, the list header is missing. On page 3, the entire page header is suppressed. On page 4, the column header is left out.

Page length of individual pages

To determine the page length of each page individually, use the NEW-PAGE statement:

Syntax

```
NEW-PAGE LINE-COUNT <length>.
```

This statement determines the page length of the subsequent pages as <length>. <length> can be a variable. If you set <length> to zero, the system uses the standard page length (Structure of the Standard List [Page 790]). The page header is part of the page and thus of the page length.

You cannot use NEW-PAGE to create or change a page footer. A page footer defined in the REPORT statement (see Determining the Page Length [Page 801]) is kept as such, independent of a NEW-PAGE statement.

For the actual list output, <length> minus the page header length and the page footer length is available.

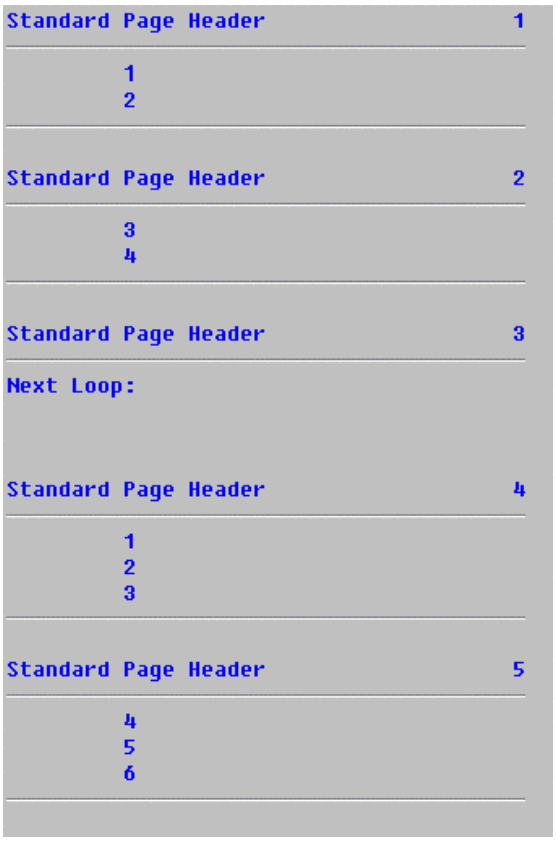
When using the LINE-COUNT option of the NEW-PAGE statement, refer to the notes in Determining the Page Length [Page 801].

To adapt the page length to the current window length, set <length> to SY-SROWS. The SY-SROWS system field contains the number of lines of the current window.

```
REPORT demo_list_new_page_line_c_1 LINE-SIZE 40 LINE-COUNT
0(1).
END-OF-PAGE.
  ULINE.
START-OF-SELECTION.
  NEW-PAGE LINE-COUNT 5.
  DO 4 TIMES.
    WRITE / sy-index.
  ENDDO.
  WRITE: / 'Next Loop:'.
  NEW-PAGE LINE-COUNT 6.
  DO 6 TIMES.
    WRITE / sy-index.
  ENDDO.
```

This program creates five pages of different lengths. The list header text element is defined as 'Standard Page Header' . The REPORT statement reserves one line of each page for the page footer. The page footer is defined in the END-OF-PAGE event as a horizontal line. The first NEW-PAGE statement sets the page length to 5, the second to 6.

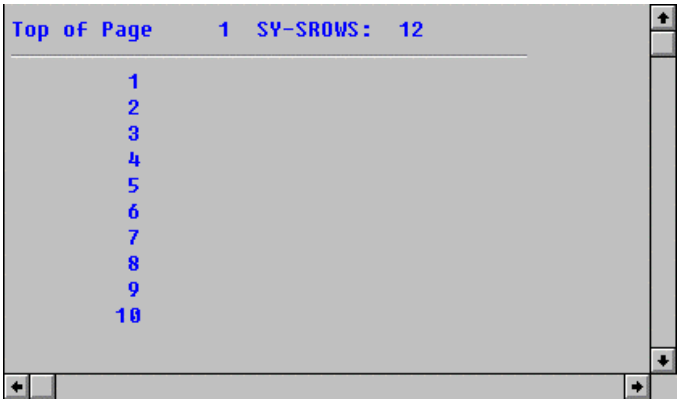
Page length of individual pages



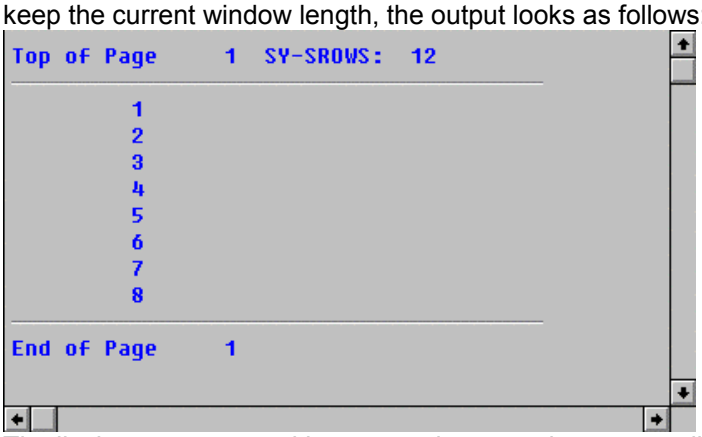
The first NEW-PAGE statement does not start a new page, since no output was written to the list before. The standard page header uses two lines of each page for the list header. The page footer uses one line. For the first DO loop, two lines per page can be used to WRITE output. All page breaks within the DO loop occur **automatically** as soon as the list processing reaches the page footer. The system then displays the page footer. The second NEW-PAGE creates a page break from page 3 to 4. Here, the END-OF-PAGE event is not processed. For the second DO loop, three lines per page can be used to WRITE output. Page breaks again occur **automatically.** The page footer appears.

```
REPORT demo_list_new_page_line_c_2 NO STANDARD PAGE HEADING
                                   LINE-SIZE 40 LINE-COUNT
0(2).
TOP-OF-PAGE.
  WRITE: 'Top of Page', sy-pagno,
           'SY-SROWS:', sy-srows.
  ULINE.
END-OF-PAGE.
  ULINE.
  WRITE: 'End of Page', sy-pagno.
START-OF-SELECTION.
* NEW-PAGE LINE-COUNT SY-SROWS.
  DO 100 TIMES.
    WRITE / sy-index.
  ENDDO.
```

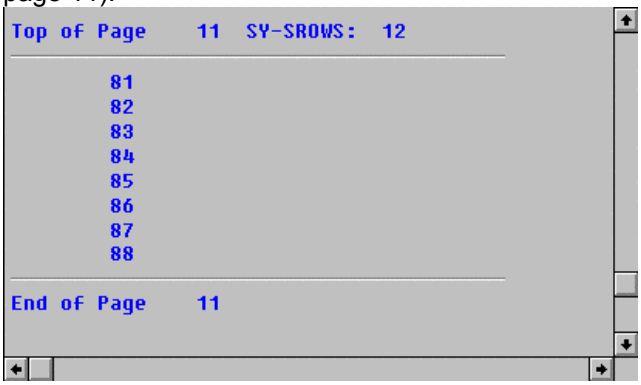
This program creates one single endless page, since the NEW-PAGE statement is marked as comment:



The system displays as many lines as possible in the current window, which has a length of 12 lines. In the figure above, the 12 lines are made up of two self-defined header lines and 10 lines of the actual list. When you scroll vertically, the page header remains visible. If you remove the asterisk in front of the NEW-PAGE statement and keep the current window length, the output looks as follows:



The list is now separated into several pages where, according to SY-SROWS, each page is 12 lines long. Of these 12 lines, two are reserved for the page header and two for the footer. In this list, the user can scroll explicitly using *Next page* (for example, to page 11):



Page Width of List Levels

You **cannot** change the width of individual pages within a list level. You can only change the width of **all** pages of a new list level. To do so, use the NEW-PAGE statement:

Syntax

```
NEW-PAGE LINE-SIZE <width>.
```

All list levels starting from the new page have a width of <width> instead of the one specified in the REPORT statement. If you set <width> to 0, the system uses the width of the standard list (see Structure of the Standard List [Page 790]).

If you set <width> to SY-SCOLS, you can adapt the width of the new list level to the window width, even if the window is smaller than the standard window. The SY-SCOLS system field contains the number of characters of a line of the current window.

Within a list level, that is, if the next page is not the beginning of a new list level, the system ignores the LINE-SIZE option.

For information on how to create new list levels, see Interactive Lists [Page 854].

Scrolling in Lists

From within the program, you can scroll through lists vertically and horizontally. Use the SCROLL keyword. Scrolling from within the program makes sense, for example, if you want to scroll to certain pages as a reaction to user input.

You can only use the SCROLL statement for completed lists. If you use SCROLL before the first output statement of a list, it does not affect the list. If you use SCROLL after the first output statement of a list, it affects the entire list, that is, all subsequent output statements as well.

After each SCROLL statement, you can query SY-SUBRC to see whether the system succeeded. SY-SUBRC is 0 if the system successfully scrolled, and 4 if scrolling was not possible, because it would exceed the list boundaries. If you are working with several list levels, SY-SUBRC may also be 8, indicating that the list level you specified does not exist (see Scrolling Through Interactive Lists [Page 887]).

The SCROLL statement allows the following options:

Vertical Scrolling

Scrolling Window by Window [Page 816]

Scrolling by Pages [Page 817]

Horizontal Scrolling

Scrolling to the List's Margins [Page 819]

Scrolling by Columns [Page 820]

Scrolling Window by Window

To scroll through a list vertically by the size of the current window and independent of the page length, use the following statement:

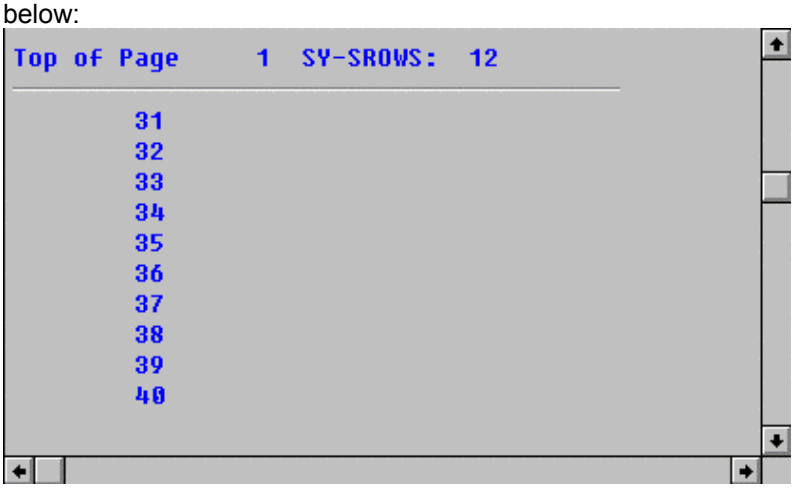
Syntax

```
SCROLL LIST FORWARD|BACKWARD [INDEX <idx>].
```

If you do not use the INDEX addition, the statement scrolls forward or backward by one whole window in the current list. If you use the INDEX <idx> addition, the system scrolls in the list with the level <idx>. For more information on scrolling in list levels, see Scrolling through Interactive Lists [Page 887].

```
REPORT demo_list_scroll_1 NO STANDARD PAGE HEADING LINE-SIZE
40.
TOP-OF-PAGE.
  WRITE: 'Top of Page', sy-pagno, 'SY-SROWS:', sy-srows.
  ULINE.
START-OF-SELECTION.
  DO 100 TIMES.
    WRITE / sy-index.
  ENDDO.
  DO 3 TIMES.
    SCROLL LIST FORWARD.
  ENDDO.
```

This executable program (report) creates a list of one endless page. Within the DO loop, the system executes the SCROLL statement three times. If the current window has a length of 12 lines (stored in SY-SROWS), the output of the program appears as below:



Note that the actual list is scrolled by SY-SROWS minus the number of header lines. The user can continue scrolling into both directions.

Scrolling by Pages

To scroll a list by pages, that is, scroll vertically depending on the page length, the SCROLL statement offers several options.

Scrolling to Specific Pages

To scroll to specific pages, use the TO option of the SCROLL statement:

Syntax

```
SCROLL LIST TO FIRST PAGE | LAST PAGE | PAGE <pag>
        [INDEX <idx>] [LINE <lin>].
```

Without the INDEX option, the statement scrolls the current list to the first, to the last, or to the page numbered <pag>. If you specify the INDEX option, the system scrolls the list of list level <idx>. For more information on list levels, see Interactive Lists [Page 854].

If you specify the LINE option, the system displays the page to which it scrolls starting from line <lin> of the actual list. It does not count the page header lines.

Scrolling by a Specific Number of Pages

To scroll a list by a specific number of pages, use the following options of the SCROLL statement:

Syntax

```
SCROLL LIST FORWARD | BACKWARD <n> PAGES [INDEX <idx>].
```

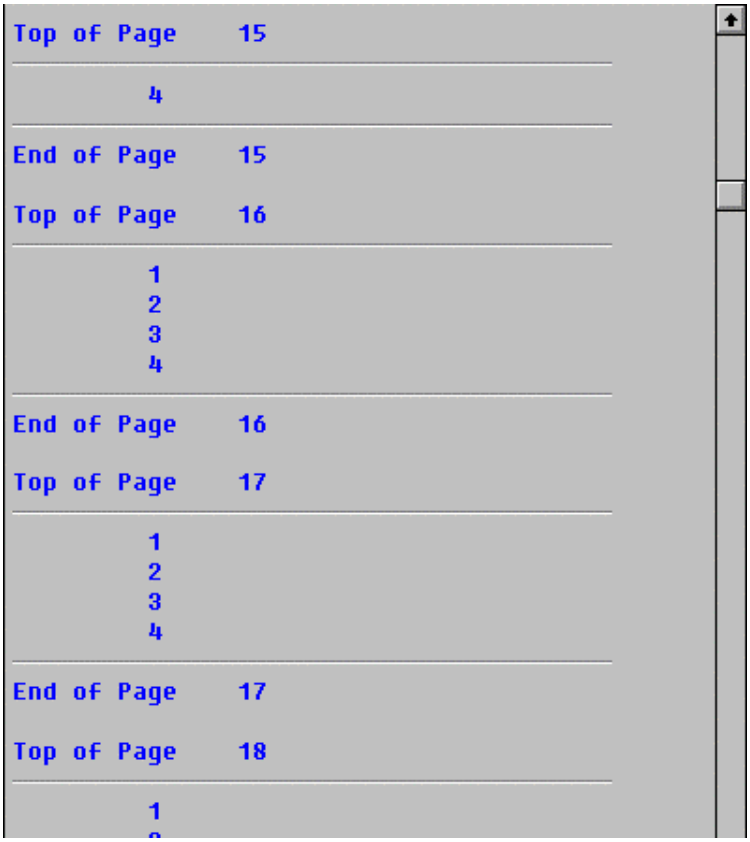
If you do not specify the INDEX option, the statement scrolls forward or backward <n> pages. The INDEX option specifies a particular list level as described above.

```
REPORT demo_list_scroll_2 NO STANDARD PAGE HEADING
                          LINE-SIZE 40 LINE-COUNT 8(2).

DATA: pag TYPE i VALUE 15,
      lin TYPE i VALUE 4.

TOP-OF-PAGE.

  WRITE: 'Top of Page', sy-pagno.
  ULINE.

END-OF-PAGE.

  ULINE.
  WRITE: 'End of Page', sy-pagno.

START-OF-SELECTION.

  DO 100 TIMES.
    DO 4 TIMES.
      WRITE / sy-index.
    ENDDO.
  ENDDO.
```

Scrolling by Pages

```
SCROLL LIST TO PAGE pag LINE lin.
```

This program creates a list of 100 pages with 8 lines per page. On each page, four lines are used for page header and page footer. Due to the SCROLL statement, the output of the program appears as follows:



The list display starts at page 15. Due to the LINE option, the first three lines of the actual list are scrolled beneath the page header.

Scrolling to the Margins of the List

To scroll horizontally to the left or right margin of a list, use the following options of the SCROLL statement:

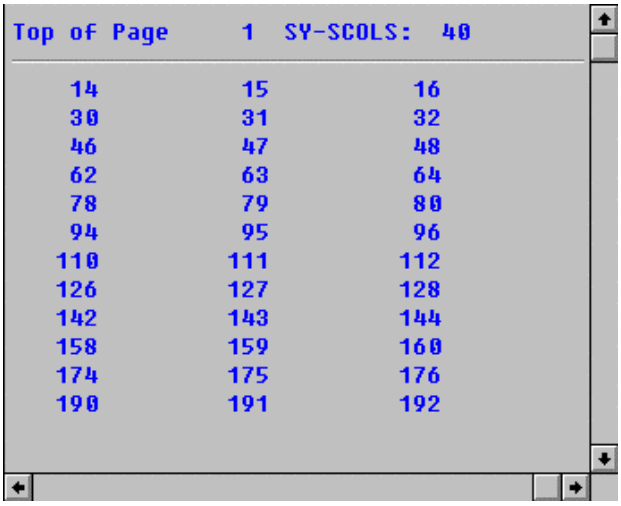
Syntax

```
SCROLL LIST LEFT | RIGHT [INDEX <idx>].
```

If you do not specify the INDEX option, the statement scrolls to the left or right margin of the current list. If you specify the INDEX option, the system scrolls the list of list level <idx>. For more information on list levels, see Interactive Lists [Page 854].

```
REPORT demo_list_scroll_3 NO STANDARD PAGE HEADING LINE-SIZE
200.

TOP-OF-PAGE.

  WRITE: AT 161 'Top of Page', sy-pagno,
                'SY-SCOLS:', sy-scols.

  ULINE.

START-OF-SELECTION.

  DO 200 TIMES.
    WRITE sy-index.
  ENDDO.

  SCROLL LIST RIGHT.
```

This program creates a one-page list with a width of 200. If the current window width (stored in SY-SCOLS) equals 40, the output of the program looks as follows:



The list display is scrolled to the right margin. The user can now use the scroll bar to scroll to the left.

Scrolling by Columns

To scroll a list horizontally by columns, the SCROLL statement offers several options. A column in this case means one character of the list line.

Scrolling to Specific Columns

To scroll to specific columns, use the TO COLUMN option of the SCROLL statement:

Syntax

```
SCROLL LIST TO COLUMN <col> [INDEX <idx>].
```

If you do not specify the INDEX option, the system displays the current list starting from column <col>. If you specify the INDEX option, the system scrolls the list of list level <idx>. For more information on list levels, see Interactive Lists [Page 854].

Scrolling by a Specific Number of Columns

To scroll a list by a specific number of columns, use the following option of the SCROLL statement:

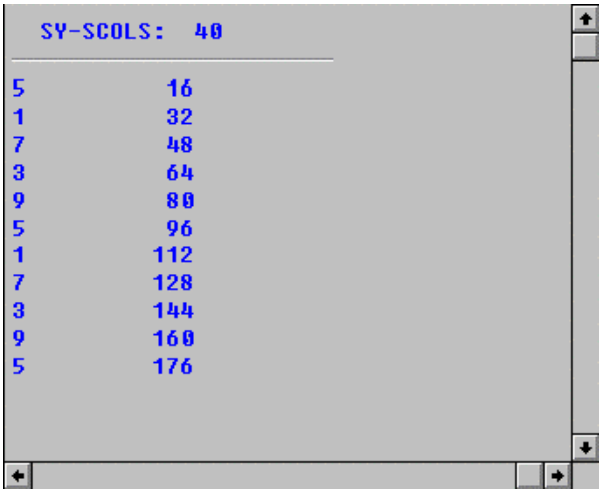
Syntax

```
SCROLL LIST LEFT | RIGHT BY <n> PLACES [INDEX <idx>].
```

If you do not specify the INDEX option, the statement scrolls forward or backward <n> columns. The INDEX option specifies a particular list level as described above.

```
REPORT demo_list_scroll_4 NO STANDARD PAGE HEADING LINE-SIZE
200.

TOP-OF-PAGE.

  WRITE: AT 161 'Top of Page', sy-pagno,
                  'SY-SCOLS:', sy-scols.
  ULINE.

START-OF-SELECTION.

  DO 200 TIMES.
    WRITE sy-index.
  ENDDO.

  SCROLL LIST TO COLUMN 178.
```

This program creates a one-page list with a width of 200. If the current window width (stored in SY-SCOLS) equals 40, the output of the program looks as follows:



The list is displayed starting from column 178. The user can now scroll to the left of the list.

Defining Where the User Can Scroll on a Page

You can restrict the area of a page that can be scrolled horizontally by the user with the scrollbars or from within the program. You can also combine the following two techniques.

Excluding Lines from Horizontal Scrolling

To exclude a line (for example, a header or comment line) from horizontal scrolling, define the line feed for that line as follows:

Syntax

```
NEW-LINE NO-SCROLLING.
```

The line following the statement cannot be scrolled horizontally. However, it can be scrolled vertically.

To undo the above statement, use:

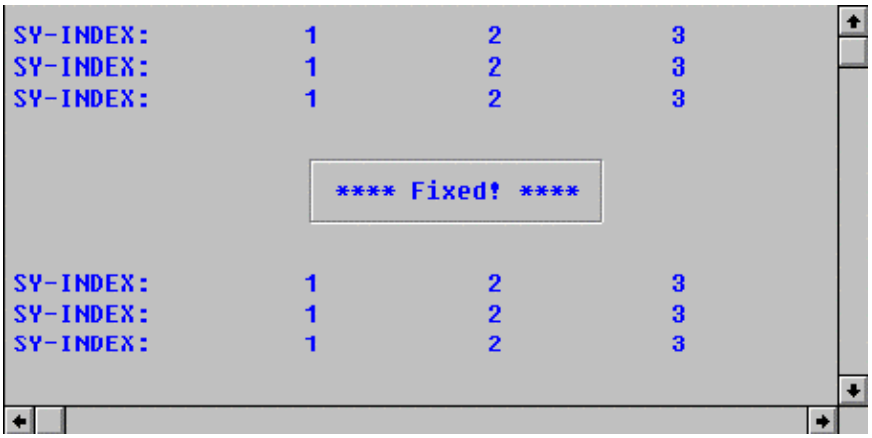
Syntax

```
NEW-LINE SCROLLING.
```

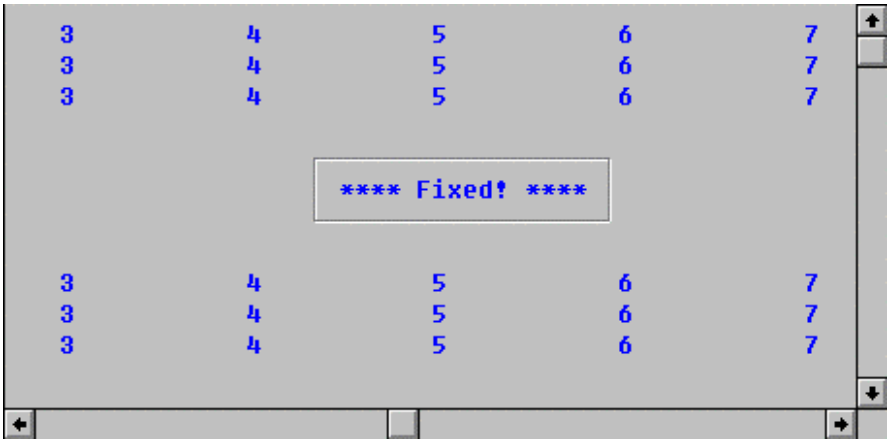
This statement only makes sense if no line was output after NEW-LINE NO-SCROLLING.

```
REPORT sapmztst NO STANDARD PAGE HEADING
                LINE-COUNT 3 LINE-SIZE 140.

START-OF-SELECTION.

DO 3 TIMES.
  WRITE: / 'SY-INDEX:'.
  DO 10 TIMES.
    WRITE sy-index.
  ENDDO.
ENDDO.

NEW-LINE NO-SCROLLING.
ULINE AT 20(20).
NEW-LINE NO-SCROLLING.
WRITE AT 20 '| **** Fixed! **** |'.
NEW-LINE NO-SCROLLING.
ULINE AT 20(20).

DO 3 TIMES.
WRITE: / 'SY-INDEX:'.
  DO 10 TIMES.
    WRITE sy-index.
  ENDDO.
ENDDO.
```

This program creates three pages of three lines each without page header or footer. The three lines of the second page cannot be scrolled due to NEW-LINE NO-SCROLLING. The program output looks like this:



If the user scrolls to the right, the output may look like this:



The lines of the first and third pages scroll past the fixed lines.

Left Boundary for Horizontal Scrolling

To determine the left boundary of the horizontally scrollable area, use:

Syntax

```
SET LEFT SCROLL-BOUNDARY [COLUMN <col>].
```

If you do not use the COLUMN option, the left boundary of the scrollable area of the current page is set to the current output position; if you use the COLUMN option, the left boundary is set to position <col>. Now, only the part to the right of this area can be scrolled horizontally.

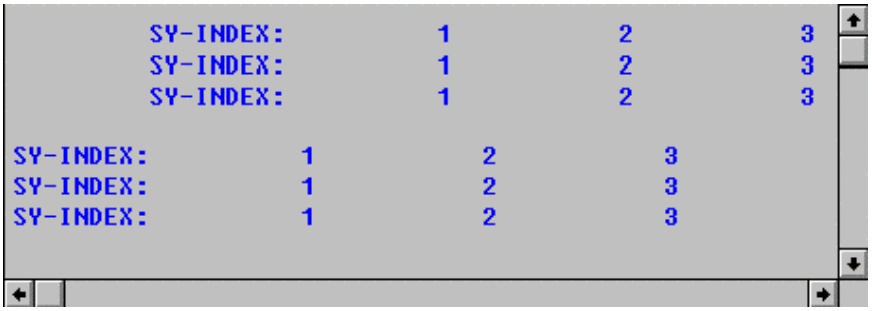
The above statement applies to the entire current page, and only to it. You must repeat the statement for each new page, otherwise the system uses the default value (left list margin).

To set the same scrollable area for all pages of a list, you can execute the statement, for example, at the TOP-OF-PAGE event.

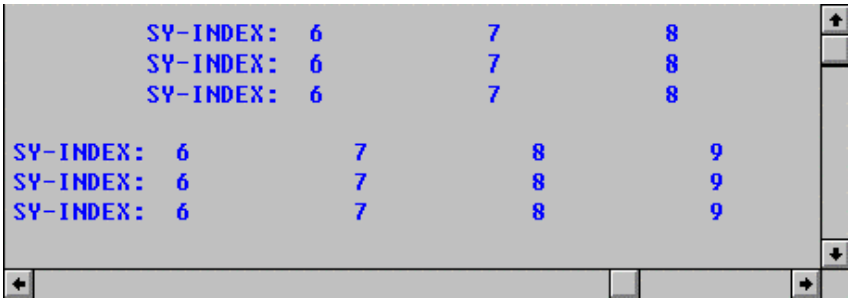
Defining Where the User Can Scroll on a Page

```
REPORT sapmztst NO STANDARD PAGE HEADING
                LINE-COUNT 3 LINE-SIZE 140.

START-OF-SELECTION.

DO 3 TIMES.
  WRITE: /10 'SY-INDEX:'.
  DO 10 TIMES.
    WRITE sy-index.
  ENDDO.
ENDDO.

SET LEFT SCROLL-BOUNDARY COLUMN 20.

DO 3 TIMES.
WRITE: / 'SY-INDEX:'.
  DO 10 TIMES.
    WRITE sy-index.
  ENDDO.
ENDDO.

SET LEFT SCROLL-BOUNDARY COLUMN 10.
```

This program creates two pages of three lines each without page header or footer. The first SET statement affects the first page, since the automatic page break does not occur until the first WRITE statement of the second DO loop. The second SET statement affects the second page. The program output looks like this:



If the user scrolls to the right, the output may look like this:



The scrollable area of the first page disappears at a different position beneath the fixed area at the left margin than the scrollable area of the second page.

Laying Out List Pages

The layout of a list page determines how clearly structured, and therefore easy to read, a list is. It is not the amount of information gathered on one page that is important, but the way the information is presented. The human eye can cope much better with small blocks of data. And it is equally important that columns or lines containing a new block of information are separated visually from the preceding blocks. When laying out a list page, you should use several blanks or vertical lines to separate individual columns. Before outputting lines that contain a new item of information, draw a blank or underscore line.

The following topics explain the possibilities ABAP offers for laying out list pages.

Positioning the Output [Page 826]

Formatting Output [Page 832]

Special Output Formats [Page 842]

Creating Blank Lines [Page 799]

Drawing Lines, Frames, and Grids [Page 846]

Determining Which Part of a Page to Scroll Horizontally [Page 822]

Positioning the Output

You can position the output of WRITE and ULINE statements anywhere on the current page. The WRITE, SKIP, or ULINE statements following a position specification may overwrite existing output. For the current output position, refer to the system fields

- SY-COLNO (for the current column)
- SY-LINNO (for the current line)

You can use these system fields to navigate on the page.

ABAP offers a number of keywords to change the absolute as well as the relative output positions. See the following topics:

Absolute Positioning [Page 827]

Relative Positioning [Page 829]

SAP intends to allow only read access to the system fields SY-COLNO and SY-LINNO. Therefore, to position your output, **only** use the statements described in these topics. **Do not** position output by directly assigning values to these system fields. In that case, SAP cannot guarantee that the contents of the system fields are consistent, since such an assignment does not trigger a plausibility check. Even though it is currently possible to assign a column number to SY-COLNO that is outside the page, it doesn't make sense to do so.

Absolute Positioning

When you specify an absolute position, the subsequent output is written to the screen starting at fixed lines and columns.

Horizontal Positioning

To specify a horizontal output position, ABAP offers two ways:

The AT option of the WRITE and ULINE statements (see Positioning WRITE Output on the List [Page 778]) and the POSITION statement. The syntax of the POSITION statement is:

Syntax

```
POSITION <col>.
```

This statement sets the horizontal output position and the SY-COLNO system field to <col>. If <col> lies outside the page, the subsequent output statements are ignored.

The system writes an output following the POSITION statement or a WRITE statement formatted using AT to the specified position, regardless of whether there is enough space. That part of the output that does not fit onto the line, is truncated. Other WRITE output then starts on the next line.

Vertical Positioning

You specify vertical output positions as follows:

Syntax

```
SKIP TO LINE <n>.
```

This statement sets the vertical output position and the SY-LINNO system field to <lin>. If <lin> does not lie between 1 and the page length, the system ignores the statement.

When you use LINE, the system also counts page header and page footer lines. Make sure that you do not unintentionally overwrite header or footer lines.

Positioning Output Beneath the Page Header

To position output on the first line following the entire page header, use the BACK statement:

Syntax

```
BACK.
```

If this statement does not follow a RESERVE statement, the subsequent output appears beneath the page header. The system sets SY-COLNO to 1 and SY-LINNO according to the length of the page header. In combination with the RESERVE statement, another rule applies (see Specifying a Relative Position [Page 829]).

If the BACK statement is executed at the TOP-OF-PAGE event, the system does not set the output position to beneath the entire page header, but only to beneath the standard page header. Any output written now overwrites the self-defined page header specified at TOP-OF-PAGE.

Example for Absolute Positioning

```
REPORT demo_list_position NO STANDARD PAGE HEADING LINE-SIZE
60.
DATA: x(3) TYPE c, y(3) TYPE c.
x = sy-colno. y = sy-linno.
TOP-OF-PAGE.
  WRITE: 'Position of Header: ', x, y.
  ULINE.
START-OF-SELECTION.
  SKIP TO LINE 10.
  POSITION 20.
  x = sy-colno. y = sy-linno.
  WRITE: '* <- Position', x, y.
  SKIP TO LINE 12.
  ULINE AT 20(20).
  BACK.
  x = sy-colno. y = sy-linno.
  WRITE: 'Position after BACK:', x, y.
```

This program creates the following list page:

```
Position of Header:   1   1
________________________________________________________________

Position after BACK:  1   3




                    * <- Position 20  10

                    ____________________
```

The system assigns the original values of SY-COLNO and SY-LINNO to the fields X and Y. Note that this assignment actually takes place at the START-OF-SELECTION event (see Defining Processing Blocks [Page 940]). The original output position is the position of the first header line. The output is written there. SKIP TO LINE and POSITION place an asterisk '*' in column 20, line 10. SKIP TO LINE and AT produce underlining. Finally, BACK sets the output position to column 1, line 3 beneath the two-line page header.

Relative Positioning

Relative positions refer to output previously written to the list. Several relative positionings occur automatically. When you use WRITE without specifying a position, the output appears after the previous output, separated by a blank column. If there is not enough space in the current line, a line feed occurs. ULINE and SKIP statements without positioning produce a line feed.

To program relative positioning, use either the SY-COLNO and SY-LINNO system fields together with the statements described in Absolute Positioning [Page 827] or the statements for relative positioning described below.

Producing a Line Feed

To produce a line feed, use either the forward slash in the AT option of WRITE or ULINE or the NEW-LINE statement.

Syntax

```
NEW-LINE.
```

This statement positions the next output in a new line. SY-COLNO is set to 1, and SY-LINNO is increased by one. The system only executes the statement if output was written to the screen since the last line feed. NEW-LINE does not create a blank line. To create a blank line, use the SKIP statement (see Creating Blank Lines [Page 799]).

An automatic line feed occurs at the NEW-PAGE statement and at the beginning of an event.

Positioning Output Underneath Other Output

You can position a WRITE output in the same column as a previous WRITE output. Use the formatting option UNDER of the WRITE statement:

Syntax

```
WRITE <f> UNDER <g>.
```

The system starts outputting <f> in the same column from which field <g> started. This statement is not limited to the current page, that is, <g> must not appear on the same page.

You must fix the vertical position yourself. Any existing output is overwritten.

The reference field <g> must be written exactly as in the corresponding WRITE statement, including all specifications such as offset (see Specifying Offset Values for Data Objects [Page 196]). If <g> is a text symbol (see Text Symbols [Page 121]), the system determines the reference field from the number of the text symbol.

Positioning Output in the First Line of a Line Block

To set the next output line to the first line of a block of lines defined with the RESERVE statement (see Programming Page Breaks [Page 806]), use the BACK statement as follows:

Relative Positioning

Syntax

```
RESERVE.
  .....
BACK.
```

If BACK follows RESERVE, the subsequent output appears in the first line written after RESERVE. You can use this statement, for example, to jump back to a specific line after writing an output from within a loop.

Examples for Relative Positioning

The first example shows how to create a column-based list by means of a self-defined page header.

```
REPORT demo_list_position_relative_1 NO STANDARD PAGE HEADING
                                     LINE-SIZE 80 LINE-COUNT
7.

DATA: h1(10) TYPE c VALUE '    Number',
      h2(10) TYPE c VALUE '    Square',
      h3(10) TYPE c VALUE '      Cube',
      n1 TYPE i, n2 TYPE i, n3 TYPE i,
      x TYPE i.

TOP-OF-PAGE.

  x = sy-colno + 8.  POSITION x. WRITE h1.
  x = sy-colno + 8.  POSITION x. WRITE h2.
  x = sy-colno + 8.  POSITION x. WRITE h3.
  x = sy-colno + 16. POSITION x. WRITE sy-pagno.
  ULINE.

START-OF-SELECTION.

  DO 10 TIMES.
    n1 = sy-index. n2 = sy-index ** 2. n3 = sy-index ** 3.
    NEW-LINE.
    WRITE: n1 UNDER h1,
           n2 UNDER h2,
           n3 UNDER h3.
  ENDDO.
```

This program creates a two-page list. In the self-defined page header, column headers are positioned relatively by using the SY-COLNO system field and the POSITION statement. The actual list output is positioned underneath the fields of the header line using the UNDER option of the WRITE statement. Line feeds are specified using NEW-LINE. The output appears as follows:

```
Number          Square          Cube                    1
------------------------------------------------------------
     1               1             1
     2               4             8
     3               9            27
     4              16            64
     5              25           125

Number          Square          Cube                    2
------------------------------------------------------------
     6              36           216
     7              49           343
     8              64           512
     9              81           729
    10             100         1.000
```

The different output positions of the individual fields result from the ABAP default of representing character strings as left-justified and numeric fields as right-justified. To influence justification, use the formatting options LEFT-JUSTIFIED, RIGHT-JUSTIFIED, and CENTERED of the WRITE statement (see Formatting Options [Page 780]).

The second example shows the effect of the BACK statement following RESERVE.

```
REPORT demo_list_position_relative_2
       NO STANDARD PAGE HEADING LINE-SIZE 40.

DATA x TYPE i.

WRITE 'Some numbers:' NO-GAP.
x = sy-colno.
ULINE AT /(x).

RESERVE 5 LINES.

DO 5 TIMES.
  WRITE / sy-index.
ENDDO.

x = sy-colno.

BACK.
WRITE AT x '   <- Start of Loop'.
```

This program creates the following output:

```
Some numbers:
-------------
        1     <- Start of Loop
        2
        3
        4
        5
```

After the first two lines have been displayed, the RESERVE statement is used. The next five lines are defined as a block. The output following BACK is written to the first line of the block. Note how the SY-COLNO system field is used to underline the first line and to position the last WRITE output.

Formatting Output

To format list output, ABAP offers several formatting options.

The formatting options of the WRITE statement are described in Formatting Options [Page 780].

Other important formatting options, for example, to determine the color of the output or to make list fields ready for input, are the formatting options of the FORMAT statement. They are described in the subsequent topics. You can use all options of the FORMAT statement as formatting options of the WRITE statement.

The FORMAT Statement [Page 833]

Special Output Formats [Page 842]

Lines on Lists [Page 846]

The FORMAT Statement

To set formatting options statically in the program, use the FORMAT statement as follows:

Syntax

```
FORMAT <option1> [ON|OFF] <option2> [ON|OFF] ....
```

The formatting options <option$_i$> set in the FORMAT statement apply to all subsequent output until they are turned off using the OFF option. The ON option to turn on a formatting option is optional, that is, you can leave it out.

To set the formatting options dynamically at runtime, use the FORMAT statement as follows:

Syntax

```
FORMAT <option1> = <var1> <option2> = <var2> ....
```

The system interprets the variables <var$_i$> as numbers. Therefore, they should be of data type I. If the contents of <var$_i$> is zero, the variable has the same effect as the OFF option. If the contents of <var$_i$> is unequal to zero, the variable either has the same effect as the ON option or, together with the COLOR option, acts like the corresponding color number (see Colors in Lists [Page 834]).

If you use the same formatting options for a WRITE statement that follows the FORMAT statement, the settings of the WRITE statement overwrite the corresponding settings of the FORMAT statement for the current output.

For each new event, the system resets all formatting options to their default values. For a list of events, see Events and their Event Keywords [Page 953]. All formatting options have the default value OFF, except the INTENSIFIED option (see Colors in Lists [Page 834]).

To set all formatting options to OFF in one go, use:

Syntax

```
FORMAT RESET.
```

The following sections describe the available formatting options.

Colors in Lists [Page 834]

Enabling Fields for Input [Page 839]

Outputting Fields as Hotspots [Page 840]

Colors in Lists

The options COLOR, INTENSIFIED, and INVERSE of the FORMAT statement influence the colors of the output list.

To set colors in the program, use:

Syntax

```
FORMAT COLOR <n> [ON] INTENSIFIED [ON|OFF] INVERSE [ON|OFF].
```

To set colors at runtime, use:

Syntax

```
FORMAT COLOR = <c> INTENSIFIED = <int> INVERSE = <inv>.
```

These formatting options do not apply to horizontal lines created by ULINE. They have the following functions:

- COLOR sets the color of the line background. If, in addition, INVERSE ON ist set, the system changes the foreground color instead of the background color.

  For <n> you can set either a color number or a color specification. Instead of color number 0, however, you must use OFF. If you set the color numbers at runtime, all values of <c> that are less than zero or greater than seven, lead to undefined results. The following table summarizes the different possibilities:

	<n>	<c>	Color	Intended for
OFF	or COL_BACKGROUND	0	depends on GUI	background
1	or COL_HEADING	1	grey-blue	headers
2	or COL_NORMAL	2	light grey	list bodies
3	or COL_TOTAL	3	yellow	totals
4	or COL_KEY	4	blue-green	key columns
5	or COL_POSITIVE	5	green	positive threshold value
6	or COL_NEGATIVE	6	red	negative threshold value
7	or COL_GROUP	7	violet	Control levels

  The default setting is COLOR OFF.

- INTENSIFIED determines the color palette for the line background.

  With one exception (COLOR OFF), the color palette for the line background specified above can be intensified or normal. The default setting is INTENSIFIED ON. For COLOR OFF, the system changes the foreground color instead of the background color. If, in addition, INVERSE ON is set, then INTENSIFIED OFF is without effect (again with the exception of COLOR OFF).

- INVERSE influences the foreground color.

  With one exception (COLOF OFF), the system takes the COLOR specified from an inverse color palette and uses it as foreground color. The background color remains

unchanged. For COLOR OFF, INVERSE has no effect, since this would set the foreground and the background to the same color.

The following statements have the same effect:

FORMAT INTENSIFIED ON. and `SUMMARY.`

FORMAT INTENSIFIED OFF and `DETAIL.`

For reasons of better readability, SAP recommends that you always use the FORMAT statement.

The following examples show the colors possible in lists and how to use them.

For another demonstration of colors in lists, call the executable program SHOWCOLO in any system.

Demonstrating the Colors Available in Lists

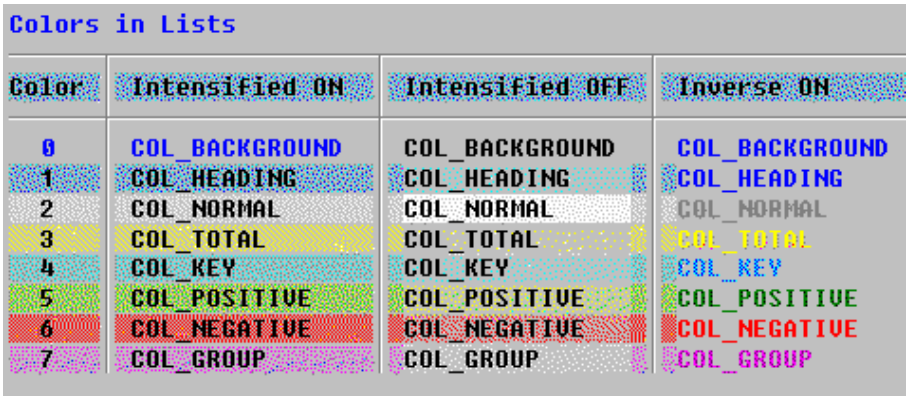
The following example shows the different combinations of the color formatting options:

```
REPORT demo_list_format_color_1.

DATA i TYPE i VALUE 0.
DATA col(15) TYPE c.

WHILE i < 8.

  CASE i.
    WHEN 0. col = 'COL_BACKGROUND '.
    WHEN 1. col = 'COL_HEADING    '.
    WHEN 2. col = 'COL_NORMAL     '.
    WHEN 3. col = 'COL_TOTAL      '.
    WHEN 4. col = 'COL_KEY        '.
    WHEN 5. col = 'COL_POSITIVE   '.
    WHEN 6. col = 'COL_NEGATIVE   '.
    WHEN 7. col = 'COL_GROUP      '.
  ENDCASE.

  FORMAT INTENSIFIED COLOR = i.
  WRITE: /(4) i, AT 7            sy-vline,
              col,               sy-vline,
              col INTENSIFIED OFF, sy-vline,
              col INVERSE.

  i = i + 1.

ENDWHILE.
```

In the FORMAT statement, the COLOR option for the subsequent WRITE statements is set at runtime. The other options are set individually for each WRITE statement in the program.

The output appears as shown in the following table:

Colors in Lists



The standard page header was created as a text element. In the online help, due to technical reasons, the colors of this list differ slightly from the colors of the R/3 system.

Example for Using Colors in Lists

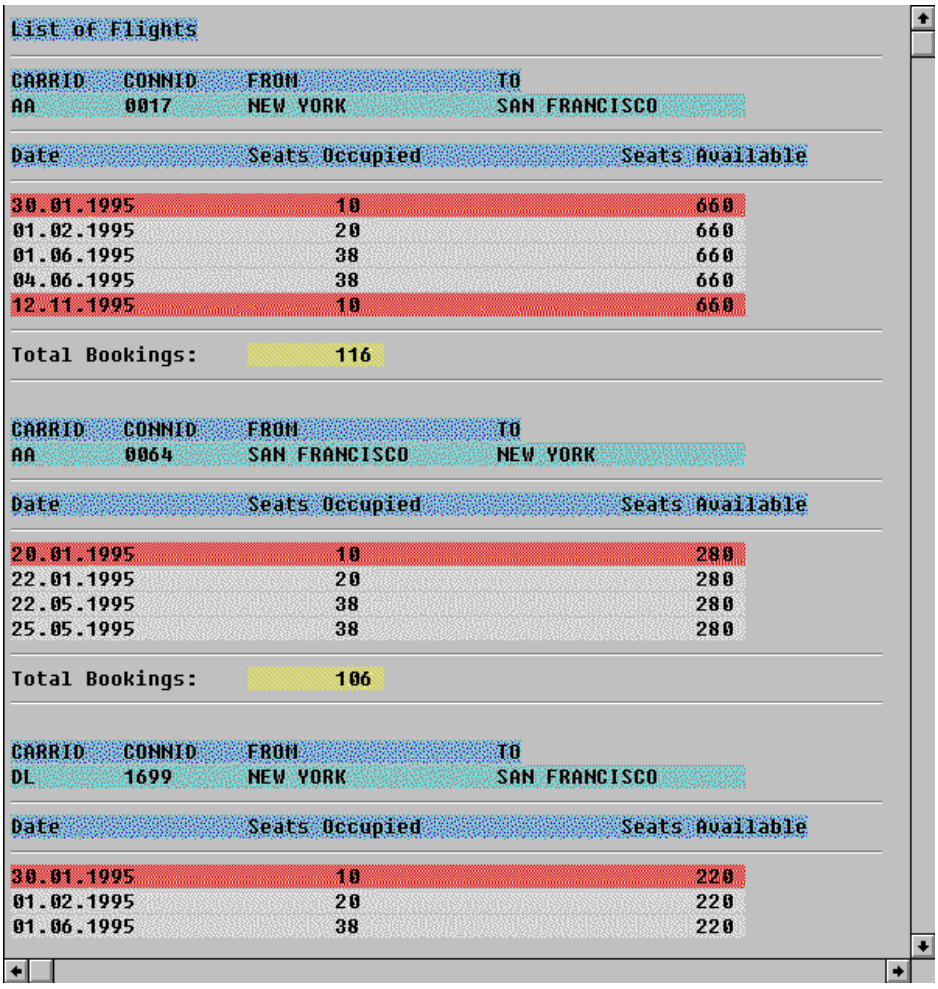
This example shows how to use colors in lists to highlight output.

The following executable program (report) is connected to the logical database F1S.

```
REPORT demo_list_format_color_2 NO STANDARD PAGE HEADING LINE-
SIZE 70.

TABLES: spfli, sflight.
DATA sum TYPE i.

TOP-OF-PAGE.

  WRITE 'List of Flights' COLOR COL_HEADING.
  ULINE.

GET spfli.

  FORMAT COLOR COL_HEADING.
  WRITE: 'CARRID', 10 'CONNID', 20 'FROM', 40 'TO'.

  FORMAT COLOR COL_KEY.
  WRITE: / spfli-carrid   UNDER 'CARRID',
           spfli-connid   UNDER 'CONNID',
           spfli-cityfrom UNDER 'FROM',
           spfli-cityto   UNDER 'TO'.
  ULINE.

  FORMAT COLOR COL_HEADING.
  WRITE: 'Date', 20 'Seats Occupied', 50 'Seats Available'.
  ULINE.

  sum = 0.

GET sflight.

  IF sflight-seatsocc LE 10.
    FORMAT COLOR COL_NEGATIVE.
  ELSE.
```

```
    FORMAT COLOR COL_NORMAL.
  ENDIF.

  WRITE: sflight-fldate   UNDER 'Date',
         sflight-seatsocc UNDER 'Seats Occupied',
         sflight-seatsmax UNDER 'Seats Available'.

  sum = sum + sflight-seatsocc.

GET spfli LATE.

  ULINE.
  WRITE: 'Total Bookings:' INTENSIFIED OFF,
         sum UNDER sflight-seatsocc COLOR COL_TOTAL.
  ULINE.
  SKIP.
```

The report creates the following output list:



All headers appear using the background color COL_HEADING. The key fields from table SPFLI use COL_KEY as background color. The list body at the event GET SFLIGHT has a different line background color (COL_NORMAL) than the list background (COL_BACKGROUND). In addition, flights where the number of

bookings falls below a certain minimum number, have a red background. The total number of bookings for each flight has a yellow background.

Note that the system resets the formatting options for each new event to the default settings (COLOR OFF, INTENSIFIED ON). For this reason, in the above program the line background of the output 'Total Bookings:' is COL_BACKGROUND again in the GET LATE event. INTENSIFIED is set to OFF to get the same foreground color as for the other output.

In the online help, due to technical reasons, the colors of this list differ slightly from the colors of the R/3 system.

Enabling Fields for Input

You can make output fields in lists input-enabled. The user can change these fields on the screen. The changes can then be printed, or you use the READ LINE statement from the interactive list processing to process the changes (see Interactive Lists [Page 854]). If you make the contents of variables input-enabled in the output list, the changes the user enters do not affect the variables themselves.

To make output fields input-enabled from within the program, use the FORMAT statement as follows:

Syntax

```
FORMAT INPUT [ON|OFF].
```

To make output fields input-enabled at runtime, use:

Syntax

```
FORMAT INPUT = <i>.
```

Use the ON option (or <i> unequal to zero) to format subsequent output as input-enabled fields. Input-enabled fields have different background and foreground colors than the remainder of the list. For input fields, the options COLOR, INVERSE, and HOTSPOT have no effects. The INTENSIFIED option changes the foreground color of the input fields.

You can make horizontal lines input-enabled by formatting them as input fields. However, blank lines you created using SKIP cannot accept input.

```
REPORT demo_list_format_input.
WRITE   'Please fill in your name before printing:'.
WRITE / '   Enter name here    ' INPUT ON.
ULINE.
WRITE 'You can overwrite the following line:'.
FORMAT INPUT ON INTENSIFIED OFF.
ULINE.
FORMAT INPUT OFF INTENSIFIED ON.
```

In this program, a WRITE statement directly receives the INPUT ON format and a horizontal ULINE line is formatted using the FORMAT statement. The header is defined as a text element. The output appears as follows:

```
Input Fields in Lists                                                          1

Please fill in your name before printing:
   Enter name here

You can overwrite the following line:
--------------------------------------------------------------------------------
```

Due to INTENSIFIED OFF, the foreground color of the second input field differs from that of the first. The user can fill the input fields on the screen, for example:

```
Input Fields in Lists                                                          1

Please fill in your name before printing:
   Zippo Typefast

You can overwrite the following line:
---------------------------------------Demonstration for overwriting lines. -------
```

Outputting Fields as Hotspots

Hotspots are special areas of an output list. If the user clicks once onto a hotspot field, an event is triggered (for example, AT LINE-SELECTION). For fields that are not defined as hotspots, the user must double-click the field or use a function key to trigger an event. For information on events during list processing, see Interactive Lists [Page 854].

To output areas as hotspots, use the following option of the FORMAT statement:

Syntax

```
FORMAT HOTSPOT [ON|OFF].
```

To designate fields as hotspots at runtime, use:

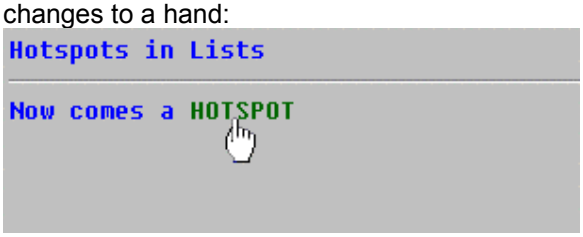
```
FORMAT HOTSPOT = <h>.
```

The ON option (or <h> unequal to zero) formats subsequent output as hotspot. If the user moves the mouse to such a field, the mouse pointer changes to a hand with pointed index finger. As long as this hand is visible, a single click triggers an event. In addition to the changed mouse pointer, you may want to use a different color to mark the hotspot.

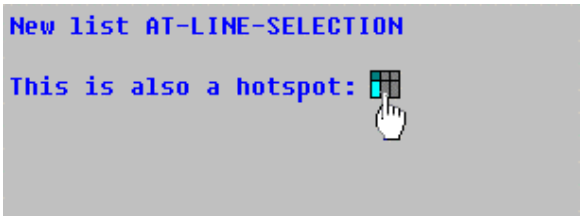
You cannot use the HOTSPOT option if INPUT ON is set, since with HOTSPOT ON the cursor cannot be positioned on an input field. In addition, you cannot format horizontal lines created with ULINE and blank lines created with SKIP as hotspots.

```
REPORT demo_list_format_hotspot.
INCLUDE <list>.
START-OF-SELECTION.
  WRITE 'Now comes a'.
  FORMAT HOTSPOT ON COLOR 5 INVERSE ON.
  WRITE 'HOTSPOT'.
  FORMAT HOTSPOT OFF COLOR OFF.
AT LINE-SELECTION.
  WRITE / 'New list AT-LINE-SELECTION'.
  SKIP.
  WRITE 'This is also a hotspot:'.
  WRITE icon_list AS ICON HOTSPOT.
```

In this program, part of the first line is formatted as hotspot in the START-OF-SELECTION event. The standard page header is defined as a text element. If the user moves the mouse over the word HOTSPOT in the output, the mouse pointer changes to a hand:



A single click triggers the AT-LINE-SELECTION event. At this event, the program creates a secondary list containing another hotspot. The hotspot in the secondary list is an icon:



For information on the AT-LINE-SELECTION event and on secondary lists, see Interactive Lists [Page 854].

Special Output Formats

For a summary of all formatting options of the WRITE statement, see Formatting Options [Page 780]. The current topic describes some special formatting options. These options format output according to certain entries that have to be made in special database tables. Usually, the customer maintains these tables when customizing the application (for information on Customizing, refer to Customizing [Ext.].

Country-specific and User-specific Output Formats

The output formats of number and date fields are defined in the user master record of the individual program user. You can change these settings from within your program, using this statement:

Syntax

```
SET COUNTRY <c>.
```

For <c>, set either a country key defined in table T005X or SPACE.

If <c> is not SPACE, the system turns off the settings from the user master record and searches table T005X for the country key. If the key exists, the system sets SY-SUBRC to 0 and formats the output of all subsequent WRITE statements according to the settings defined in T005X. If the country key you specified does not exist, the system sets SY-SUBRC to 4 and formats for all subsequent WRITE statements the decimal characters as period '.' and date specifications as MM/DD/YY.

If <c> is SPACE, the system does not read table T005X but uses the settings in the user master record. In this case, SY-SUBRC is always zero.

Maintaining table T005X is part of Customizing. However, you can use *System → Services → Table maintenance* to display or change entries.

```
REPORT demo_list_set_country LINE-SIZE 40.

DATA: num TYPE p DECIMALS 3 VALUE '123456.789'.

ULINE.
WRITE: / 'INITIAL:'.
WRITE: / num, sy-datum.
ULINE.

SET COUNTRY 'US'.
WRITE: / 'US,      SY-SUBRC:', sy-subrc.
WRITE: / num, sy-datum.
ULINE.

SET COUNTRY 'GB'.
WRITE: / 'GB,      SY-SUBRC:', sy-subrc.
WRITE: / num, sy-datum.
ULINE.

SET COUNTRY 'DE'.
WRITE: / 'DE,      SY-SUBRC:', sy-subrc.
```

```
WRITE: / num, sy-datum.
ULINE.

SET COUNTRY 'XYZ'.
WRITE: / 'XYZ,     SY-SUBRC:', sy-subrc.
WRITE: / num, sy-datum.
ULINE.

SET COUNTRY space.
WRITE: / 'SPACE,   SY-SUBRC:', sy-subrc.
WRITE: / num, sy-datum.
ULINE.
```

This program outputs the packed number NUM and the system field SY-DATUM using different formatting options.

```
Initial:
     123.456,789  1996/02/07
-------------------------------
US,      SY-SUBRC:     0
     123,456.789  02/07/1996
-------------------------------
GB,      SY-SUBRC:     0
     123.456.789  02-07-1996
-------------------------------
DE,      SY-SUBRC:     0
     123.456,789  07.02.1996
-------------------------------
XYZ,     SY-SUBRC:     4
     123,456.789  02/07/1996
-------------------------------
SPACE,   SY-SUBRC:     0
     123.456,789  1996/02/07
-------------------------------
```

The first and the last output are user-specific. For all other output, the system reads table T005X. It does not find the entry 'XYZ' and sets the output format itself. The entries in T005X are customer-specific.

Currency-specific Output Formats

To format the output of a number field according to a specific currency, use the CURRENCY option of the WRITE statement:

Syntax

```
WRITE <f> CURRENCY <c>.
```

This statement determines the number of decimal places in the output according to the currency <c>. If the contents of <c> exist in table TCURX as currency key CURRKEY, the system sets the number of decimal places according to the entry CURRDEC in TCURX. Otherwise, it uses the default setting of two decimal places. This means that table TCURX must contain only exceptions where the number of decimal places is unequal to 2.

The output format for currencies does not depend on the decimal places of a number that may exist in the program. The system uses only the sequence of digits. This sequence of digits thus

represents an amount specified in the smallest unit of the currency in use, for example Cents for US Dollar (USD) or Francs for Belgian Francs (BEF). For processing currency amounts in ABAP programs, SAP therefore recommends that you use data type P without decimal places.

```
REPORT demo_list_write_currency LINE-SIZE 40.

DATA: num1 TYPE p  DECIMALS 4 VALUE '12.3456',
      num2 TYPE p  DECIMALS 0 VALUE '123456'.

SET COUNTRY 'US'.

WRITE: 'USD', num1 CURRENCY 'USD', num2 CURRENCY 'USD',
     / 'BEF', num1 CURRENCY 'BEF', num2 CURRENCY 'BEF',
     / 'KUD', num1 CURRENCY 'KUD', num2 CURRENCY 'KUD'.
```

This program defines two packed numbers NUM1 and NUM2, containing the same sequence of digits, but different numbers of decimal places. These numbers appear in the output in several currencies:

```
USD          1,234.56          1,234.56
BEF           123,456           123,456
KUD           123.456           123.456
```

For each currency, the output formats of NUM1 and NUM2 are the same, since they refer to the sequence of digits only. The currency US Dollar (USD) appears in the default setting of two decimal places, since the smallest unit is one Cent and a hundredth of a Dollar. For Belgian Francs (BEF), CURRDEC in TCURX is set to 0, since the Belgian Franc has no smaller units. Dinars from Kuwait (KUD) have units of a thousandth and therefore three decimal places (CURRDEC is 3).

Unit-specific Output Formats

You can format fields of type P according to certain units. For example, quantities should not have decimal places, weights should have three decimals, and so on. To do this, you can use the UNIT option in the WRITE statement as follows:

Syntax

```
WRITE <f> UNIT <u>.
```

This statement sets the number of decimal places according to the unit <u>. The contents of <u> must be an entry in database table T006 in column MSEHI. The entry in column DECAN then determines the number of decimal places of the field <f> to be displayed. If the system does not find the entry <u> in table T006, it ignores the option.

The following restrictions apply for this operation:

- <f> must be a packed number (type P).
- If <f> has fewer decimal places than the unit <u>, the system ignores the option.
- If <f> has more decimal places than the unit <u>, the system uses this option only, if the operation does not truncate any digits unequal to 0.

```
REPORT demo_list_write_unit LINE-SIZE 40.

DATA: num1 TYPE p DECIMALS 1 VALUE 1,
      num2 TYPE p DECIMALS 4 VALUE '2.5'.

SET COUNTRY 'US'.

WRITE: 'KG', num1 UNIT 'KG', num2 UNIT 'KG',
     / 'PC', num1 UNIT 'PC', num2 UNIT 'PC'.
```

This program defines two packed numbers, NUM1 with one decimal place and NUM2 with four decimal places. If the unit 'KG' (kilograms) has three decimal places in table T006 and 'PC' (pieces) has zero decimal places in T006, the output appears as follows:

```
KG                1.0              2.500
PC                  1            2.5000
```

The system ignores the option UNIT 'KG' for NUM1, since NUM1 has less than three decimal places. The UNIT 'PC' option shortens the output of NUM1 to zero decimal places. The UNIT 'PC' option shortens the output of NUM1 to zero decimal places. For NUM2, the system ignores the option UNIT 'PC', since otherwise a decimal place unequal to zero would have been truncated.

Lines in Lists

ABAP offers several possibilities to create horizontal and vertical lines. For a list of the corresponding statements, see Lines and Blank Lines [Page 783].

The system automatically joins lines that meet to united lines or frames. Lines meet if in the direction of at least one of the lines no blank characters or blank lines separate the lines. Depending on the type and number of lines meeting at a certain point, you can draw various line sections. To exceptionally prevent the system from joining lines, you can use special lines:

Note in this context that page header and page footer occupy lines of the page. This may lead to undesired joined lines if you forget to create enough blank lines. For example, outputting vertical lines '|' in the first line after the standard page header automatically joins these lines to the underline of the page header.

For a demonstration of automatically joined lines, call the executable program SHOWLINE in any system.

Straight Lines

The example below shows how to create horizontal and vertical straight lines.

```
REPORT demo_list_straight_lines NO STANDARD PAGE HEADING.

SKIP TO LINE 3.
ULINE AT 2(1).
WRITE 4  '-'.
WRITE 6  '--'.
WRITE 9  '---'.
ULINE AT 12(4).

SKIP TO LINE 1.
POSITION 18.
WRITE '|'.

SKIP TO LINE 3.
DO 4 TIMES.
  NEW-LINE.
  POSITION 18.
  WRITE '|'.
ENDDO.
```

The output appears as follows:

The first ULINE statement creates a horizontal line that covers one column. The hyphen in the first WRITE statement appears as a normal output field. The two hyphens of the second WRITE statement create a straight line that covers two columns. The next three hyphens together with the ULINE statement create a straight line that cover seven columns.

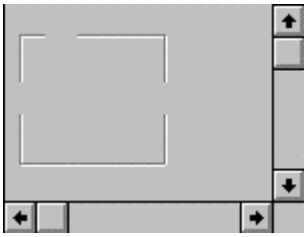
Outputting the first '|' character creates a vertical line in the first line. The other four '|' characters are joined to a vertical straight line that covers four lines, starting at line 3.

Corners

The example below shows how to create different corners.

```
REPORT demo_list_edges NO STANDARD PAGE HEADING.

WRITE   '--'.
WRITE / '|'.

SKIP TO LINE 1.
ULINE AT 5(6).
NEW-LINE.
WRITE 10 '|'.

SKIP TO LINE 4.
WRITE: '|         |',
     / '----------'.
```

This produces the following output:



Wherever the ends of vertical lines meet the ends of horizontal lines, corners appear.

T Sections

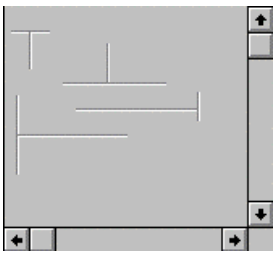
The example below shows how to create different T sections.

```
REPORT demo_list_t_pieces NO STANDARD PAGE HEADING.

WRITE '---'.
WRITE /2 '|      |'.
ULINE AT /5(8).

SKIP TO LINE 4.
DO 3 TIMES.
  WRITE '|'.
```

Lines in Lists

```
  NEW-LINE.
ENDDO.
SKIP TO LINE 5.
WRITE '---------'.
SKIP TO LINE 4.

ULINE AT 6(10).
WRITE 15 '|' .
```

This produces the following output:



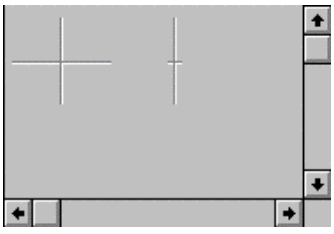
Wherever line ends meet lines at a right angle, T sections appear.

Crosses

The example below shows how to create crosses.

```
REPORT demo_list_crosses NO STANDARD PAGE HEADING.

WRITE   '   |'.
WRITE  /'-------'.
WRITE  /'   |'.

SKIP TO LINE 1.
DO 3 TIMES.
  WRITE 12 sy-vline.
  NEW-LINE.
ENDDO.
SKIP TO LINE 2.
ULINE AT 12(1).
```

The output looks like this:



If two lines intersect, a cross appears.

Using Special Lines

If you use tightly nested frames or tight hierarchy representations, you might want to keep certain line sections apart, even though there is no space left for inserting a blank character or blank line between them.

In this case, you can use special lines defined as system-defined constants in the include program <LINE>:

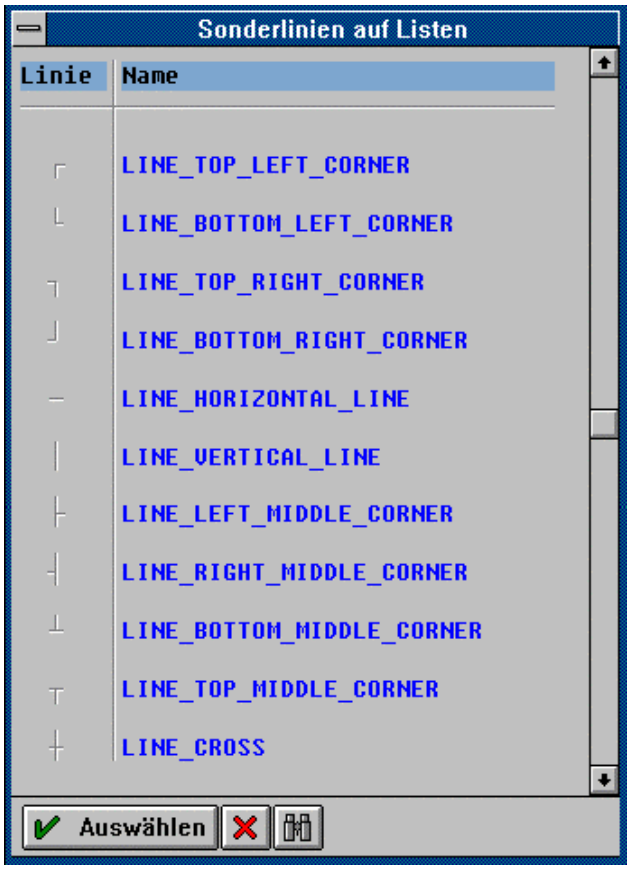
Syntax

```
WRITE <lin> AS LINE.
```

To be able to use special lines in your program, you must include the include program <LINE> or the more comprehensive include program <LIST> into your program. The include program <LINE> contains a short description of the special lines.

The system displays special lines in the output list exactly the way they are defined. Lines are joined only where they really meet. The system does not automatically make special lines longer to make them meet.

The easiest way to output special lines is to use a statement structure (see Using WRITE via a Statement Structure [Page 785]). From the screen *Assemble a WRITE Statement*, select the radio button *Line* and then choose *Display*. The following dialog box appears:



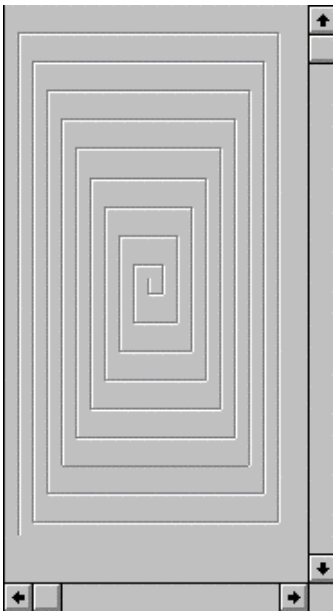
It contains all available special lines which you can easily include into your program code.

Lines in Lists

The following program shows on one hand how to use special lines to create a tight pattern. On the other hand, it also demonstrates how you can program lines in lists dynamically.

```
REPORT demo_list_special_lines NO STANDARD PAGE HEADING LINE-
SIZE 60.

INCLUDE <line>.

DATA: x0 TYPE i VALUE 10,
      y0 TYPE i VALUE 10,
      n TYPE i VALUE 16,
      i TYPE i VALUE 0,
      x TYPE i, y TYPE i.

x = x0. y = y0. PERFORM pos.

WHILE i LE n.
  WRITE line_bottom_left_corner AS LINE.
  x = x + 1. PERFORM pos.
  ULINE AT x(i).
  x = x + i. PERFORM pos.
  WRITE line_bottom_right_corner AS LINE.
  y = y - 1. PERFORM pos.
  DO i TIMES.
    WRITE '|'.
    y = y - 1. PERFORM pos.
  ENDDO.
  WRITE line_top_right_corner AS LINE.
  i = i + 1.
  x = x - i. PERFORM pos.
  ULINE AT x(i).
  x = x - 1. PERFORM pos.
  WRITE line_top_left_corner AS LINE.
  y = y + 1. PERFORM pos.
  DO i TIMES.
    WRITE '|'.
    y = y + 1. PERFORM pos.
  ENDDO.
  i = i + 1.
ENDWHILE.

FORM pos.
  SKIP TO LINE y.
  POSITION x.
ENDFORM.
```

In this program, the position X,Y is set for each output using the subroutine POS. The output appears as follows:



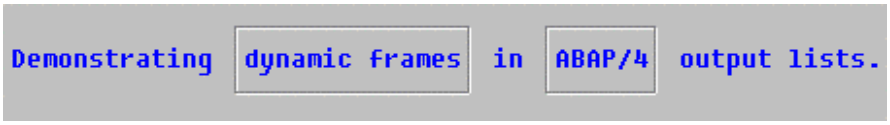
The program creates a tight helix structure which cannot be generated without using special lines. You can set the number of coils using variable N.

Programming Frames

You can use the line types available in ABAP to program frames. The sample program below defines a macro WRITE_FRAME which you can use instead of the WRITE <f> statement. The system draws a frame around a field <f> specified in WRITE_FRAME that dynamically adapts to the length of the field.

```
REPORT demo_list_write_frame NO STANDARD PAGE HEADING LINE-
SIZE 60.

DATA: x TYPE i, y TYPE i, l TYPE i.

DEFINE write_frame.
  x = sy-colno. y = sy-linno.
  write: '|' no-gap, &1 no-gap, '|' no-gap.
  l = sy-colno - x.
  y = y - 1. skip to line y. position x.
  uline at x(l).
  y = y + 2. skip to line y. position x.
  uline at x(l).
  y = y - 1. x = sy-colno. skip to line y. position x.
END-OF-DEFINITION.

SKIP.
WRITE       'Demonstrating'.
write_frame 'dynamic frames'.
WRITE       'in'.
write_frame 'ABAP'.
WRITE       'output lists.'.
```

The function of the macro WRITE_FRAME defined in the above program is demonstrated in the output below. For more information on macros, see Defining and Calling Macros [Page 444].



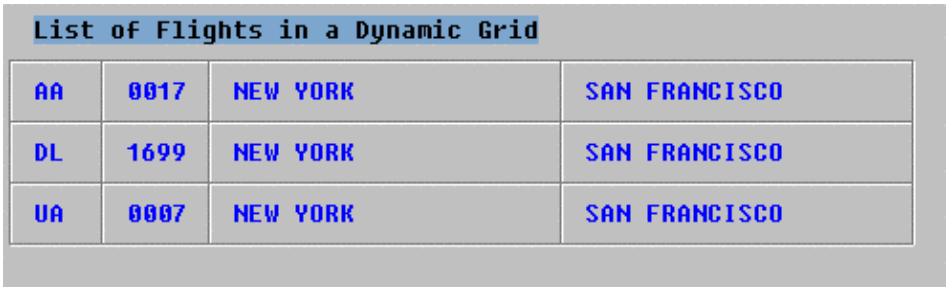
Programming Grids

You can use the line types available in ABAP to program a grid for a table-type list. The sample program below defines two macros NEW_GRID and WRITE_GRID that belong together. NEW_GRID is used to initialize a grid and for line feeds within the grid. You can use WRITE_GRID instead of the WRITE <f> statement. For each field output using WRITE_GRID, the system draws a vertical grid line to the right of the field and a horizontal grid line below it. The horizontal line dynamically adapts to the length of the field. The lines of all output fields together form a grid.

```
REPORT demo_list_grid LINE-SIZE 60 NO STANDARD PAGE HEADING.

TABLES spfli.
DATA:  x TYPE i, y TYPE i, l TYPE i.

TOP-OF-PAGE.
  WRITE 3 'List of Flights in a Dynamic Grid'
          COLOR COL_HEADING.
  ULINE.

START-OF-SELECTION.

  DEFINE new_grid.
    y = sy-linno. y = y + 2. skip to line y.
    x = sy-colno. position x. write '|'.
  END-OF-DEFINITION.

  DEFINE write_grid.
    x = sy-colno. y = sy-linno. position x.
    write:  &1, '|'.
    l = sy-colno - x + 1.
    x = x - 2. y = y + 1. skip to line y. position x.
    uline at x(l).
    y = y - 1. x = sy-colno. skip to line y. position x.
  END-OF-DEFINITION.

GET spfli.

  new_grid.
  write_grid: spfli-carrid,
              spfli-connid,
              spfli-cityfrom,
              spfli-cityto.
```

The functions of the macros NEW_GRID and WRITE_GRID defined in the above program are demonstrated in the output below. For information on macros, see Macros [Page 444]. The executable program is connected to the logical database

F1S. After the user has entered the corresponding values on the selection screen, the output may appear as follows:



Note that the topmost grid line comes from the ULINE statement in the statement block following TOP-OF-PAGE. The system automatically joins the vertical lines of the list body with this line.

Interactive Lists

Lists are displayed in a special container screen. As with all other screens, you can link a dialog status to it using the SET PF-STATUS statement. In the dialog status, you link function codes to function keys, menu entries, icons in the standard toolbar, and pushbuttons in the application toolbar.

On a normal screen, a user action triggers the PAI event. In the screen flow logic, you code a processing block that calls ABAP dialog modules in this event. In list processing, the event is intercepted by the list processor and processed. Instead of calling dialog modules, one of the three following list events may be called, **depending on the function code triggered by the user**.

- AT PF<nn> (obsolete)
- AT LINE-SELECTION
- AT USER-COMMAND

If you have written the corresponding event blocks in your program, they are executed. You can access the function code in the system field SY-UCOMM.

The output from any list statements that you write in these event blocks is written to detail lists. With one ABAP program, you can maintain one basic list and up to 19 detail lists.

Detail Lists [Page 855]

Dialog Status for Lists [Page 860]

Context Menus for Lists [Page 866]

List Events in an ABAP Program [Page 868]

Lists in Dialog Boxes [Page 872]

Passing Data from Lists to Programs [Page 874]

Manipulating Detail Lists [Page 886]

Detail Lists

When the system processes event blocks that are **not** assigned to interactive list events, and when processing dialog modules, the ABAP program writes its list output to the basic list.

The ABAP system field [Page 1444] SY-LSIND contains the index of the list currently being created. While the basic list is being created, SY-LSIND is zero.

By default, the basic list has a standard list status and a standard page header. The TOP-OF-PAGE and END-OF-PAGE events can occur while the basic list is being created. All output in these events is placed in the page header or footer of the basic list. In executable programs, the basic list is automatically sent to the list processor and displayed at the end of the END-OF-SELECTION event. Otherwise, it is displayed after the PAI processing block on the screen from which the LEAVE TO LIST-PROCESSING statement occurred.

Creating Detail Lists

Each time the user executes an action on a list, the runtime environment checks whether there is an event block defined that corresponds to the function code. If there is, SY-LSIND is **automatically increased by one**, and the relevant event block is executed. Any list output arising during this event block places its data into a new list (list level) with the index SY-LSIND. In order to create a new list level, the GUI status of the basic list must allow user actions, and the relevant event blocks must be defined in the program.

All lists created during an interactive list event are detail lists. Each interactive list event creates a new detail list. With one ABAP program, you can maintain one basic list and up to 20 detail lists. If the user creates a list on the next level (that is, SY-LSIND increases), the system stores the previous list and displays the new one. The user can interact with whichever list is currently displayed.

The system displays this list after processing the entire processing block of the event keyword or after leaving the processing block due to EXIT or CHECK. By default, the new list overlays the previous list completely. However, you can display a list in a dialog box [Page 872]. If no other dialog status is set in the event block for the detail list, the system uses the status from the previous list level. However, there is no standard page header for detail lists (see below).

Consequences of Event Control

The fact that you program detail lists in event blocks has important consequences. You cannot nest processing blocks (see Structure of an ABAP Program [Page 44]). Therefore, you cannot process other events within the processing blocks of interactive lists.

Especially, you **cannot**

- use separate processing blocks to process further interactive events. A certain user action always triggers the same processing block in your program. You must use control statements (IF, CASE) within the processing block to make sure that the system processes the desired statements. There is a range of system field that you can use for this.
- use the event TOP-OF-PAGE to influence the list structure of secondary lists. To layout the page headers of the secondary lists, you must use the event TOP-OF-PAGE DURING LINE-SELECTION (see below). However, the system does process the event END-OF-PAGE in secondary lists.
- use events such as GET and GET LATE to retrieve data for secondary lists, but must use SELECT statements. You can use the logical database assigned to the executable program

(report) only for the basic list. If you want to use a logical database during interactive events, you must either call [Page 1018] another executable program using SUMIT, or (better) call the logical database using a function module [Page 1185].

Navigating in Lists

To return from a high list level to the next-lower level (SY-LSIND), the user can choose *Back* from a detail list. The system releases the last list to have been displayed, and returns to the previous list level. The system deletes the contents of the released list.

To determine the list level in which the output from an event block will be displayed, you can change the value of the system fields SY-LSIND. This is one of the few exceptions to the rule that states that you must never overwrite ABAP system fields [Page 1444]. The system accepts only index values which correspond to existing list levels. It then deletes all existing list levels whose index is greater or equal to the index you specify. For example, if you set SY-LSIND to 0, the system deletes **all** secondary lists and overwrites the basic list with the current secondary list.

When you change SY-LSIND, the change only takes effect at the end of the corresponding event. If you work with statements that access the list with index SY-LSIND (using the INDEX addition - for example, SCROLL), you should set the new value of SY-LISND after these statements. The best place to set it is in the last statement of the event block.

System Fields for Details Lists

After each user action on a list, the following ABAP system fields [Page 1444] will be set in the corresponding event block:

System field	Information
SY-LSIND	Index of the list created during the current event (basic list = 0)
SY-LISTI	Index of the list level from which the event was triggered
SY-LILLI	Absolute number of the line from which the event was triggered
SY-LISEL	Contents of the line from which the event was triggered
SY-CUROW	Position of the line in the window from which the event was triggered (counting starts with 1)
SY-CUCOL	Position of the column in the window from which the event was triggered (counting starts with 2)
SY-CPAGE	Page number of the first displayed page of the list from which the event was triggered
SY-STARO	Number of the first line of the first page displayed of the list from which the event was triggered (counting starts with 1). This line may contain the page header.
SY-STACO	Number of the first column displayed in the list from which the event was triggered (counting starts with 1)
SY-UCOMM	Function code that triggered the event
SY-PFKEY	Status of the list currently being displayed.

Page Headers for Detail Lists

On detail lists, the system does not display a standard page header and it does not trigger the event TOP-OF-PAGE. To create page headers for detail list, you must use a different TOP-OF-PAGE event:

```
TOP-OF-PAGE DURING LINE-SELECTION.
```

The system triggers this event for **each** detail list. If you want to create different page headers for different list levels, you must program the processing block of this event accordingly, for example by using system fields such as SY-LSIND or SY-PFKEY in control statements (IF, CASE).

As on the basic list, the page header of a detail list remains displayed even when you scroll vertically.

Messages on Detail Lists

ABAP allows you to react to incorrect or possibly-incorrect user input by displaying messages [Page 927]. The seriousness of the error determines how program processing continues.

In list processing, the message processing [Page 931] for the individual message types is as follows:

- A (=Abend): Termination

  The system displays the message in a dialog box. After the user confirms the message using `ENTER`, the system terminates the entire transaction (for example SA38).

- E (=Error) or W (=Warning):

  The system displays the message in the status bar. Once the user has confirmed the error by pressing ENTER, the current event block is terminated and the previous list level remains displayed. If you use an error or warning message while creating the basic list, the entire program is terminated.

- I (=Information):

  The system displays the message in a dialog box. Once the user has confirmed the message (ENTER), the program continues processing after the MESSAGE statement.

- S (= status)

  The system displays the message in the status bar of the current list.

- X (= Exit) Runtime error:

  This message type triggers a runtime error and generates a short dump.

Using Detail Lists

A classic report is a program that generates a single list, which must contain all of the required detail information. This procedure may result in extensive lists from which the user has to pick the relevant data. For background processing, this is the only possible method. After starting a background job, there is no way of influencing the program. The desired selections must be made beforehand and the list must provide detailed information.

For dialog sessions, there are no such restrictions. The user is present during the execution of the program and can control and manipulate the program flow directly. To be able to use all advantages of the online environment, classical reporting was developed into interactive reporting.

Detail Lists

Interactive reporting allows the user to participate actively in retrieving and presenting data during the session. Instead of one extensive and detailed list, with interactive reporting you create a condensed basic list from which the user can call detailed information by positioning the cursor and entering commands. Interactive reporting thus reduces information retrieval to the data actually required. Detailed information is presented in detail lists.

Apart from creating detail lists, interactive reporting also allows you to call transactions or other executable programs (reports) from lists. These programs then use values displayed in the list as input values. The user can, for example, call a transaction from within a list to change the database table whose data is displayed in the list.

Examples

Creating Detail Lists

```
REPORT demo_list_interactive_1.
START-OF-SELECTION.
  WRITE: 'Basic List, SY-LSIND =', sy-lsind.
AT LINE-SELECTION.
  WRITE: 'Secondary List, SY-LSIND =', sy-lsind.
```

When you run the program, the basic list appears. The GUI status [Page 860] automatically permits the function *Choose* (F2). When you choose a list line, the system triggers the AT LINE-SELECTION event, and the first detail list overlays the basic list. This list has no standard page header. It also inherits the GUI status of the basic list. By choosing *Choose*, the user can now create up to 19 of these lists. Trying to produce more than 19 lists results in a runtime error. Using *Back* , the user can return to previous lists.

Navigation in detail lists.

```
REPORT demo_list_interactive_2.
START-OF-SELECTION.
  WRITE: 'Basic List, SY-LSIND =', sy-lsind.
AT LINE-SELECTION.
  IF sy-lsind = 3.
    sy-lsind = 0.
  ENDIF.
  WRITE: 'Secondary List, SY-LSIND =', sy-lsind.
```

When you run the program, the basic list appears:

```
Basic List, SY-LSIND =    0
```

The GUI status [Page 860] automatically permits the function *Choose* (F2). If the user positions the cursor on the list line and chooses *Choose* to trigger the AT LINE-SELECTION event, the system displays a detail list that contains the following line:

```
Secondary List, SY-LSIND =  1
```

Choosing *Choose* again produces:

```
Secondary List, SY-LSIND =  2
```

Back leads to the previous list level. Choosing *Choose* for the third time produces a detail list that contains the following line (because of the IF condition):

```
Secondary List, SY-LSIND =  0
```

The system deletes list levels 1 and 2. Choosing *Back* returns to the point at which the list processing started. If you choose *Choose*, the system creates a detail list with index 1. However, the list on level 0 is no longer a basic list (no page header), but is itself a detail list.

Page Headers for Detail Lists

```
REPORT demo_list_interactive_3.
START-OF-SELECTION.
  WRITE  'Basic List'.
AT LINE-SELECTION.
  WRITE  'Secondary List'.
TOP-OF-PAGE DURING LINE-SELECTION.
  CASE sy-lsind.
    WHEN 1.
      WRITE 'First Secondary List'.
    WHEN 2.
      WRITE 'Second Secondary List'.
    WHEN OTHERS.
      WRITE: 'Secondary List, Level:', sy-lsind.
  ENDCASE.
  ULINE.
```

When you run the program, the basic list appears. The user can choose *Choose* to create detail lists. The detail lists have page headers that are set according to the value of SY-LSIND.

Messages on Detail Lists

```
REPORT demo_list_interactive_4 NO STANDARD PAGE HEADING.
AT LINE-SELECTION.
  WRITE 'Basic List'.
  MESSAGE s888(sabapdocu) WITH text-001.
AT LINE-SELECTION.
  IF sy-lsind = 1.
    MESSAGE i888(sabapdocu) WITH text-002.
  ENDIF.
  IF sy-lsind = 2.
    MESSAGE e888(sabapdocu) WITH text-003 sy-lsind text-004.
  ENDIF.
  WRITE: 'Secondary List, SY-LSIND:', sy-lsind.
```

When the program runs, the system displays the basic list and the success message 100 in the status line. A single click triggers the AT-LINE-SELECTION event. When the system creates the first detail list, it displays a dialog box with the information message 100. You cannot create the second detail list, because the message 200 has message type E:

Dialog Status for Lists

To allow the user to communicate with the system when a list is displayed, the lists must be able to direct user actions to the ABAP program. As described in User Actions on Screens [Page 535], function codes are used to do this. Function codes are maintained in the GUI status of the list screen. You define a GUI status using the Menu Painter tool in the ABAP Workbench. The system assigns function codes to list-specific user actions.

The most important of these functions is for selecting list lines by double-clicking. As described in Using a GUI Status, the double-click function is **always** linked to the F2 key. If a function code is assigned to the F2 key in the GUI status, it will be triggered when you double-click.

The Standard List Status

As with normal screens, you can define your own GUI status for lists and attach it to a list level using the SET PF-STATUS statement. If you do not set a particular GUI status, the system sets a default list status for the list screen in an executable program. In other programs, for example, when you call a list from screen processing [Page 896], you must set this status explicitly using the statement

```
SET PF-STATUS space.
```

This default interface always contains at least the functions described in the Standard List [Page 789] section.

Unlike normal dialog statuses, the default list status is **affected by the ABAP program**.

If you define event blocks in your program using the event keywords AT LINE-SELECTION or AT PF<nn>, the system **automatically** assigns extra functions to other function keys that provide additional functions.

- AT PF<nn> (obsolete)
  All function keys of the keyboard `F<nn>` that are not used for predefined system functions, are set to the function codes PF<nn>, where <nn> is a number between 01 and 24. During list processing, the function codes PF<nn> are linked to the events AT PF<nn>. **Choosing PF<nn> always triggers AT PF<nn>.** You should no longer use event blocks for PF<nn>.
- AT LINE-SELECTION
  For this event, the F2 key (double-click) is assigned the function code PICK and function code *Choose*. The function also always appears in the application toolbar. During list processing, the PICK function code is assigned to the event AT LINE-SELECTION. **PICK always triggers AT LINE-SELECTION**.

All other function codes are either intercepted by the runtime environment or trigger the event AT USER-COMMAND. Function codes that trigger AT USER-COMMAND must be defined in your own GUI status. The easiest way to do this is to use the standard list status and add extra functions of your own to it.

Dialog Status for Lists

You can create a dialog status for a list using the Menu Painter [Ext.] in the ABAP Workbench [Ext.]. As soon as you create the status, you should choose *Extras* → *Adjust template* and select *List status* for your template. This status already contains standard function codes for list processing. They are distributed as follows on the menus, standard toolbar, and application toolbar:

Code	Menu	Standard toolbar	Function key	Description
%PC	*List*			Save list to file
%SL	*List*			Save list in SAPoffice
%ST	*List*			Save list in report tree
PRI	*List*		CTRL-P	Print displayed list
%EX	*List*		Shift-F3	Exit processing
PICK	*Edit*		F2	Event AT LINE-SELECTION
RW	*Edit*		F12, ESC	Cancel processing
%SC	*Edit*		CTRL-F	Find
%SC+	*Edit*		CTRL-G	Find next
BACK	*Goto*		F3	Back one level
P--			CTRL-PgUp	Scroll to first window page
P-			PgUp	Scroll to previous window page
P+			PgDn	Scroll to next window page
P++			Ctrl-PgDn	Scroll to last window page
%CTX			Shift-F10	Context menu on list

In addition, the following function codes are predefined, but not set as status functions. You can assign them freely to any empty status element.

Code	Description
PF<nn>	Event AT PF<nn>
PP<n>	Scroll to top of list page <n>
PP-[<n>]	Scroll back one list page or <n> pages
PP+[<n>]	Scroll forward one list page or <n> pages

Dialog Status for Lists

PS<n>	Scroll to column <n>
PS--	Scroll to first column of the list
PS-[<n>]	Scroll left by one or <n> columns
PS+[<n>]	Scroll right by one or <n> columns
PS++	Scroll to last column of the list
PZ<n>	Scroll to line <n>
PL-[<n>]	Scroll back to first line of the page or by <n> lines
PL+[<n>]	Scroll to last line of the page or by <n> lines
/....	For other system commands

The runtime environment directly queries and processes all function codes of the above tables, except PICK and PF<nn>. These function codes do not trigger events, and you cannot use them for the AT USER-COMMAND event.

The function code PICK triggers the AT LINE-SELECTION event, whenever the cursor is positioned on a list line. The function codes PF<nn> always trigger the AT PF<nn> event. Therefore, you cannot use them for the AT USER-COMMAND event either.

To trigger AT USER-COMMAND, you can define any number of function codes yourself. If they do not have a function code listed in the above table, they will trigger AT USER-COMMAND. Note the following special function key assignments:

- Function key `F2`:

  A double-click is always equivalent to pressing function key `F2`. Each function code you assign to `F2` is therefore activated by double-clicking. Double-clicking triggers the AT LINE-SELECTION event only if the function code PICK is assigned to function key `F2`. If you assign your own function code to `F2`, double-clicking triggers the AT USER-COMMAND event. If a predefined function code is assigned to `F2`, double-clicking triggers the corresponding action from the runtime environment.

- Function key `SHIFT-F10`:

  The `SHIFT-F10` key is always the equivalent of right-clicking. When you use context menus in lists [Page 866], it is assigned the function code %CTX.

You can change the list-specific template if you have other requirements. You can

- replace function code PICK with your own function code to prevent the AT LINE-SELECTION event from being triggered in the report. You can then program all reactions to user actions in a single processing block (AT USER-COMMAND).
- delete predefined function codes whose functionality you do not want to support. For example, this allows you to prevent the user from printing the list or saving it in a file on the presentation server.
- modify the standard key settings. For example, you can assign your own function code to `F3` to navigate within the lists according to your requirements, instead of returning one list level (*Back*). This may be important if you keep several lists on the same logical level and therefore do not want to delete the displayed list as would the standard `F3` setting. Or you may want to display a warning before leaving a list level.

Setting a Dialog Status

You set the dialog status for lists in the same way as for normal screens, that is, using the statement

```
SET PF-STATUS <stat> [EXCLUDING <f>|<itab>]
                     [OF PROGRAM <prog>]
                     [IMMEDIATELY].
```

This statement sets the status <stat> for the current output list. The dialog status <stat> must be defined for the current program, unless you have used the OF PROGRAM addition to set a status from another program <prog>. The status is active for all subsequent list levels until you set another status. The SY-PFKEY system field always contains the status of the current list.

Using SET PF-STATUS, you can display different user interfaces for different list levels to provide the user with different functions according to the individual requirements. Use SET PF-STATUS SPACE to set the standard list status. This depends on the event blocks in the program, as described above.

The EXCLUDING option allows you to change the appearance and available functions of a status from within the program. This is useful if the individual user interfaces for a range of list levels are very similar. You can define a single global status, and then just deactivate the functions you do not need using EXCLUDING. Specify <f> to deactivate the function code stored in field <f>. Specify <itab> to deactivate all function codes stored in the internal table <itab>. Field <f> and the lines of table <itab> should be defined with reference to the system field SY-UCOMM.

The IMMEDIATELY addition is intended specially for list processing. You use it while creating a detail list within an event block to change the status of the list currently displayed (index SY-LISTI). Without this option, the system changes the status of the current secondary list (SY-LSIND) that is displayed only at the end of the processing block.

Setting a Title for a List

As with normal screens, you can set a title for a list as follows:

```
SET TITLEBAR <ttl> [WITH <g1> ... <g9>]
                   [OF PROGRAM <prog>].
```

During list processing, this statement sets the title of the user interface for the output list. It remains active for all screens until you specify another using SET TITLEBAR. The GUI title <title> must be a component of the current ABAP program, unless you use the OF PROGRAM addition in the SET TITLEBAR statement to set a GUI status of another program <prog>.

You can use the WITH option of the SET TITLEBAR statement to replace these placeholders in the title at runtime with the contents of the corresponding fields <g_1> .... <g9>. The system also replaces '&' placeholders in succession by the contents of the corresponding <g_i> parameters. To display an ampersand character ‘&’, repeat it in the title ‘&&’.

Examples

Example for dialog status in a list.

```
REPORT demo_list_menu_painter.
START-OF-SELECTION.
  SET PF-STATUS 'TEST'.
  WRITE:  'Basic list, SY-LSIND =', sy-lsind.
AT LINE-SELECTION.
```

```
  WRITE:  'LINE-SELECTION, SY-LSIND =', sy-lsind.
AT USER-COMMAND.
  CASE sy-ucomm.
    WHEN 'TEST'.
      WRITE:  'TEST, SY-LSIND =', sy-lsind.
  ENDCASE.
```

This program uses a status TEST, defined in the Menu Painter.

1. Function key F5 has the function code *TEST* and the text *Test for demo*.
2. Function code TEST is entered in the *List* menu.
3. The function codes PICK and TEST are assigned to pushbuttons.

The user can trigger the AT USER-COMMAND event either by pressing F5, or by choosing *List* → *Test for demo*, or by choosing the pushbutton *Test for demo*.The user can trigger the AT LINE-SELECTION event by selecting a line.

Example of setting a dialog status for the current list

```
REPORT demo_list_set_pf_status_1.
DATA: fcode TYPE TABLE OF sy-ucomm,
      wa_fcode TYPE sy-ucomm.
START-OF-SELECTION.
  wa_fcode = 'FC1 '. APPEND wa_fcode TO fcode.
  wa_fcode = 'FC2 '. APPEND wa_fcode TO fcode.
  wa_fcode = 'FC3 '. APPEND wa_fcode TO fcode.
  wa_fcode = 'FC4 '. APPEND wa_fcode TO fcode.
  wa_fcode = 'FC5 '. APPEND wa_fcode TO fcode.
  wa_fcode = 'PICK'. APPEND wa_fcode TO fcode.
  SET PF-STATUS 'TEST'.
  WRITE: 'PF-Status:', sy-pfkey.
AT LINE-SELECTION.
  IF sy-lsind = 20.
    SET PF-STATUS 'TEST' EXCLUDING fcode.
  ENDIF.
  WRITE: 'Line-Selection, SY-LSIND:', sy-lsind,
       / '                SY-PFKEY:', sy-pfkey.
AT USER-COMMAND.
  IF sy-lsind = 20.
      SET PF-STATUS 'TEST' EXCLUDING fcode.
    ENDIF.
    WRITE: 'User-Command, SY-LSIND:', sy-lsind,
         / '              SY-UCOMM:', sy-ucomm,
         / '              SY-PFKEY:', sy-pfkey.
```

Suppose that the function codes FC1 to FC5 are defined in the status TEST and assigned to pushbuttons: The function code PICK is assigned to function key F2. When the program starts, the user can create detail lists by selecting a line or choosing one of the function codes FC1 to FC5. For all secondary lists up to level 20, the user interface TEST is the same as for the basic list:
On list level 20, EXCLUDING ITAB deactivates all function codes that create detail lists. This prevents the user from causing a program termination by trying to create detail list number 21.

Example of setting a dialog status for the current list

```
REPORT demo_list_set_pf_status_2.
START-OF-SELECTION.
  WRITE: 'SY-LSIND:', sy-lsind.
AT LINE-SELECTION.
  SET PF-STATUS 'TEST' IMMEDIATELY.
```

After executing the program, the output screen shows the basic list and the user interface predefined for line selection (with function code PICK).
When you choose *Choose*, the user interface changes. However, since the AT LINE-SELECTION processing block does not contain an output statement, the system does not create a detail list: The status TEST is defined as in the previous example.

Example: Titles of detail lists.

```
REPORT demo_list_title .
START-OF-SELECTION.
  WRITE 'Click me!' HOTSPOT COLOR 5 INVERSE ON.
AT LINE-SELECTION.
  SET TITLEBAR 'TIT' WITH sy-lsind.
  WRITE 'Click again!' HOTSPOT COLOR 5 INVERSE ON.
```

In this program, a new title is set for each detail list. The title is defined as follows: "Title for Detail List &1".

Context Menus for Lists

As with normal screens, the system creates a standard context menu when you use a dialog status in a list [Page 860]. You can call this standard context menu using the right-hand mouse button (`Shift + F10`). It displays all of the functions assigned to function keys.

You can define context menus [Page 639] for list lines in the same way as for screen elements. To do this, you must assign a special function code to the function key `Shift+F10` in the dialog status of the list. To define context menus for a list, you must first have defined a dialog status for the list [Page 860] and set it using SET PF-STATUS.

In this dialog status, which you will normally create using the *List status* template, the *List with context menu* option must be selected in the function key setting attributes. To do this, place the cursor on a function key setting in the Menu Painter and choose *Attributes* or *Goto* → *Attributes* → *F key setting*. The function code %CTX is assigned to function key `Shift+F10`. Since the introduction of context menus on lists, it is no longer possible to assign `Shift+F10` freely to any function in the Menu Painter. In any existing dialog status where a function code was assigned to `Shift+F10`, it has been reassigned to `Shift+Ctrl+0`. You must activate the function code %CTX manually before it has any effect in the dialog status.

As on screens, context menus on lists are generated dynamically in ABAP programs as objects of the class CL_CTMENU. For context menus on lists, you must program a callback routine in the ABAP program:

```
FORM on_ctmenu_request USING <l_menu> TYPE REF TO cl_ctmenu.
  ...
ENDFORM.
```

In this subroutine, you can define a context menu using the object reference <l_menu> as described in the Context Menus [Page 639] section. To define a specific context menu, for example, you could get the cursor position on the list using GET CURSOR. If required, you may need to find out the current list level from the corresponding system fields (for example, SY-LISTI).

When you right-click a list line (or choose `Shift+F10`), the callback routine is executed, and the context menu defined in it is displayed. If the user chooses a menu item, the system carries on processing according to the function code assigned to it. The function is either executed by the runtime environment, or the corresponding event is triggered (in which case, the function code is placed in the system field SY-UCOMM).

If you right-click outside a list line, the system displays the standard context menu.

```
REPORT demo_list_context_menu .

DATA: wa_spfli   TYPE spfli,
      wa_sflight TYPE sflight.

START-OF-SELECTION.
  SET PF-STATUS 'BASIC'.
  SELECT * FROM spfli INTO wa_spfli.
    WRITE: / wa_spfli-carrid,
             wa_spfli-connid,
             wa_spfli-cityfrom,
             wa_spfli-cityto.
```

```
    HIDE: wa_spfli-carrid, wa_spfli-connid.
  ENDSELECT.
  CLEAR wa_spfli.

AT USER-COMMAND.
  CASE sy-ucomm.
    WHEN 'DETAIL'.
      CHECK NOT wa_spfli IS INITIAL.
      WRITE sy-lisel COLOR COL_HEADING.
      SELECT * FROM sflight INTO wa_sflight
               WHERE carrid = wa_spfli-carrid
                 AND connid = wa_spfli-connid.
        WRITE / wa_sflight-fldate.
      ENDSELECT.
  ENDCASE.

FORM on_ctmenu_request USING l_menu TYPE REF TO cl_ctmenu.
  DATA lin TYPE i.
  IF sy-listi = 0.
    GET CURSOR LINE lin.
    IF lin > 2.
      CALL METHOD l_menu->add_function
                  EXPORTING fcode = 'DETAIL'
                            text  = text-001.
    ENDIF.
    CALL METHOD l_menu->add_function
                EXPORTING fcode = 'BACK'
                          text  = text-002.
  ENDIF.
ENDFORM.
```

In the dialog status BASIC for the basic list, %CTX is assigned to `Shift+F10`. In the callback routine, a context menu is defined. The definition depends on the cursor position and the list currently displayed.

If the user right-clicks the two-line default page header on the basic list, the system displays a single-line context menu. The *Back* function is executed by the runtime environment. If you right-click a list line, a two-line context menu is displayed. The *Detail* function triggers the event AT USER-COMMAND.

List Events in an ABAP Program

After a user interaction on a list, the ABAP runtime environment checks whether it should process the function code itself (for example, %EX or %PC), or whether it should trigger the relevant event. Unlike normal screens, in which the PAI event is triggered, the runtime environment triggers three special events when list events occur. You can use the corresponding event blocks in your ABAP program to react to the user action. From within the program, you can use the SY-UCOMM system field to access the function code. There is no OK_CODE field that is filled.

Event Blocks for Function Codes PF<nn> (Obsolete)

When the user chooses a function code PF<nn> (<nn> can be between 01 and 24), the system always triggers the AT PF<nn> event. In the standard list status, the function keys `F<nn>` that are not reserved for predefined system functions all have the function code PF<nn> as long as a corresponding event block is defined in the program.

```
AT PF<nn>.
<statements>.
```

These event blocks are executed when the user chooses the corresponding function key. The position of the cursor in the list is irrelevant.

If you use these event blocks at all, it should **only be for temporary test versions**. In production programs, you should only use AT USER-COMMAND with a dialog status [Page 860] of your own to assign function codes to function keys. When you use your own interfaces, the system displays a function text explaining what the function does. This does not happen when you use AT PF<nn> event blocks.

Event Block for Function Code PICK

When the user triggers the function code PICK, AT LINE-SELECTION is always triggered if the cursor is positioned on a list line. The function code PICK is, by default, always linked with function key `F2` and hence with the mouse double-click. Consequently, if you have a simple program that does not react to any further user actions, you only need to write this event block.

```
AT LINE-SELECTION.
  <statements>.
```

As described in the section Dialog Status for Lists [Page 860], the function code PICK is always added to the standard list status when you have an AT LINE-SELECTION event in your program.

If you assign PICK to other function keys or menu entries, AT LINE-SELECTION is also triggered when the user chooses then. You should avoid this for the sake of the semantics.

Conversely, if you have a more extensive program that does not react to line selection, you should not use the function code PICK. Instead you should assign a different function code to `F2`, to ensure that as many events as possible trigger the AT USER-COMMAND event.

Event Block for User-Defined Function Codes

If the user chooses a function code during list processing that is neither processed by the system, or PICK or PF<nn>, the system triggers the event AT USER-COMMAND. For this event, you must define your own GUI status for a list [Page 860]. To react to your own function codes in a program, you must define the following event block:

```
AT USER-COMMAND.
<statements>.
```

In this event block, you can use an IF or CASE structure to tell the function codes apart. They are available in the system field SY-UCOMM. There are further system fields that are filled in list events, such as SY-LSIND and SY-PFKEY, that allow you to make further case distinctions.

Triggering a List Event from the Program

You can trigger a list event from the program as follows:

```
SET USER-COMMAND <fc>.
```

This statement takes effect after the current list is completed. Before the list is displayed, the event assigned to function code <fc> is triggered, regardless of the dialog status you are using.

The effect is the same as when the user chooses the function. In other words, predefined list function codes are trapped and processed by the runtime environment, the function codes PICK and PF<nn> trigger the AT LINE-SELECTION and AT PF<nn> events, and user-defined function codes trigger the AT USER-COMMAND event block.

Function code PICK triggers an event only if the cursor is located on a list line.

Using this statement in conjunction with the function codes reserved for system functions, you can call the system functions from the program. For example, you can use SET USER-COMMAND '%SC' to call the *Find* dialog box directly, or to position the list correctly before it is displayed.

If you use several SET USER-COMMAND statements while creating a list, the system executes only the last one.

Examples

Example for AT LINE-SELECTION.

```
REPORT demo_list_at_line_selection.
START-OF-SELECTION.
  WRITE  'Basic List'.
AT LINE-SELECTION.
  WRITE: 'Secondary List by Line-Selection',
       / 'SY-UCOMM =', sy-ucomm.
```

When you run the program, the basic list appears with the standard list status. The detail list shows that SY-UCOMM has the value PICK.

Example for AT PF<nn>.

```
REPORT demo_list_at_pf.
START-OF-SELECTION.
  WRITE 'Basic List, Press PF5, PF6, PF7, or PF8'.
AT pf5.
  PERFORM out.
AT pf6.
  PERFORM out.
AT pf7.
  PERFORM out.
```

```
AT pf8.
  PERFORM out.
FORM out.
  WRITE: 'Secondary List by PF-Key Selection',
       / 'SY-LSIND =', sy-lsind,
       / 'SY-UCOMM =', sy-ucomm.
ENDFORM.
```

After executing the program, the system displays the basic list. The user can press the function keys F5, F6, F7, and F8 to create secondary lists. If, for example, the 14th key the user presses is F6, the output on the displayed secondary list looks as follows:

```
Secondary List by PF-Key Selection
SY-LSIND = 14
SY-UCOMM = PF06
```

Example for AT USER-COMMAND.

```
REPORT demo_list_at_user_command NO STANDARD PAGE HEADING.
START-OF-SELECTION.
  WRITE: 'Basic List',
       / 'SY-LSIND:', sy-lsind.
TOP-OF-PAGE.
  WRITE 'Top-of-Page'.
  ULINE.
TOP-OF-PAGE DURING LINE-SELECTION.
  CASE sy-pfkey.
    WHEN 'TEST'.
      WRITE 'Self-defined GUI for Function Codes'.
      ULINE.
  ENDCASE.
AT LINE-SELECTION.
  SET PF-STATUS 'TEST' EXCLUDING 'PICK'.
  PERFORM out.
  sy-lsind = sy-lsind - 1.
AT USER-COMMAND.
  CASE sy-ucomm.
    WHEN 'FC1'.
      PERFORM out.
      WRITE / 'Button FUN 1 was pressed'.
    WHEN 'FC2'.
      PERFORM out.
      WRITE / 'Button FUN 2 was pressed'.
    WHEN 'FC3'.
      PERFORM out.
      WRITE / 'Button FUN 3 was pressed'.
    WHEN 'FC4'.
      PERFORM out.
      WRITE / 'Button FUN 4 was pressed'.
    WHEN 'FC5'.
      PERFORM out.
      WRITE / 'Button FUN 5 was pressed'.
  ENDCASE.
  sy-lsind = sy-lsind - 1.
```

```
FORM out.
  WRITE: 'Secondary List',
       / 'SY-LSIND:', sy-lsind,
       / 'SY-PFKEY:', sy-pfkey.
ENDFORM.
```

When you run the program, the system displays the following basic list with a the page header defined in the program:
You can trigger the AT LINE-SELECTION event by double-clicking a line. The system sets the status TEST and deactivates the function code PICK. The status TEST contains function codes FC1 to FC5. These are assigned to pushbuttons in the application toolbar. The page header of the detail list depends on the status.
Here, double-clicking a line no longer triggers an event. However, there is now an application toolbar containing five user-defined pushbuttons. You can use these to trigger the AT USER-COMMAND event. The CASE statement contains a different reaction for each pushbutton.
For each interactive event, the system decreases the SY-LSIND system field by one, thus canceling out the automatic increase. All detail lists now have the same level as the basic list and thus overwrite it. While the detail list is being created, SY-LSIND still has the value 1.

Example for SET USER-COMMAND.

```
REPORT demo_list_set_user_command NO STANDARD PAGE HEADING.
START-OF-SELECTION.
  SET USER-COMMAND 'MYCO'.
  WRITE 'Basic List'.
AT USER-COMMAND.
  CASE sy-ucomm.
    WHEN 'MYCO'.
      WRITE 'Secondary List from USER-COMMAND,'.
      WRITE: 'SY-LSIND', sy-lsind.
      SET USER-COMMAND 'PF05'.
  ENDCASE.
AT pf05.
  WRITE 'Secondary List from PF05,'.
  WRITE: 'SY-LSIND', sy-lsind.
  SET CURSOR LINE 1.
  SET USER-COMMAND 'PICK'.
AT LINE-SELECTION.
  WRITE 'Secondary List from LINE-SELECTION,'.
  WRITE: 'SY-LSIND', sy-lsind.
  SET USER-COMMAND '%SC'.
```

This program creates one basic list and three detail lists. When the program starts, the third detail list is displayed immediately, along with a dialog box for searching in the list. The dialog box is displayed by setting the predefined function code %SC. To view the other lists, the user chooses *Back*.
Note that in the event AT PF05, the SET CURSOR statement is used to position the cursor on a list line in order to support the function code PICK.

Lists in Dialog Boxes

You can display a list in a dialog box instead of on the full screen using the WINDOW statement:

```
WINDOW STARTING AT <left> <upper> [ENDING AT <right> <lower>].
```

When you use this statement, the system displays the current list (index SY-LSIND) as a dialog box. Use <left> and <upper> to specify the column and line of the top left-hand corner of the dialog box in relation to the screen containing the basic list. If <upper> is 0, the list appears on a full screen. The coordinates of the lower right-hand corner depend on the space required by the secondary list. You can specify the ENDING option, using <right> and <lower> to set the column and line of the lower right-hand corner. The window will not exceed these values. By default, the system uses the values of the lower right corner of the window on which the event occurred.

If the width of the dialog box is smaller than that of the preceding list, the system creates a horizontal scrollbar on the dialog box. To prevent that, you must adapt the width of the detail list to the width of the dialog box by using the following statement:

```
NEW-PAGE LINE-SIZE <width>.
```

The WINDOW statement takes effect only within the processing block of an interactive event, that is, only for detail lists. The list functions for lists in dialog box are the same as for fullscreen lists.

The GUI status type is different for dialog boxes. Dialog boxes have **no menu bar and no standard toolbar**. The application toolbar appears at the bottom of the dialog box. This is a feature common to all dialog boxes in the R/3 System.

If you do not set your own status but have defined an AT LINE-SELECTION or AT PF<nn> processing block in your program, the system uses a standard list status for a dialog box containing the corresponding function codes.

To define a status for a dialog box, choose *Dialog box* as the status type in the Menu Painter. The system does not provide a menu bar or a standard toolbar. In the application toolbar, the function codes PRI, %SC, %SC+, and RW are preset to allow the user to print the list, search for patterns, and leave the window.

```
REPORT demo_list_window NO STANDARD PAGE HEADING.
START-OF_SELECTION.
  SET PF-STATUS 'BASIC'.
  WRITE 'Select line for a demonstration of windows'.
AT USER-COMMAND.
  CASE sy-ucomm.
    WHEN 'SELE'.
      IF sy-lsind = 1.
        SET PF-STATUS 'DIALOG'.
        SET TITLEBAR 'WI1'.
        WINDOW STARTING AT 5 3 ENDING AT 40 10.
        WRITE 'Select line for a second window'.
      ELSEIF sy-lsind = 2.
        SET PF-STATUS 'DIALOG' EXCLUDING 'SELE'.
        SET TITLEBAR 'WI2'.
        WINDOW STARTING AT 45 10 ENDING AT 60 12.
        WRITE 'Last window'.
```

```
      ENDIF.
  ENDCASE.
```

This program sets status BASIC for the basic list. In status BASIC, the function code PICK proposed for function key `F2` is replaced by the user-defined function code SELE (text *SELECT*) which is also assigned to a pushbutton:
For this reason, *SELECT*, `F2`, and double-clicking the mouse all trigger the event AT USER-COMMAND. In the corresponding processing block, list levels 1 and 2 have the status DIALOG and appear in a dialog box. In the status DIALOG, exactly as for status BASIC, the function code PICK proposed for function key `F2` due to status type *Dialog box* is replaced by the function code SELE and assigned to the application toolbar after RW.
Titles WI1 and WI2 are defined for the dialog boxes. In the second dialog box, the EXCLUDING option of the SET PF-STATUS statement deactivates function code SELE.
The dialog boxes are displayed with horizontal scrollbars, since the list always has the standard list width of the basic list, and cannot be adapted to the dialog box.

Passing Data from Lists to Programs

To effectively use interactive lists for interactive reporting, it is not sufficient for the program to react to events triggered by user actions on the output list. You must also be able to interpret the lines selected by the user and their contents.

To do this, you use information that the interactive lists pass to the program. ABAP provides two ways of passing data:

- Automatic data transfer [Page 875] using system fields.

  You use automatic data transfer mainly for auxiliary data that you need to better localize user actions.

- Program-controlled data transfer [Page 877] using ABAP statements.

  You use program-controlled data transfer to transfer the contents of individual output fields into the processing blocks of the interactive events and to continue processing with these values.

Passing Data Automatically

Automatic data transfer happens by means of the system fields that are filled by the system for each interactive event. For an overview of the relevant system fields, see Detail Lists [Page 855].

System fields provide you with information about the list index, the position of the list in the output window, and the cursor position. The only system field that contains the contents of the selected line is SY-LISEL.

The example below demonstrates how the system fills these system fields during interactive events.

```
REPORT demo_list_system_fields NO STANDARD PAGE HEADING
                                 LINE-COUNT 12 LINE-SIZE 40.

DATA: l TYPE i, t(1) TYPE c.

DO 100 TIMES.
  WRITE: / 'Loop Pass:', sy-index.
ENDDO.

TOP-OF-PAGE.

  WRITE: 'Basic List, Page', sy-pagno.
  ULINE.

TOP-OF-PAGE DURING LINE-SELECTION.

  WRITE 'Secondary List'.
  ULINE.

AT LINE-SELECTION.

  DESCRIBE FIELD sy-lisel LENGTH l TYPE t.

  WRITE: 'SY-LSIND:', sy-lsind,
       / 'SY-LISTI:', sy-listi,
       / 'SY-LILLI:', sy-lilli,
       / 'SY-CUROW:', sy-curow,
       / 'SY-CUCOL:', sy-cucol,
       / 'SY-CPAGE:', sy-cpage,
       / 'SY-STARO:', sy-staro,
       / 'SY-LISEL:', 'Length =', l, 'Type =', t,
       /  sy-lisel.
```

This program creates a list of ten pages. When you run the program, you can place the cursor within the basic list and create a detail list by choosing *Choose*. This displays the contents of the system fields.

SY-LSIND is the index of the current list, SY-LISTI is the index of the previous list. SY-LILLI is the number of the selected line in the list, SY-CUROW contains the line number of the selected line on the current screen. SY-CUCOL is the position of the cursor in the window. This position exceeds the corresponding unscrolled list column by one. SY-CPAGE is the currently displayed page of the list. SY-STARO is the number of the topmost actual list line displayed on the current page. This does not include the page header. SY-CPAGE and SY-STARO do not depend on the cursor

position. For SY-LISEL, the program displays length, data type, and contents. The length of SY-LISEL is always 255, independent of the list's width.

Using SY-LISEL

The system field SY-LISEL is a type C field with length 255. Although it contains the selected line, it is only of limited use for passing the values of single fields, as it is a character string. To process certain parts of SY-LISEL, you must specify the corresponding offsets.

To pass data, it is therefore a better idea to use one of the methods described in Passing Data by Program Statements [Page 877]. SY-LISEL is, however, suitable for designing the headers of detail lists, or for checking that a selected line is not a space or an underscore.

```
REPORT demo_list_sy_lisel NO STANDARD PAGE HEADING.

DATA num TYPE i.

SKIP.
WRITE 'List of Quadratic Numbers between One and Hundred'.
SKIP.
WRITE 'List of Cubic Numbers between One and Hundred'.

TOP-OF-PAGE.
  WRITE 'Choose a line!'.
  ULINE.

TOP-OF-PAGE DURING LINE-SELECTION.
  WRITE sy-lisel.
  ULINE.

AT LINE-SELECTION.
  IF sy-lisel(4) = 'List'.
    CASE sy-lilli.
      WHEN 4.
        DO 100 TIMES.
          num = sy-index ** 2.
          WRITE: / sy-index, num.
        ENDDO.
      WHEN 6.
        DO 100 TIMES.
          num = sy-index ** 3.
          WRITE: / sy-index, num.
        ENDDO.
    ENDCASE.
  ENDIF.
```

When you start the program, a list appears on which two lines can be selected.

If the user chooses the top line, a list of square numbers is displayed. If the user selects the lower line on the basic list, the list of cubic numbers appears.

The contents of the selected line becomes the header of the detail list. The IF statement uses the first four characters of SY-LISEL to make sure that only valid lines are selected. Due to the logical conditions of the processing block following AT LINE-SELECTION, a line selection on a detail list does not produce any further output.

Passing Data by Program Statements

To pass individual output fields or additional information from a line to the corresponding processing block during an interactive event, use these statements:

- HIDE

  The HIDE statement is one of the fundamental statements for interactive reporting. You use the HIDE technique when creating a basic list. It defines the information that can be passed to subsequent detail lists.

- READ LINE

  Use the statements READ LINE and READ CURRENT LINE to read data from the lines of existing list levels. These statements are closely connected to the HIDE technique.

- GET CURSOR

  Use the statements GET CURSOR FIELD and GET CURSOR LINE to pass the output field or output line on which the cursor was positioned during the interactive event to the ABAP program.

- DESCRIBE LIST

  The DESCRIBE LIST statement allows you to read certain list attributes, such as the number of lines or pages, into program variables.

The following sections describe these statements in more detail:

The HIDE Technique

You use the HIDE technique while creating a list level to store line-specific information for later use. To do so, use the HIDE statement as follows:

```
HIDE <f>.
```

This statement places the contents of the variable <f> for the current output line (system field SY-LINNO) into the HIDE area. The variable <f> must not necessarily appear on the current line.

To make your program more readable, always place the HIDE statement directly after the output statement for the variable <f> or after the last output statement for the current line.

As soon as the user selects a line for which you stored HIDE fields, the system fills the variables in the program with the values stored. A line can be selected

- by an interactive event.

  For each interactive event, the HIDE fields of the line on which the cursor is positioned during the event are filled with the stored values.

- by the READ LINE statement.

You can think of the HIDE area as a table, in which the system stores the names and values of all HIDE fields for each list and line number. As soon as they are needed, the system reads the values from the table.

The example below presents some of the essential features of interactive reporting. The basic list contains summarized information. By means of the HIDE technique, each detail list contains more details.

The following program is connected to the logical database F1S.

```
REPORT demo_list_hide NO STANDARD PAGE HEADING.

TABLES: spfli, sbook.

DATA: num TYPE i,
      dat TYPE d.

START-OF-SELECTION.
  num = 0.
  SET PF-STATUS 'FLIGHT'.

GET spfli.
  num = num + 1.
  WRITE: / spfli-carrid, spfli-connid,
           spfli-cityfrom, spfli-cityto.
  HIDE:    spfli-carrid, spfli-connid, num.

END-OF-SELECTION.
  CLEAR num.

TOP-OF-PAGE.
  WRITE 'List of Flights'.
  ULINE.
  WRITE 'CA  CONN FROM                 TO'.
  ULINE.

TOP-OF-PAGE DURING LINE-SELECTION.
  CASE sy-pfkey.
    WHEN 'BOOKING'.
      WRITE sy-lisel.
      ULINE.
    WHEN 'WIND'.
      WRITE:  'Booking', sbook-bookid,
           /  'Date   ', sbook-fldate.
      ULINE.
  ENDCASE.

AT USER-COMMAND.
  CASE sy-ucomm.
    WHEN 'SELE'.
      IF num NE 0.
        SET PF-STATUS 'BOOKING'.
        CLEAR dat.
        SELECT * FROM sbook WHERE carrid = spfli-carrid
                            AND   connid = spfli-connid.
          IF sbook-fldate NE dat.
            dat = sbook-fldate.
            SKIP.
            WRITE / sbook-fldate.
            POSITION 16.
          ELSE.
            NEW-LINE.
            POSITION 16.
          ENDIF.
          WRITE sbook-bookid.
          HIDE: sbook-bookid, sbook-fldate, sbook-custtype,
```

```
                sbook-smoker, sbook-luggweight, sbook-class.
        ENDSELECT.
        IF sy-subrc NE 0.
          WRITE / 'No bookings for this flight'.
        ENDIF.
        num = 0.
        CLEAR sbook-bookid.
      ENDIF.
    WHEN 'INFO'.
      IF NOT sbook-bookid IS INITIAL.
        SET PF-STATUS 'WIND'.
        SET TITLEBAR 'BKI'.
        WINDOW STARTING AT 30 5 ENDING AT  60 10.
        WRITE: 'Customer type   :', sbook-custtype,
             / 'Smoker          :', sbook-smoker,
             / 'Luggage weight :', sbook-luggweight UNIT 'KG',
             / 'Class           :', sbook-class.
      ENDIF.
  ENDCASE.
```

At the event START-OF-SELECTION, the system sets the status FLIGHT for the basic list. In status FLIGHT, function code SELE (text *SELECT*) is assigned to function key `F2` and to a pushbutton. So the event AT USER-COMMAND is triggered if the user double-clicks, presses F2, or chooses the pushbutton *SELECT*.

The three fields SPFLI-CARRID, SPFLI-CONNID, and NUM are stored in the HIDE area while creating the basic list. After selecting a line, the system displays the detail list defined in the AT USER-COMMAND event for function code SELE. In the AT USER-COMMAND event, the system refills all fields of the selected line that were stored in the HIDE area. You use NUM to check whether the user selected a line from the actual list. The detail list has status BOOKING, where `F2` is assigned to function code INFO (text: *Booking Information*). The detail list presents data that the program selected by means of the HIDE fields of the basic list. For each list line displayed, the system stores additional information in the HIDE area.

If the user selects a line of the detail list, the system displays the "hidden" information in a dialog box with the status WIND. The status has the type *Dialog box*, and contains the proposed functions for a list status. The program uses SBOOK-BOOKID to check whether the user selected a valid line.

The program itself sets all page headers and the title bar of the dialog box.

Reading Lines from Lists

All of the lists generated by a single program are stored internally in the system. You can therefore access any list in a program that was created for the same screen and that has not yet been deleted by returning to a lower list level. To read lines, use the statements READ LINE and READ CURRENT LINE.

To read a line from a list after an interactive list event, use the READ LINE statement:

```
READ LINE <lin> [INDEX <idx>]
        [FIELD VALUE <f1> [INTO <g1>] ... <fn> [INTO <gn>]]
        [OF CURRENT PAGE|OF PAGE <p>].
```

Passing Data by Program Statements

The statement without any options stores the contents of line <lin> from the list on which the event was triggered (index SY-LILLI) in the SY-LISEL system field and fills all HIDE information stored for this line back into the corresponding fields. As far as SY-LISEL and the HIDE area are concerned, READ LINE has the same effect as an interactive line selection.

If the selected line <lin> exists, the system sets SY-SUBRC to 0, otherwise to 4.

The options have the following effects:

- INDEX <idx>

  The system reads the information for line <lin> from the list of level <idx>.

- FIELD VALUE <f_1> [INTO <g_1>] ... <f_n> [INTO <g_n>]

  The system interprets the output values of the variables <f_i> in line <lin> as character strings and places them either into the same fields <f_i> or, when using INTO, into the fields <g_i>.When refilling the fields, the system applies the conversion rules.

  Fields that do not appear in a line do not affect the target field. If a field appears several times in a line, the system uses only the first one.

  The system transports the field contents using the output format, that is, including all formatting characters. This may cause problems, such as converting editing characters to decimal characters or other incompatible cases.

  You use this option mainly to process user entries in list fields that accept input, since you cannot use the HIDE technique in this case.

- OF CURRENT PAGE

  With this option, <lin> is not the number of the line of the entire list, but the number of the line of the currently displayed page of the addressed list. The system field SY-CPAGE stores the corresponding page number.

- OF PAGE <p>

  With this option, <lin> is not the number of the line of the entire list, but the number of a line on page <p> of the addressed list.

This statement reads a line twice in succession. To do this, you use the READ CURRENT LINE statement in your program:

```
READ CURRENT LINE [FIELD VALUE <f1> [INTO <g1>] ...].
```

This statement reads a line read before by an interactive event (F2) or by READ LINE. The FIELD VALUE option is the same as for READ LINE.

This allows you to fill fields in the same line using values from the program.

The following program is connected to the logical database F1S.

```
REPORT demo_list_read_line NO STANDARD PAGE HEADING.

TABLES: sflight.

DATA: box(1) TYPE c, lines TYPE i, free TYPE i.

START-OF-SELECTION.
  SET PF-STATUS 'CHECK'.
```

```
GET sflight.
  WRITE: box AS CHECKBOX, sflight-fldate.
  HIDE: sflight-fldate, sflight-carrid, sflight-connid,
        sflight-seatsmax, sflight-seatsocc.

END-OF-SELECTION.
  lines = sy-linno - 1.

TOP-OF-PAGE.
  WRITE: 'List of flight dates'.
  ULINE.

TOP-OF-PAGE DURING LINE-SELECTION.
  WRITE:  'Date:', sflight-fldate.
  ULINE.

AT USER-COMMAND.
  CASE sy-ucomm.
    WHEN 'READ'.
      box = space.
      SET PF-STATUS 'CHECK' EXCLUDING 'READ'.
      DO lines TIMES.
        READ LINE sy-index FIELD VALUE box.
        IF box = 'X'.
          free = sflight-seatsmax - sflight-seatsocc.
          IF free > 0.
            NEW-PAGE.
            WRITE: 'Company:', sflight-carrid,
                   'Connection: ',sflight-connid,
                 / 'Number of free seats:', free.
          ENDIF.
        ENDIF.
      ENDDO.
  ENDCASE.
```

After the selection screen has been processed, a basic list appears, on which the user can select checkboxes. The user-defined page header for the basic list uses the parameters CITY_FR and CITY_TO. The parameters are defined in the logical database F1S.

The program uses the status CHECK, where the user-defined function code READ (text *Read Lines*) is assigned to function key F5 and to a pushbutton in the application toolbar. The corresponding user action triggers the AT USER-COMMAND event. The system now reads all lines of the basic list using READ LINE in a DO loop. For each line, it fills the corresponding fields with all values previously stored in the HIDE area. By means of the FIELD VALUE option, the system in addition reads the checkbox BOX. If seats are still free, the system writes the company, the connection, and the number of free seats into a detail list for each line in which the checkbox was selected.

The system starts a new page with an individual page header for each output. On the detail list, the function code READ is deactivated using the EXCLUDING addition of the SET PF-STATUS statement.

After returning from the detail list, the checkboxes are still filled.

Reading Lists at the Cursor Position

To retrieve information about the current cursor position in an interactive event, use the GET CURSOR statement. You can retrieve information either about the current field or the current line.

For field information, use this syntax:

```
GET CURSOR FIELD <f> [OFFSET <off>] [LINE <lin>]
           [VALUE <val>] [LENGTH <len>].
```

This statement transfers the name of the field on which the cursor is positioned during a user action into the variable <f>. If the cursor is on a field, the system sets SY-SUBRC to 0, otherwise to 4.

The system transports the names of global variables, constants, field symbols, or reference parameters of subroutines. For literals, local fields, and VALUE parameters of subroutines, the system sets SY-SUBRC to 0, but transfers SPACE as the name.

The options have the following effects:

- OFFSET <off>

  The field <off> contains the position of the cursor within the field. If the cursor is on the first column, <off> = 0.

- LINE <lin>

  The field <lin> contains the number of the list line on which the cursor is positioned (SY-LILLI).

- VALUE <val>

  The field <val> contains the character string output representation of the field on which the cursor is positioned. The representation includes formatting characters.

- LENGTH <len>

  The field <len> contains the output length of the field on which the cursor is positioned.

For field information, use this syntax:

```
GET CURSOR LINE <lin> [OFFSET <off>] [VALUE <val>] [LENGTH <len>].
```

This statement transfers the number of the line on which the cursor is positioned during a user action into the variable <lin>. If the cursor is on a list line, the system sets SY-SUBRC to 0, otherwise to 4. You can use this statement to prevent the user from selecting invalid lines.

The options have the following effects:

- OFFSET <off>

  The field <off> contains the position of the cursor within the list line. If the cursor is on the first column, <off> = 0.

- VALUE <val>

  The field <val> contains the character string output representation of the line on which the cursor is positioned. The representation includes formatting characters.

- LENGTH <len>

  The field <len> contains the output length of the line on which the cursor is positioned.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING LINE-SIZE 40.

DATA: HOTSPOT(10) VALUE 'Click me!',
      F(10), OFF TYPE I, LIN TYPE I, VAL(40), LEN TYPE I.

FIELD-SYMBOLS <FS>.
ASSIGN HOTSPOT TO <FS>.
WRITE 'Demonstration of GET CURSOR statement'.
SKIP TO LINE 4.
POSITION 20.
WRITE <FS> HOTSPOT COLOR 5 INVERSE ON.

AT LINE-SELECTION.
  WINDOW STARTING AT 5 6 ENDING AT 45 20.
  GET CURSOR FIELD F OFFSET OFF
             LINE LIN VALUE VAL LENGTH LEN.
  WRITE: 'Result of GET CURSOR FIELD: '.
  ULINE AT /(28).
  WRITE: / 'Field: ', F,
         / 'Offset:', OFF,
         / 'Line:  ', LIN,
         / 'Value: ', (10) VAL,
         / 'Length:', LEN.
  SKIP.
  GET CURSOR LINE LIN OFFSET OFF VALUE VAL LENGTH LEN.
  WRITE: 'Result of GET CURSOR LINE: '.
  ULINE AT /(27).
  WRITE: / 'Offset:', OFF,
         / 'Value: ', VAL,
         / 'Length:', LEN.
```

In this program, the HOTSPOT field is assigned to the field symbol <FS> and displayed as hotspot on the output screen. If the user positions the cursor on a list line and selects it, a dialog box appears containing the results of the GET CURSOR statements in the AT LINE-SELECTION event.

Note that after GET CURSOR FIELD, the name of the field assigned to the field symbol <FS> is stored in F, and not the name of the field symbol.

Determining the Attributes of Lists

If you need to know the attributes of list levels that are not stored in system variables, you can use the DESCRIBE LIST statement.

To retrieve the number of lines or pages of a list, use:

```
DESCRIBE LIST NUMBER OF LINES|PAGES <n> [INDEX <idx>].
```

This statement writes the number of lines or pages of the list level <idx> into the variable <n>. If a list with index <idx> does not exist, the system sets SY-SUBRC unequal to 0, otherwise to 0.

To retrieve the page number for a certain line number, use:

```
DESCRIBE LIST LINE <lin> PAGE <pag> [INDEX <idx>].
```

Passing Data by Program Statements

This statement writes for list level <idx> the page number on which the list line number <lin> is found into the variable <pag>. SY-SUBRC is set as follows: If there is no list with the index <idx>, it is 8. If there is no line with number <line>, it is 4. Otherwise, it is 0.

To retrieve the attributes of a certain page, use:

```
DESCRIBE LIST PAGE <pag> [INDEX <idx>] [<options>]
```

This statement retrieves for list level <idx> the attributes specified in <options> for page <pag>. SY-SUBRC is set as follows: If there is no list with the index <idx>, it is 8. If there is no page with number <pag>, it is 4. Otherwise, it is 0.

The <options> of the statement are:

- LINE-SIZE <col>

  writes the page width into the variable <col>.

- LINE-COUNT <len>

  writes the page length into the variable <len>.

- LINES <lin>

  writes the number of displayed lines into the variable <lin>.

- FIRST-LINE <lin1>

  writes the absolute number of the first line into the variable <lin1>.

- TOP-LINES <top>

  writes the number of page header lines into the variable <top>.

- TITLE-LINES <ttl>

  writes the number of list header lines of the standard page header into the variable <ttl>.

- HEAD-LINES <head>

  writes the number of column header lines of the standard page header into the variable <head>.

- END-LINES <end>

  writes the number of page footer lines into the variable <end>.

Use DESCRIBE LIST for completed lists only, since for lists in creation (index is SY-LSIND) some attributes are not up to date.

```
REPORT demo_list_describe_list NO STANDARD PAGE HEADING
                               LINE-SIZE 40 LINE-COUNT 5(1).

DATA: lin TYPE i, pag TYPE i,
      col TYPE i, len TYPE i, lin1 TYPE i,
      top TYPE i, tit TYPE i, head TYPE i, end TYPE i.

DO 4 TIMES.
  WRITE / sy-index.
ENDDO.
```

```
TOP-OF-PAGE.
  WRITE 'Demonstration of DESCRIBE LIST statement'.
  ULINE.

END-OF-PAGE.
  ULINE.

AT LINE-SELECTION.
  NEW-PAGE LINE-COUNT 0.
  WINDOW STARTING AT 1 13 ENDING AT 40 25.
  DESCRIBE LIST: NUMBER OF LINES lin INDEX 0,
                 NUMBER OF PAGES pag INDEX 0.
  WRITE: 'Results of DESCRIBE LIST: '.
  ULINE AT /(25).
  WRITE: / 'Lines: ', lin,
         / 'Pages: ', pag.
  SKIP.
  DESCRIBE LIST LINE sy-lilli PAGE pag INDEX 0.
  WRITE: / 'Line', (1) sy-lilli, 'is on page', (1) pag.
  SKIP.
  DESCRIBE LIST PAGE pag INDEX 0 LINE-SIZE   col
                                 LINE-COUNT  len
                                 LINES       lin
                                 FIRST-LINE  lin1
                                 TOP-LINES   top
                                 TITLE-LINES tit
                                 HEAD-LINES  head
                                 END-LINES   end.
  WRITE: 'Properties of Page', (1) pag, ':',
       / 'Width:       ',  col,
       / 'Length:      ',  len,
       / 'Lines:       ',  lin,
       / 'First Line:  ',  lin1,
       / 'Page Header: ',  top,
       / 'Title Lines: ',  tit,
       / 'Header Lines:',  head,
       / 'Footer Lines:',  end.
```

This program creates a two-page list of five lines per page. Two lines are used for the self-defined page header and one line for the page footer. If the user selects a line, a dialog box appears containing the list attributes.

While creating the secondary list, all variants of the DESCRIBE LIST statement apply to the basic list. The system displays the results in the dialog window. The lines and pages to be described are addressed dynamically using SY-LILLI.

Manipulating Detail Lists

This section describes how you can manipulate the appearance and attributes of detail lists.

Scrolling in Detail Lists

You can scroll in a detail list using the SCROLL statement. The Scrolling in Lists [Page 815] section contains a full description of the statement and how to use it for basic lists.

When you use the SCROLL statement with detail lists, you must remember the following:

- You can only use the SCROLL statement for completed lists. If you use SCROLL before the first output statement of a list, it does not affect the list. If you use SCROLL after the first output statement of a list, it affects the entire list, that is, all subsequent output statements as well.
- When you create a secondary list, a SCROLL statement without INDEX option always refers to the previously displayed list on which the interactive event occurred (index SY-LISTI).
- Only when creating the basic list does the SCROLL statement refer to the list currently being created.
- You can use the INDEX option to scroll explicitly on existing list levels. To do this, the lists need not be displayed. When the user displays the list again, it is scrolled to the specified position. If the specified list level does not exist, the system sets SY-SUBRC to 8.
- If, during an interactive event, you want to scroll through the list you are currently creating, use SY-LSIND as the index in the SCROLL statement. Note that changing SY-LSIND only takes effect at the end of the event, regardless of where you change it in the processing block. If you want to set the list level explicitly, you can change SY-LSIND in the last statement of the processing block. This ensures that a SCROLL statement within the processing block accesses the correct list.

Another way of scrolling interactive lists from within the program is to use the SET USER-COMMAND [Page 868] statement in conjunction with the corresponding predefined function codes (P...).

```
REPORT demo_list_scroll NO STANDARD PAGE HEADING LINE-SIZE
50.

SET PF-STATUS 'SELECT'.

WRITE 'Create a secondary list by choosing SELECT'.

AT USER-COMMAND.
  NEW-PAGE LINE-SIZE 200.
  CASE sy-ucomm.
    WHEN 'SELE'.
      SET PF-STATUS 'SCROLLING'.
      DO 200 TIMES. WRITE sy-index. ENDDO.
      SCROLL LIST RIGHT BY 48 PLACES INDEX sy-lsind.
      sy-lsind = sy-lsind - 1.
    WHEN 'LEFT'.
      SCROLL LIST LEFT BY 12 PLACES.
    WHEN 'RGHT'.
      SCROLL LIST RIGHT BY 12 PLACES.
  ENDCASE.
```

Scrolling in Detail Lists

This program creates a basic list of one line with the status SELECT. In the status SELECT, the function code SELE (text (*SELECT*) is assigned to function key `F2` and to a pushbutton in the application toolbar.

After choosing *SELECT*, the system triggers the AT USER-COMMAND event and creates a detail list with status SCROLLING. In the status SCROLLING, the function codes *LEFT* (text LEFT) and *RGTH* (text *RIGHT)* are assigned to the function keys `F5` and `F6` and to the application toolbar. The detail list is 200 characters wide. The SCROLL statement scrolls the detail list (SY-LSIND = 1) by 48 columns to the right after it has been created. Then, SY-LSIND is decreased by 1 and the scrolled list replaces the basic list.

By clicking on *LEFT* and *RIGHT,* the user can scroll to the left and to the right in the displayed list. The SCROLL statements are programmed for the corresponding function codes within the AT USER-COMMAND event.

Setting the Cursor from within the Program

You can set the cursor on the current list dynamically from within your program. You can do this to support the user with entering values into input fields or selecting fields or lines. If input fields occur on a list, the system by default places the cursor into the first input field.

To set the cursor, use the SET CURSOR statement. This statement sets the cursor in the most recently-created list. While the basic list is being created, this is always the basic list itself. For a detail list, it is the previous list.

With SET CURSOR, you can set the cursor to an absolute position, to a field, or to a line.

Setting the Cursor Explicitly

To set the cursor to a certain position in the output window, use:

```
SET CURSOR <col> <lin>.
```

This statement sets the cursor to column <col> of line <lin> of the output window.

The system sets the cursor only to positions that are visible in the display. For positions outside the displayed area, it ignores the statement. To set the cursor to a part of the list currently not displayed, you must scroll the list first.

You can set the cursor only to lines that contain list output. These include lines skipped with the SKIP statement, but no underlines. If <lin> is below the bottom list line, the system sets the cursor to the bottom line.

```
REPORT demo_list_set_cursor_1 NO STANDARD PAGE HEADING LINE-
SIZE 80.

SET PF-STATUS 'SCROLLING'.

NEW-LINE NO-SCROLLING.
WRITE 'Input Fields:'.
NEW-LINE NO-SCROLLING.
WRITE '-------------'.

NEW-LINE.
DO 5 TIMES.
  WRITE:  ' Input', (1) sy-index, '   ' INPUT ON NO-GAP.
ENDDO.

FORMAT INPUT OFF.
SET CURSOR 11 3.

AT USER-COMMAND.
  CASE sy-ucomm.
    WHEN 'LEFT'.
      IF sy-staco > 1.
        SCROLL LIST LEFT BY 12 PLACES.
      ENDIF.
    WHEN 'RGHT'.
      IF sy-staco < 40.
        SCROLL LIST RIGHT BY 12 PLACES.
      ENDIF.
```

```
  ENDCASE.
  SET CURSOR 11 3.
```

This program creates a basic list that contains five input fields. The cursor is set to the first input field. The user can use the pushbuttons *LEFT* and *RIGHT* to scroll the list horizontally. After each scroll movement, the cursor is set to an input field again.

Setting the Cursor to a Field

To set the cursor to a certain field on a line of the displayed list, use:

```
SET CURSOR FIELD <f> LINE <lin> [OFFSET <off>].
```

This statement sets the cursor on line <lin> onto the field whose name is stored in <f>. If a field appears more than once on a line, the system sets the cursor to the first field. If the field does not appear on the line or if it is outside the displayed area, the system ignores the statement. You can use the SCROLL statement to scroll the line into the visible area of the screen.

Use the OFFSET option to set the cursor to position <off> of the field stored in <f>. <off> = 0 indicates the first position.

When you set the cursor, you must take into account the header lines of the list.

```
REPORT demo_list_set_cursor_2 LINE-SIZE 50.

DATA: input1(5) TYPE c VALUE '*****',
      input2(5) TYPE c VALUE '*****',
      input3(5) TYPE c VALUE '*****'.

SET PF-STATUS 'INPUT'.

WRITE   'Input Fields:'.
WRITE / '-------------'.
SKIP.

WRITE: 'Input 1', input1 INPUT ON,
     / 'Input 2', input2 INPUT ON,
     / 'Input 3', input3 INPUT ON.

AT USER-COMMAND.
  CASE sy-ucomm.
    WHEN 'INP1'.
      SET CURSOR FIELD 'INPUT1' LINE 6 OFFSET 4.
    WHEN 'INP2'.
      SET CURSOR FIELD 'INPUT2' LINE 7 OFFSET 4.
    WHEN 'INP3'.
      SET CURSOR FIELD 'INPUT3' LINE 8 OFFSET 4.
  ENDCASE.
```

This program creates a basic list containing three input fields. In the status INPUT, the function codes INP1, INP2, and INP3 are assigned to the function keys `F5`, `F6`, and `F7` and to the application toolbar. When you choose one of these functions, the system triggers the AT USER-COMMAND event. It places the cursor on the corresponding input field.

Setting the Cursor to a Line

To set the cursor to a certain line of the list in the output window, use:

```
SET CURSOR LINE <lin> [OFFSET <off>].
```

This statement sets the cursor to line <lin>. The system ignores the statement, if the line does not appear in the list or in the visible area of the window. You can use the SCROLL statement to scroll the line into the visible area of the screen.

Use the OFFSET option to set the cursor to column <off> of line <lin>. <off> = 0 indicates the first column. The system ignores the statement, if the position is outside the visible area of the list.

When you set the cursor, you must take into account the header lines of the list.

```
REPORT demo_list_set_cursor_3 LINE-SIZE 30
                              NO STANDARD PAGE HEADING.

DATA: inp(2) TYPE c, top(2) TYPE c.

SET PF-STATUS 'LINE_NUMBER'.

DO 50 TIMES.
  WRITE: / 'Line ', (2) sy-index.
ENDDO.

TOP-OF-PAGE.
  WRITE: 'Line number:', inp INPUT ON.
  ULINE.
  SKIP.

AT USER-COMMAND.
  DESCRIBE LIST PAGE 1 TOP-LINES top.
  CASE sy-ucomm.
    WHEN 'LINE'.
      READ LINE 1 FIELD VALUE inp.
      SCROLL LIST TO PAGE 1 LINE inp.
      inp = inp + top.
      SET CURSOR LINE inp OFFSET 6.
  ENDCASE.
```

This program creates a basic list with an input field and a set of lines.

In the status LINE_NUMBER, the function code LINE (text *LINE NUMBER*) is assigned to function key F5 and to a pushbutton of the application toolbar. If the user enters a number into the input field and chooses *LINE NUMBER*, the system scrolls to the specified number and sets the cursor to it:

The system reads the input field using READ LINE. For setting the cursor, it uses DESCRIBE LIST to take into account the size of the page header. Note that this example makes excessive use of automatic type conversion.

Modifying List Lines

To modify the lines of a completed list from within the program, use the MODIFY LINE statement. There are two ways to specify the line you want to modify:

- Explicitly:

  ```
  MODIFY LINE <n> [INDEX <idx>|OF CURRENT PAGE|OF PAGE <p>]
                  [<modifications>].
  ```

  Without the first line of options, this statement modifies line <n> of the list on which an interactive event occurred (index SY-LISTI). With the options of the first line, you can specify the line you want to modify as follows:

  – Use INDEX <idx> to specify line <n> of the list level with the index <idx>.

  – Use OF CURRENT PAGE to specify line <n> of the currently displayed page (page number SY-CPAGE) .

  – Use OF PAGE <p> to specify line <n> of page <p>.

  You can refer to the line most recently read:

  ```
  MODIFY CURRENT LINE [<modifications>].
  ```

  This statement modifies the line most recently read by means of line selection (F2) or of the READ LINE statement.

Without the option <modifications>, the above statements fill the current contents of the SY-LISEL system field into the specified line. The line's HIDE area is overwritten by the current values of the corresponding fields. However, this does not influence the displayed values.

If the system succeeded in modifying the specified line, it sets SY-SUBRC to 0, otherwise to a value unequal to 0.

Apart from the ones described above, the option <modifications> contains several other possibilities to modify the line.

Modifying Line Formatting

To modify the formatting of the line you want to change, use the option LINE FORMAT of the MODIFY statement as follows:

```
MODIFY ... LINE FORMAT <option1> <option2> ... .
```

This statement sets the output format of the entire modified line according to the format specified with <option$_i$>. You can specify the same format options as for the FORMAT statement.

Modifying Field Contents

To explicitly modify the contents of fields in the line you want to change, use the option FIELD VALUE of the MODIFY statement:

```
MODIFY ... FIELD VALUE <f1> [FROM <g1>] <f2> [FROM <g2>] ... .
```

This statement overwrites the contents of the fields <f_i> in the list line with the current contents of the fields <f_i> or <g_i>. If necessary, the system converts the field type to type C.

If a field <f_i> occurs more than once in the line, the system modifies only the first. If a field <f_i> does not occur in the line at all, the system ignores the option.

The system modifies existing fields <f_i> regardless of the current contents you write from SY-LISEL into the line. If you made changes to the line at the output position of a field <f_i> using SY-LISEL, the FIELD VALUE option overwrites them.

Changing Field Formatting

To modify the formatting of fields in the line you want to change, use the option FIELD FORMAT of the MODIFY statement as follows:

```
MODIFY ... FIELD FORMAT <f1> <options1> <f2> <options2> ... .
```

This statement sets the output format of the fields <f_i> occurring in the line according to the format specified in <options$_i$>. You can specify formats from the FORMAT statement in <optionsi>.

The option FIELD FORMAT overwrites the specifications of the LINE FORMAT option for the corresponding field(s). If a field <f_i> occurs more than once in the line, the system modifies only the first. If a field <f_i> does not occur in the line at all, the system ignores the option.

Examples

Example of line formatting.

```
REPORT demo_list_modify_line_format LINE-SIZE 40
                                    NO STANDARD PAGE HEADING.
DATA c TYPE i VALUE 1.
WRITE 'Select line to modify the background'.
AT LINE-SELECTION.
  IF c = 8.
     c = 0.
  ENDIF.
  MODIFY CURRENT LINE LINE FORMAT COLOR = c.
  ADD 1 TO c.
```

This program creates an output line whose background color the user can change by selecting the line again and again:

Example for field contents.

```
REPORT demo_list_modify_field_value LINE-SIZE 40
                                    NO STANDARD PAGE HEADING.
DATA c TYPE i.
WRITE: '   Number of selections:', (2) c.
AT LINE-SELECTION.
  ADD 1 TO c.
  sy-lisel(2) = '**'.
  MODIFY CURRENT LINE FIELD VALUE c.
```

This program creates an output line, in which the user can modify the field C by selecting the line. At the same time, the system overwrites the first two characters of the line with two asterisks '**' due to the modification of SY-LISEL.

Example for field formatting.

```
REPORT demo_list_modify_field_format NO STANDARD PAGE HEADING.
```

Modifying List Lines

```
DATA: box(1) TYPE c, lines TYPE i, num(1) TYPE c.
SET PF-STATUS 'CHECK'.
DO 5 TIMES.
  num = sy-index.
  WRITE: / box AS CHECKBOX, 'Line', num.
  HIDE: box, num.
ENDDO.
lines = sy-linno.
TOP-OF-PAGE.
  WRITE  'Select some checkboxes'.
  ULINE.
AT USER-COMMAND.
  CASE sy-ucomm.
    WHEN 'READ'.
      SET PF-STATUS 'CHECK' EXCLUDING 'READ'.
      box = space.
      DO lines TIMES.
        READ LINE sy-index FIELD VALUE box.
        IF box = 'X'.
          WRITE: / 'Line', num, 'was selected'.
          box = space.
          MODIFY LINE sy-index
                      FIELD VALUE  box
                      FIELD FORMAT box INPUT OFF
                                   num COLOR 6 INVERSE ON.
        ENDIF.
      ENDDO.
  ENDCASE.
```

This program creates a basic list with the status CHECK. In the status CHECK, function code READ (text *Read Lines*) is assigned to function key F5 and to a pushbutton. The user can mark checkboxes and then choose *Read Lines*.
In the AT USER-COMMAND event, the system reads the lines of the list using READ LINE. It continues processing the selected lines on a secondary list. When returning to the basic list, the system deletes the marks in the checkboxes of the selected lines using MODIFY LINE and sets the format INPUT OFF to the checkboxes. In addition, it changes the format of field NUM.
The user can now mark only those lines that have not yet been changed.

Lists and Screens

When you run an executable program, the list processor is automatically started at the end of the last processing block, and the basic list created during the program is displayed.

If you want to display lists during screen processing, you must program it explicitly. Conversely, you can switch to screen processing from list processing.

Starting Lists from Screen Processing [Page 896]

Calling Screens from List Processing [Page 900]

Starting Lists from Screen Processing

This section describes how to switch from screen processing to list processing. It contains a short technical introduction, followed by a recommended procedure.

Switching Between Screen and List Processing

Screen processing always involves a screen sequence [Page 1000] that you start either using CALL SCREEN or a transaction code. During screen processing, the ABAP program is controlled by the dialog processor. In the ABAP program, the PBO and PAI modules are executed as they are called from the screen flow logic.

To pass control from the dialog processor to the list processor, you must include the following statement in one of the dialog modules:

LEAVE TO LIST-PROCESSING [AND RETURN TO SCREEN <nnnn>].

You can include this statement in either the PBO or the PAI event. Its effect is to start the list processor and display the basic list **after the PAI processing of the current screen**. The basic list contains any list output from all PBO and PAI modules that have been executed up to that point.

If detail lists are defined in the corresponding event blocks of the ABAP program (AT LINE-SELECTION, AT USER-COMMAND), user actions on the basic list will lead to the detail list, and further interaction will lead to further list levels.

You can leave list processing in two ways:

- By leaving the basic list using the *Back*, *Exit*, or *Cancel* function.
- By using the following statement during list processing:

  LEAVE LIST-PROCESSING.

In both cases, control returns from the list processor to the dialog processor. Each time this occurs, the **entire list system is initialized**. Any subsequent list output statements in PBO and PAI modules apply to an empty basic list.

By default, the dialog processor returns to the PBO processing of the screen from which the list processor was called. The optional addition AND RETURN TO SCREEN allows you to specify a different screen in the current screen sequence at whose PBO event you want to resume processing. In particular, the statement

LEAVE TO LIST-PROCESSING AND RETURN TO SCREEN 0.

can be used to end the current screen sequence and return to the point from which it had originally been called.

Recommended Procedure

If you want to display lists during screen processing, you should create a separate screen for each list system that you want to call. This screen **encapsulates** the creation and display of the basic list. It can then be called from anywhere in the program using CALL SCREEN.

The actual screen mask of this screen can remain empty. You do not need any PAI modules, and only a single PBO module. In the PBO module, you define the basic list of the list system and call the list processor.

1. First, use the

   LEAVE TO LIST-PROCESSING AND RETURN TO SCREEN 0.

   statement to call the list display at the end of the screen, and to ensure that, after leaving the list, you return to the point from which the screen was called.
2. Next, set a GUI status for the list; for example, the default list status SPACE or a list status of your own.
3. Use one of the following statements to ensure that the empty screen is not displayed:

   SUPPRESS DIALOG.

   or

   LEAVE SCREEN. Instead, the list is displayed immediately at the end of the screen.
4. Now define the **entire** basic list, and place any necessary data in the HIDE area.

If you want to process user actions on the list, you need to define the relevant event blocks in your ABAP program. If you want to call more than one independent list system in the program, you must ensure that you can tell them apart in the list event processing. You cannot do this using SY-DYNNR, since the container screen for a list is always number 120. Instead, you could assign a different GUI status to each list, and distinguish between the list systems using the value of SY-PFKEY, or you could place some unique information in the HIDE area of each list system.

Example

```
REPORT demo_leave_to_list_processing .

TABLES sdyn_conn.

DATA: wa_spfli TYPE spfli,
      flightdate TYPE sflight-fldate.

CALL SCREEN 100.

MODULE status_0100 OUTPUT.
  SET PF-STATUS 'SCREEN_100'.
ENDMODULE.

MODULE cancel INPUT.
  LEAVE PROGRAM.
ENDMODULE.

MODULE user_command_0100.
  CALL SCREEN 500.
  SET SCREEN 100.
ENDMODULE.

MODULE call_list_500 OUTPUT.
  LEAVE TO LIST-PROCESSING AND RETURN TO SCREEN 0.
  SET PF-STATUS space.
  SUPPRESS DIALOG.
```

```
  SELECT  carrid connid cityfrom cityto
    FROM  spfli
    INTO  CORRESPONDING FIELDS OF wa_spfli
    WHERE carrid = sdyn_conn-carrid.
    WRITE: / wa_spfli-carrid, wa_spfli-connid,
             wa_spfli-cityfrom, wa_spfli-cityto.
    HIDE: wa_spfli-carrid, wa_spfli-connid.
  ENDSELECT.
  CLEAR: wa_spfli-carrid.
ENDMODULE.

TOP-OF-PAGE.
  WRITE text-001 COLOR COL_HEADING.
  ULINE.

TOP-OF-PAGE DURING LINE-SELECTION.
  WRITE sy-lisel COLOR COL_HEADING.
  ULINE.

AT LINE-SELECTION.
  CHECK not wa_spfli-carrid is initial.
  SELECT  fldate
    FROM  sflight
    INTO  flightdate
    WHERE carrid = wa_spfli-carrid AND
          connid = wa_spfli-connid.
     WRITE / flightdate.
  ENDSELECT.
  CLEAR: wa_spfli-carrid.
```

This example switches to list processing during the screen processing for screen 100. Screen 100 has a single input field - the component CARRID from the ABAP Dictionary structure SDYN_CONN.

It has the following flow logic:

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0100.

PROCESS AFTER INPUT.
  MODULE CANCEL AT EXIT-COMMAND.
  MODULE USER_COMMAND_0100.
```

The PAI module USER_COMMAND_100 calls screen 500 using the CALL SCREEN statement. This screen encapsulates a basic list. It has the following flow logic:

```
PROCESS BEFORE OUTPUT.
  MODULE CALL_LIST_500.

PROCESS AFTER INPUT.
```

The module CALL_LIST_500 defines the basic list and switches to list processing. Since the next screen after list processing is screen 0, screen 500 is a one-screen screen chain. After list processing, control returns to the position in USER_COMMAND_100 from which screen 500 was called.

If the user selects a line on the basic list, a detail list appears. This is achieved through the event block AT LINE-SELECTION. The program also contains event

blocks for TOP-OF-PAGE and TOP-OF-PAGE DURING LINE-SELECTION, which define page headers for both the basic list and detail list.

Since there is only one list system in this program, there is no need for case distinctions within the list events.

Calling Screens from List Processing

To call a screen from list processing, use the statement

CALL SCREEN <nnnn>.

This inserts a screen sequence into the program flow as described in the section Calling Screen Sequences [Page 1007]. The list processor passes control to the dialog processor.

The context from the time of the call is retained. If you call a screen sequence during processing of a particular list level, it is processed until the end of a screen with next screen 0. Then the dialog processor returns control to the list processor, and processing carries on after the CALL SCREEN statement.

Example

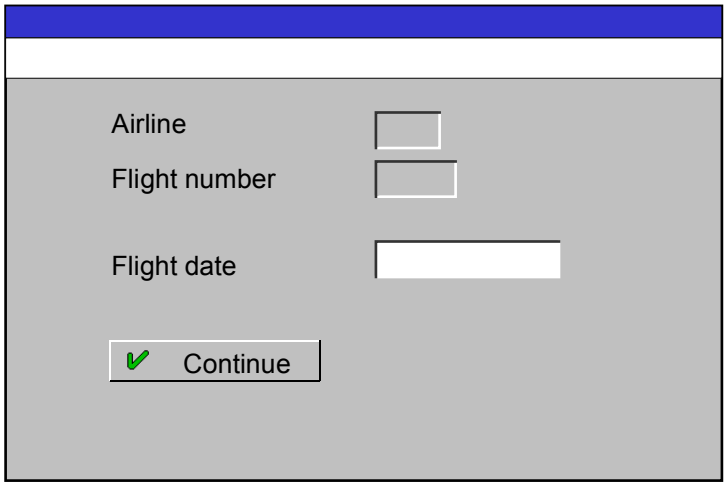
The following program is connected to the logical database F1S.

```
REPORT demo_call_screen_from_list.

NODES  spfli.
TABLES demo_conn.

DATA: ok_code LIKE sy-ucomm,
      save_ok LIKE sy-ucomm.

TOP-OF-PAGE.
  WRITE 'Liste von Flügen'(001) COLOR COL_HEADING.
  ULINE.

GET spfli FIELDS carrid connid.
  WRITE: / spfli-carrid, spfli-connid.
  HIDE:    spfli-carrid, spfli-connid.

END-OF-SELECTION.
  CLEAR:   spfli-carrid, spfli-connid.

AT LINE-SELECTION.
  CHECK not spfli-carrid is initial.
  demo_conn-carrid = spfli-carrid.
  demo_conn-connid = spfli-connid.
  CALL SCREEN 100.
  CLEAR: spfli-carrid, spfli-connid.

MODULE cancel INPUT.
  LEAVE PROGRAM.
ENDMODULE.

MODULE status_0100 OUTPUT.
  SET PF-STATUS 'SCREEN_100'.
ENDMODULE.

MODULE user_command_0100 INPUT.
  save_ok = ok_code.
  CLEAR ok_code.
  CASE save_ok.
```

```
    WHEN 'BACK'.
      LEAVE TO SCREEN 0.
    WHEN OTHERS.
      SELECT  SINGLE *
        FROM  sflight
        INTO  CORRESPONDING FIELDS OF demo_conn
        WHERE carrid = demo_conn-carrid AND
              connid = demo_conn-connid AND
              fldate = demo_conn-fldate.
        IF sy-subrc ne 0.
          MESSAGE e047(sabapdocu).
        ELSE.
          SET SCREEN 200.
        ENDIF.
  ENDCASE.
ENDMODULE.

MODULE status_0200 OUTPUT.
  SET PF-STATUS 'SCREEN_200'.
ENDMODULE.

MODULE user_command_0200 INPUT.
  save_ok = ok_code.
  CLEAR ok_code.
  CASE  save_ok.
    WHEN 'BACK'.
      LEAVE TO SCREEN 100.
    WHEN 'SAVE'.
      ...              "e.g. update records in database
      MESSAGE i888(sabapdocu) WITH text-002.
      SET SCREEN 0.
  ENDCASE.
ENDMODULE.
```

When you run the program, you can enter selections on the selection screen of the logical database, after which a basic list is displayed.

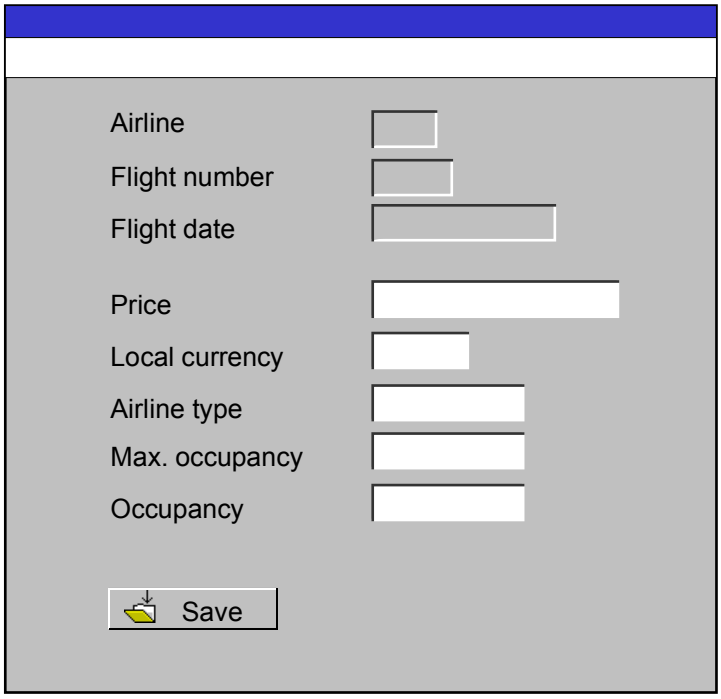
If you select a list line, the AT LINE-SELECTION event is triggered, and screen 100 is called. The contents of two of the screen fields are set in the ABAP program. The layout of screen 100 is:



and its flow logic is:

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0100.

PROCESS AFTER INPUT.
  MODULE CANCEL AT EXIT-COMMAND.
  FIELD DEMO_CONN-FLDATE MODULE USER_COMMAND_0100.
```

You can enter a flight date for the flight that you chose from the list. If you then choose *Continue*, the corresponding data is read from table SFLIGHT, and screen 200 is set as the next screen. The layout of screen 200 is:



and its flow logic is:

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0200.

PROCESS AFTER INPUT.
  MODULE CANCEL AT EXIT-COMMAND.
  MODULE USER_COMMAND_0200.
```

The user can change and save data for the selected flight. (The programming for the database update is not shown in this example.)

After the screen has been processed, the next screen is set to 0, concluding the screen sequence. The system returns to the basic list.

Equally, if you choose *Back* (function code BACK) on either of the two screens, control returns to the basic list.

Printing Lists

Ablegen von Listen mit SAP ArchiveLink

Drucksteuerung [Page 919]

BC - SAP-Druckhandbuch [Ext.]

You use lists to output structured, formatted data. By default, the system sends a list (basic list and secondary lists) to the output screen after creating it. This section describes how to send lists to the SAP spool system instead of the output screen.

Within ABAP, sending a list to the SAP spool system is generally called 'printing lists'. This, however, must not necessarily mean that the list is actually printed on a printer. You can also use the spool system to store a list temporarily, or to store lists in an optical archive instead of printing it.

ABAP offers two possibilities to print a list:

You can print a list **afte**r and **while** creating it.

Printing a List after Creating it [Page 905]

Printing a List while Creating it [Page 907]

Additional Information about this Section

For more comprehensive information about spool administration, refer to the Printing Guide.

For further information about archiving lists, refer to the SAP ArchiveLink and Storing ABAP Lists documentation.

Printing a List after Creating it

Ablegen von Listen mit SAP ArchiveLink [Page 926]

When printing a list after creating it, you do not use any of the print-specific statements described in the topics below to send the list from within the program to the SAP spool system.

By default, the system sends the completed list to the output screen. If the *Print* function (function code PRI) in the status of the user interface for the list has been activated, the user can send the screen list to the SAP spool system by choosing *Print* (see Printing Output Lists). The system requests the print parameters in the *Print Screen List* dialog box (see Print Parameters [Page 908]). For information about how to change the default values on this screen, refer to Print Parameters - Setting Default Values.

Printing a list after creating it has the following problems:

- The list is formatted for screen display, and not for printing. This is not always suitable for printing, for the following reasons:
  - A list on an output screen usually consists of a single page (see notes in Setting the Page Length [Page 801]). When printing, the system 'cuts' this logical page into several physical pages whose format depends on the print parameters specified. The system places the page header on each of these print pages. If the page header contains a page number, this number is the same on all pages (SY-PAGNO). This means that you cannot number the pages.
  - If the list contains page breaks programmed using NEW-PAGE (see Unconditional Page Breaks), these page breaks are not adapted to the format of the print pages, which may lead to further automatic page breaks. For a print page created by an automatic page break, the system uses the same page header as for the previous page, since only NEW-PAGE increases the system field SY-PAGNO.
  - If the list consists of several pages due to the LINE-COUNT option in the REPORT or NEW-PAGE statement (see Lists with Several Pages [Page 805]), you can either not print the list at all, since the page length specified exceeds the maximum page length of a print page, or you do not make full use of the physical print page.
  - You can set the width of a list on an output screen to any value between 1 and 255 columns (see Setting the List Width [Page 798]). This list width is not adapted to a printer format. For example, a normal printer prints lists wider than 132 columns with the same small letter size as lists of 255 columns because there is no print format in between.
- When creating a list for screen output, you cannot include print control statements into the list (see Print Control [Page 919]).
- You cannot output footer lines defined in the program at the end of each printed page. However, you can instead select *Footer line* in the *Print Screen List* dialog box. The system then reserves one line on each page for the system-defined footer line.
- Screen lists do not contain any index information for optical archiving. You can only create index information for optical archiving while the list is being created (see Indexing Print Lists for Optical Archiving).

The printout of a completed list from the output screen is a hard copy of the screen rather than a real program-controlled printout. Use this method for testing purposes only, or for lists whose formats are acceptable to the printer. You should print complex lists (like lists with extensive page

headers that should not appear on each printed page) from the program (see Printing a List During Creation [Page 907]).

If you want to offer the user the possibility of starting a program-controlled print process from the output screen, use the methods of interactive reporting (see User Actions on Lists [Page 854]). For example, you first create a list for the output screen. Use a user interface of your own in which you replace function code PRI with a different function code. You can then recreate the list for the spool system in the AT USER-COMMAND event (see Printing a List During Creation [Page 907]).

Printing a List while Creating it

When you print a list while creating it, you receive best print output, since the system formats the list according to the requirements of the printer. The system sets list width and page length according to the print format. This prevents lines from being wider than the print format in use. Page breaks occur at the end of a physical print page.

The program must know the print format before it starts creating the list. The print format is part of the print parameters. Print parameters are set either interactively by the user or from within the program.

Print Parameters [Page 908]

You can print a list while creating it in one of the following ways:

- If your program displays a selection screen, the user can choose *Execute + print* on the selection screen.

Execute and Print [Page 909]

- You can start print output from within your program using the NEW-PAGE PRINT ON statement.

Printing from Within a Program [Page 912]

- You can call an executable program using the SUBMIT ... TO SAP-SPOOL statement.

Printing Lists of Called Executable Programs [Page 916]

- You can include a report into a background job using the function module JOB_SUBMIT. For further information about background jobs and the function module JOB_SUBMIT, refer to the Basis Programming Interfaces [Ext.] documentation.

When printing a list while creating it, you can manipulate the print format:

Print Control

When printing a list while creating it, the system sends each completed page to the spool system and then deletes it. The length of a printed list, therefore, is restricted only by the capacity of the spool system. In contrast to lists for display, the system does not store list levels when printing. Since an entire list in printing never exists, you cannot refer to the contents of previous pages.

Print Parameters

Ablegen von Listen mit SAP ArchiveLink

Drucklisten für die optische Archivierung indizieren [Ext.]

You must set print parameters before the printing process starts.

When printing lists **after** creating them, the system uses the print format specified in the print parameters to split the completed list and fit it onto the print pages, truncating it if necessary.

When printing lists **while** creating them, the system uses the print format to actually format the list in the program.

Print parameters are set either interactively by the user or from within the program.

Print Parameters - Overview

Print Parameters - Setting Default Values

Setting Print Parameters in a Program

Execute and Print

The easiest way of printing a list while creating it, is for the user to choose *Execute + print* on the report's selection screen. The user can choose between displaying the list on the screen (choosing *Execute*) or printing it directly without displaying it (choosing *Execute + print*).

If the user chooses *Execute + print* on the selection screen of the report, the system displays the *Print List Output* dialog window before creating the list. The user enters the print parameters. The function module SET_PRINT_PARAMETERS allows you to set default values for the dialog box. For further information, refer to Print Parameters - Setting Default Values.

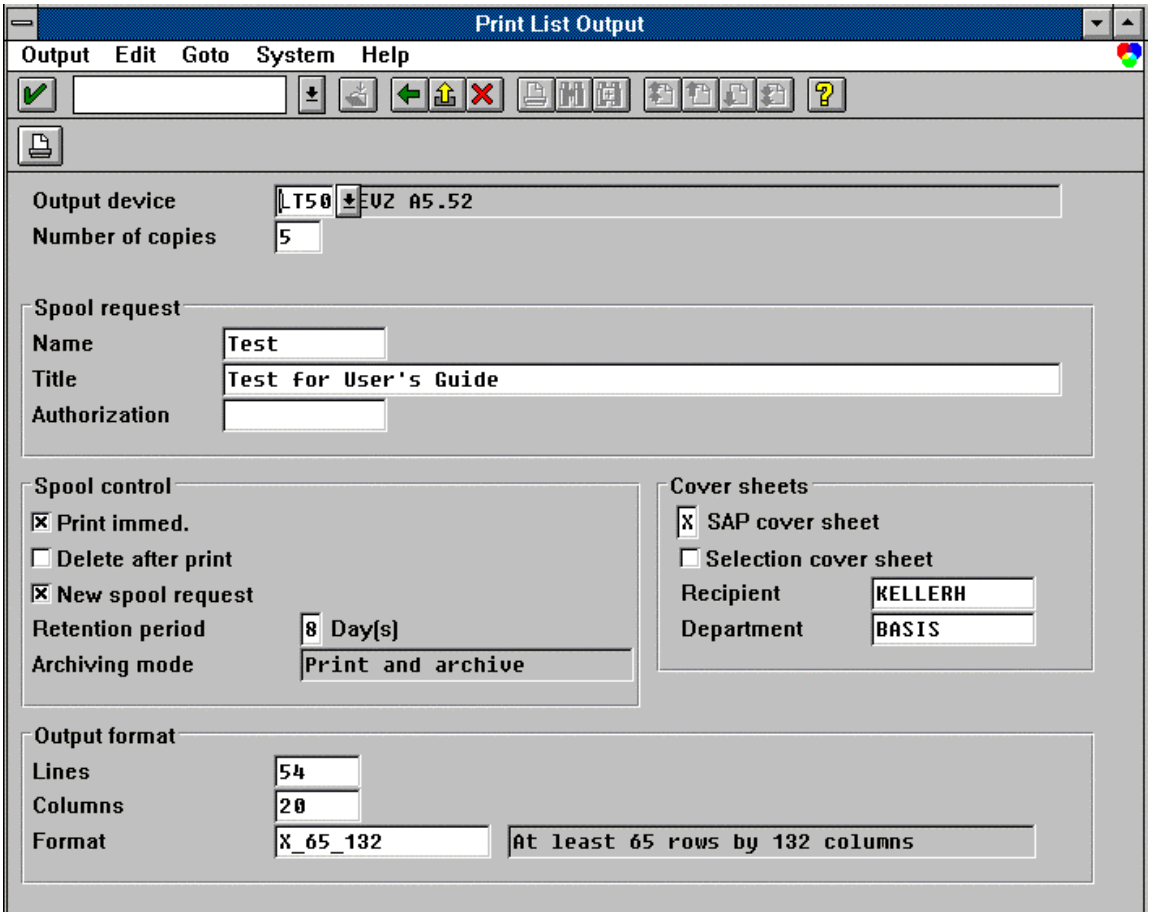
Consequently, you must program the list in such a way that it can both be displayed and printed. Therefore, in the REPORT statement, do not specify the page width wider than 132 characters (LINE-SIZE option) and do not set the page length (LINE-COUNT option) at all.

Using *Execute + print,* the user can print only the basic list of the report. To print secondary lists that you create during interactive events on the displayed list, use NEW-PAGE PRINT ON (see Printing from Within the Program [Page 912]).

```
REPORT SAPMZTST NO STANDARD PAGE HEADING LINE-COUNT 0(2).

PARAMETERS P TYPE I.

INITIALIZATION.

  CALL FUNCTION 'SET_PRINT_PARAMETERS'
     EXPORTING
        ARCHIVE_MODE   = '3'
        COPIES         = '5'
        DEPARTMENT     = 'BASIS'
        DESTINATION    = 'LT50'
        EXPIRATION     = ' '
        IMMEDIATELY    = 'X'
        LAYOUT         = 'X_65_132'
        LINE_COUNT     = 54
        LINE_SIZE      = 20
        LIST_NAME      = 'Test'
        LIST_TEXT      = 'Test for User"s Guide'
        NEW_LIST_ID    = 'X'
        RECEIVER       = 'KELLERH'
        RELEASE        = ' '
        SAP_COVER_PAGE = 'X'.

START-OF-SELECTION.

  DO P TIMES.
    WRITE / SY-INDEX.
  ENDDO.

TOP-OF-PAGE.

  WRITE: 'Page', SY-PAGNO.
  ULINE.
```

```
END-OF-PAGE.
  ULINE.
  WRITE: 'End of', SY-PAGNO.
```

After executing this program, the user can enter a value for parameter P on the selection screen (for example 100) and choose *Execute + print.* The following dialog box appears:



The function module SET_PRINT_PARAMETERS fills the input fields with the default values. Because of the function module, the *Lines* field is ready for input, even though the LINE-COUNT addition is used in the REPORT statement. The option, in this case, is needed to reserve space for two footer lines.

After choosing *Print* on the *Print List Output* dialog window, the system displays the *Archive Parameters* dialog box, since the import parameter ARCHIV_MODE sets the archiving mode to *Print and archive*.

If the user enters 100 for the parameter P on the selection screen, the system creates an SAP cover page and two print pages that look like this.

First page:

```
Page   1
--------------------
    1
```

```
    2
    3

  .....

      49
   50
--------------------
End of   1
```

Second page:

```
Page   2
--------------------
   58
   59
   60

  .....

      99
   100
--------------------
End of   2
```

On each page, you can output up to 54 lines, including page header and footer lines. Note that the system triggers page breaks and creates page headers and footers exactly as described in Creating Complex Lists [Page 788].

If the user chooses *Execute* on the selection screen instead of *Execute + print*, the system displays the list as one page and without page footer on the output screen.

Printing from within the Program

To start the printing process from within the program while creating a list, use the NEW-PAGE statement with the PRINT ON option:

Syntax

```
NEW-PAGE PRINT ON [NEW-SECTION]
                  [<params> | PARAMETERS <pripar>]
                  [ARCHIVE PARAMETERS <arcpar>]
                  [NO DIALOG].
```

All output after this statement is placed on a new page (see Unconditional Page Break), and the system interprets all subsequent output statements as print statements. In other words, starting from NEW-PAGE PRINT ON, the system no longer creates the list for display but for the spool system.

If you use the NEW-PAGE PRINT ON statement without the NEW-SECTION option while already creating a list for the spool system, the statement is of no effect.

If you use the NEW-SECTION option, you reset pagination (SY-PAGNO system field) to 1. If the system is already creating a list for the spool system, NEW-SECTION may have two effects:

- If the print parameters specified match the parameters of the current list and the print parameter PRNEW equals SPACE, the system does not create a new spool request.
- If the print parameters specified do not match the parameters of the current list or the print parameter PRNEW is unequal to SPACE, the system closes the current spool request and creates a new spool request.

The other options determine the print parameters (see below).

The end of a processing block (events during data retrieval or interactive events) automatically ends the print process. To stop creating the list for the spool system explicitly, use the PRINT OFF option of the NEW-PAGE statement:

Syntax

```
NEW-PAGE PRINT OFF.
```

This statement creates a page break and sends the last page to the spool system. Any output statements following this statement appear in the list on the output screen.

Determining Print Parameters

To determine the print parameters for printed output following the NEW- PAGE PRINT ON statement, use the options of the statement.

You can use several options <params> to specify each print parameter (for example DESTINATION <dest>). The keyword documentation explains each option. Use the NO DIALOG option to tell the system whether to display or suppress the *Print List Output* dialog box. This method of setting print parameters has its disadvantages, since the system does not check whether the parameters specified are complete. Incomplete print parameters are detected only if you use the *Print List Output* dialog box. However this is not possible for background jobs. If the print parameters are incomplete and you use the NO DIALOG option, the system sends a warning after the syntax check, but it does not stop processing. This may cause unpredictable results when executing the program.

SAP, therefore, recommends not to use the <params> options. Use the PARAMETERS option instead, and the ARCHIVE PARAMETERS option if necessary. To create the corresponding arguments <pripar> and <arcpar>, use the export parameters of the function module GET_PRINT_PARAMETERS (see Setting Print Parameters from the Program). This is the only method that guarantees a complete set of print parameters and an executable print request. Since the function module GET_PRINT_PARAMETERS has its own user dialog, always use the NO DIALOG option in the NEW-PAGE PRINT ON statement.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING.

DATA: VAL,
      PRIPAR LIKE PRI_PARAMS,
      ARCPAR LIKE ARC_PARAMS,
      LAY(16), LINES TYPE I, ROWS TYPE I.

CALL FUNCTION 'GET_PRINT_PARAMETERS'
     IMPORTING
          OUT_PARAMETERS        = PRIPAR
          OUT_ARCHIVE_PARAMETERS = ARCPAR
          VALID                 = VAL
     EXCEPTIONS
          ARCHIVE_INFO_NOT_FOUND = 1
          INVALID_PRINT_PARAMS   = 2
          INVALID_ARCHIVE_PARAMS = 3
          OTHERS                 = 4.

IF VAL <> SPACE AND SY-SUBRC = 0.
  SET PF-STATUS 'PRINT'.
  WRITE '   Select a format!'.
ENDIF.

TOP-OF-PAGE DURING LINE-SELECTION.
  WRITE: 'Page', SY-PAGNO.
  ULINE.

AT USER-COMMAND.
  CASE SY-UCOMM.
    WHEN 'PORT'.
      LAY = 'X_65_80'.
      LINES = 60.
      ROWS  = 55.
      PERFORM FORMAT.
    WHEN 'LAND'.
      LAY = 'X_65_132'.
      LINES = 60.
      ROWS  = 110.
      PERFORM FORMAT.
  ENDCASE.

FORM FORMAT.
  CALL FUNCTION 'GET_PRINT_PARAMETERS'
       EXPORTING
            IN_ARCHIVE_PARAMETERS  = ARCPAR
            IN_PARAMETERS          = PRIPAR
```

```
          LAYOUT              = LAY
          LINE_COUNT          = LINES
          LINE_SIZE           = ROWS
          NO_DIALOG           = 'X'
        IMPORTING
          OUT_ARCHIVE_PARAMETERS = ARCPAR
          OUT_PARAMETERS      = PRIPAR
          VALID               = VAL
        EXCEPTIONS
          ARCHIVE_INFO_NOT_FOUND = 1
          INVALID_PRINT_PARAMS   = 2
          INVALID_ARCHIVE_PARAMS = 3
          OTHERS              = 4.
  IF VAL <> SPACE AND SY-SUBRC = 0.
    PERFORM LIST.
  ENDIF.
ENDFORM.

FORM LIST.
  NEW-PAGE PRINT ON
    NEW-SECTION
    PARAMETERS PRIPAR
    ARCHIVE PARAMETERS ARCPAR
    NO DIALOG.
  DO 440 TIMES.
    WRITE (3) SY-INDEX.
  ENDDO.
ENDFORM.
```

This program immediately calls the function module GET_PRINT_PARAMETERS without passing import parameters. In the *Print List Output* dialog box, the user can enter the print and archiving parameters for this program. The system passes these parameters, using the export parameters of the function module, to the structures PRIPAR and ARCPAR. To guarantee that the parameters are complete and consistent, the program runs the user dialog and checks the return value of VALID.

After completing the dialog, the system displays the following basic list:

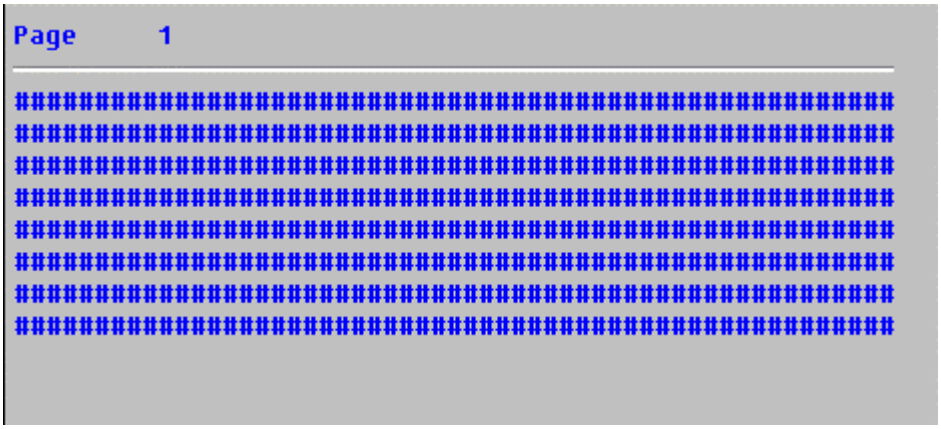
In the status PRINT of the basic list, the function codes PORT and LAND are assigned to the function keys `F5` and `F6`, and to two pushbuttons of the application toolbar (see Defining Individual User Interfaces). If the user chooses one of these functions, the AT USER-COMMAND event occurs, assigning to the variables LAY, LINES, and ROWS the values for portrait or landscape output format and calling the subroutine FORMAT.

The subroutine FORMAT calls the function module GET_PRINT_PARAMETERS. When it does so, it passes the parameters PRIPAR and ARCPAR as import parameters. The values from LAY, LINES, and ROWS are assigned to the import parameters LAYOUT, LINE_COUNT and LINE_SIZE respectively. There is no user dialog. The system returns the parameters to the structures PRIPAR and ARCPAR.

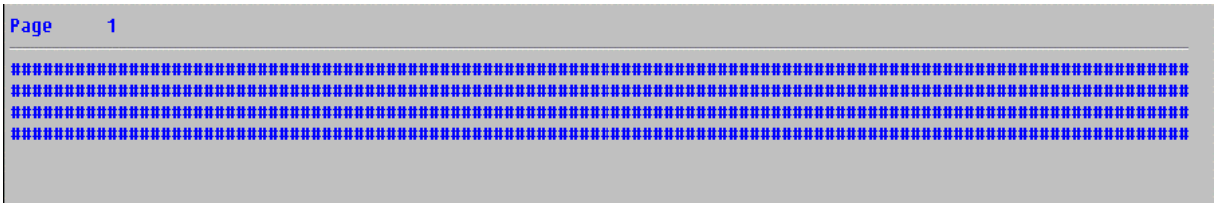
The function of the subroutine call is to set the components PAART, LINCT, and LINSZ of the structure PRIPAR with new values.

After checking the parameters for completeness and consistency, the program calls the subroutine LIST. This subroutine sends a list to the spool system using NEW-PAGE PRINT ON, thereby determining the print and archiving parameters using PRIPAR and ARCPAR. A user dialog is not necessary, since all settings required were made by GET_PRINT_PARAMETERS.

To view the stored spool requests, the user can choose *System → Services → Print requests.* After choosing *PORTRAIT,* the spool request may look like this:



After choosing *LANDSCAPE,* however, the spool request looks like this:



Printing Lists from a Called Program

Druckparameter [Page 908]

To send the output of reports you call from within your program using SUBMIT to the spool system, you must include the TO SAP-SPOOL option into the SUBMIT statement:

Syntax

```
SUBMIT <rep> TO SAP-SPOOL
       [<params>|SPOOL PARAMETERS <pripar>]
       [ARCHIVE PARAMETERS <arcpar>]
       [WITHOUT SPOOL DYNPRO].
```

The SUBMIT statement is described in detail in the section Calling Executable Programs [Page 1018]. If you use the TO SAP-SPOOL option, the list is formatted for printing while it is being created, and then sent to the spool system. The remaining additions set the print parameters.

Setting the Print Parameters

The same applies to setting the print parameters as described for the NEW-PAGE PRINT ON statement (see Printing from within the Program).

Although you can set each print parameter individually using one of the <params> additions (refer to the keyword documentation), or by using a user dialog in the SUBMIT statement, you should only retrieve the parameters using the function module GET_PRINT_PARAMETERS. (For further information, refer to Setting Print Parameters from within the Program). The function module GET_PRINT_PARAMETERS decouples the user dialog from the SUBMIT statement, and guarantees a complete parameter set even without executing the user dialog. To determine the parameters, use only the options SPOOL PARAMETERS and ARCHIVE PARAMETERS and, to suppress the user dialog of the SUBMIT statement, use the WITHOUT SPOOL DYNPRO option.

The following executable program is connected to the logical database F1S:

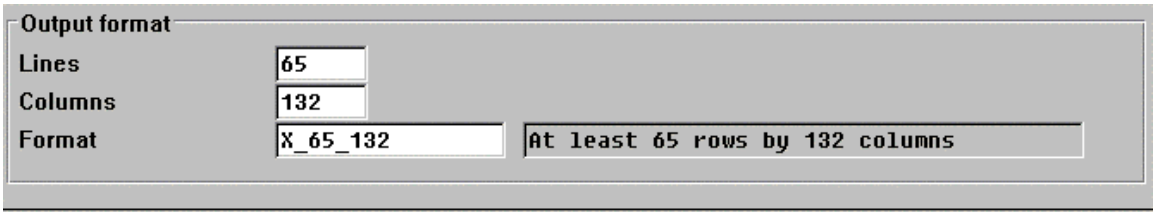
```
REPORT SAPMZTS1.

TABLES SPFLI.

GET SPFLI.

  NEW-LINE.
  WRITE: SPFLI-MANDT, SPFLI-CARRID, SPFLI-CONNID,
  SPFLI-CITYFROM, SPFLI-AIRPFROM, SPFLI-CITYTO,
  SPFLI-AIRPTO, SPFLI-FLTIME, SPFLI-DEPTIME, SPFLI-ARRTIME,
  SPFLI-DISTANCE, SPFLI-DISTID, SPFLI-FLTYPE.
```

The following program calls SAPMZTS1 and sends the output to the spool system:

```
REPORT SAPMZTST NO STANDARD PAGE HEADING.

DATA: VAL,
      PRIPAR LIKE PRI_PARAMS,
      ARCPAR LIKE ARC_PARAMS.
```

```
CALL FUNCTION 'GET_PRINT_PARAMETERS'
     EXPORTING
        LAYOUT              = 'X_65_132'
        LINE_COUNT            = 65
        LINE_SIZE           = 132
    IMPORTING
        OUT_PARAMETERS        = PRIPAR
        OUT_ARCHIVE_PARAMETERS = ARCPAR
        VALID               = VAL
    EXCEPTIONS
        ARCHIVE_INFO_NOT_FOUND = 1
        INVALID_PRINT_PARAMS   = 2
        INVALID_ARCHIVE_PARAMS = 3
        OTHERS              = 4.

IF VAL <> SPACE AND SY-SUBRC = 0.
  SUBMIT SAPMZTS1 TO SAP-SPOOL
    SPOOL   PARAMETERS PRIPAR
    ARCHIVE PARAMETERS ARCPAR
    WITHOUT SPOOL DYNPRO.
ENDIF.
```

After starting the program, the function module GET_PRINT_PARAMETERS triggers a user dialog, displaying the area *Output format* of the *Print List Output* dialog window filled with the values from the import parameters:



After the user has entered and confirmed the print parameters, SAPMZTS1 is called. In the call, the export parameters of GET_PRINT_PARAMETERS are passed as print and archiving parameters. SAPMZTS1 creates neither a screen display nor a user dialog. It sends the created list directly to the spool system. The user can view the stored spool request choosing *System → Services → Print requests.* Using the output format specified above, the spool request may look like this:

```
002 AA  0017 NEW YORK           JFK SAN FRANCISCO      SFO 06:01:00 13:30:00 16:31:00 2.572  MLS
002 AA  0064 SAN FRANCISCO      SFO NEW YORK           JFK 05:21:00 09:00:00 17:21:00 2.572  MLS
002 DL  1699 NEW YORK           JFK SAN FRANCISCO      SFO 06:22:00 17:15:00 20:37:00 2.572  MLS
002 DL  1984 SAN FRANCISCO      SFO NEW YORK           JFK 05:25:00 10:00:00 18:25:00 2.572  MLS
002 LH  0400 FRANKFURT          FRA NEW YORK           JFK 08:24:00 10:10:00 11:34:00 6.162  KM
002 LH  0402 FRANKFURT          FRA NEW YORK           JFK 01:35:00 13:30:00 15:05:00 6.162  KM
002 LH  0454 FRANKFURT          FRA SAN FRANCISCO      SFO 12:20:00 10:10:00 12:30:00 9.096  KM
002 LH  0455 SAN FRANCISCO      SFO FRANKFURT          FRA 13:30:00 15:00:00 10:30:00 9.096  KM
002 LH  2402 FRANKFURT          FRA BERLIN             SXF 01:05:00 10:30:00 11:35:00   430  KM
002 LH  2407 BERLIN             TXL FRANKFURT          FRA 01:05:00 07:10:00 08:15:00   430  KM
002 LH  2415 BERLIN             SXF FRANKFURT          FRA 01:05:00 09:25:00 10:30:00   430  KM
002 LH  2436 FRANKFURT          FRA BERLIN             THF 01:05:00 17:30:00 18:35:00   430  KM
002 LH  2462 FRANKFURT          FRA BERLIN             TXL 01:05:00 06:30:00 07:35:00   430  KM
002 LH  2463 BERLIN             SXF FRANKFURT          FRA 01:05:00 21:25:00 22:30:00   430  KM
002 LH  3577 ROM                FCO FRANKFURT          FRA 02:00:00 07:05:00 09:05:00   957  KM
002 SQ  0026 FRANKFURT          FRA NEW YORK           JFK 08:20:00 08:30:00 09:50:00 3.851  MLS
002 UA  0007 NEW YORK           JFK SAN FRANCISCO      SFO 06:10:00 14:45:00 17:55:00 2.572  MLS
002 UA  0941 FRANKFURT          FRA SAN FRANCISCO      SFO 12:36:00 14:30:00 21:06:00 5.685  MLS
002 UA  3504 SAN FRANCISCO      SFO FRANKFURT          FRA 13:35:00 15:00:00 10:30:00 5.685  MLS
```

Print Control

You can manipulate the output of a list during the print process from within the executable program (report). The statements SET MARGIN and PRINT-CONTROL, described in the topics below, take effect only if the list is directly sent to the spool. The statements do not affect a list that is displayed on the screen and printed from there using *List → Print*.

Setting the Left and Upper Margin [Page 920]

Setting the Print Format [Page 922]

Indexing Print Lists for Optical Archiving

Determining Left and Upper Margins

To determine the size of the left and of the upper margin of a print page, use this statement:

Syntax

SET MARGIN <x> [<y>].

This statement causes the current print page to be sent to the spool system with <x> columns of space on the left page margin and, if specified, with <y> lines of space on the upper page margin.

The statement sets the contents of the system fields SY-MACOL and SY-MAROW to <x> and <y>. For printouts, these system fields determine the number of columns on the left margin and the number of lines on the upper margin.

The values you set apply to all subsequent pages, until you use another SET MARGIN statement. If you specify more than one SET MARGIN statement on one page, the system always uses the last one.

The following executable program (report) is connected to the logical database F1S.

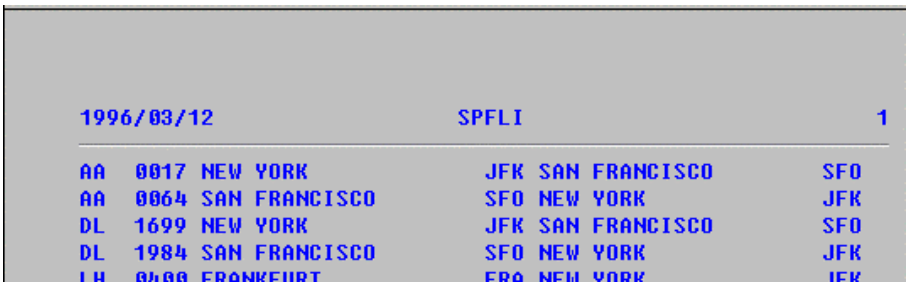
```
REPORT SAPMZTST LINE-SIZE 60.

TABLES SPFLI.

SET MARGIN 5 3.

GET SPFLI.
  WRITE: / SPFLI-CARRID,  SPFLI-CONNID, SPFLI-CITYFROM,
     SPFLI-AIRPFROM, SPFLI-CITYTO, SPFLI-AIRPTO.
```

If, after starting the report, the user chooses *Execute* on the selection screen, the following list appears on the output screen:

```
SPFLI                                                        1

AA  0017 NEW YORK            JFK SAN FRANCISCO        SFO
AA  0064 SAN FRANCISCO       SFO NEW YORK             JFK
DL  1699 NEW YORK            JFK SAN FRANCISCO        SFO
DL  1984 SAN FRANCISCO       SFO NEW YORK             JFK
LH  0400 FRANKFURT           FRA NEW YORK             JFK
```

The SET MARGIN statement has no effect on the display.

If, after starting the report, the user chooses *Execute + print* on the selection screen, the list is printed while it is created. The user can view the stored spool request using *System → Services → Print requests*:



The list is shifted to the right by five columns and to the bottom by three lines.

Determining the Print Format

BC - SAP-Druckhandbuch [Ext.]

To determine the print format, use the PRINT-CONTROL statement:

Syntax

PRINT-CONTROL <formats> [LINE <lin>] [POSITION <col>].

Without using the options LINE and POSITION, this statement sets the print format specified in <formats> for all characters that are printed starting from the current output position (system fields SY-COLNO and SY-LINNO). The LINE option sets the print format to start from line <lin>. The POSITION option sets the print format to start from column <pos>.

In <formats>, you can specify several different print formats. The system converts the values into a printer-independent code, the so-called print-control code. When printing, the system translates the print-control code to printer-specific control characters of the selected printer.

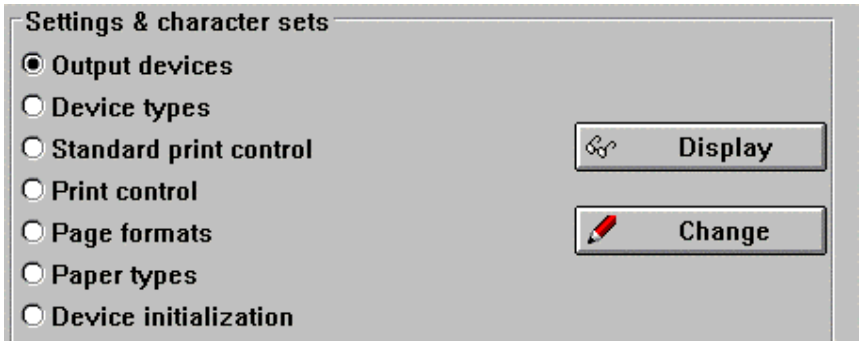
The table below lists the valid <formats> options and the corresponding print-control codes:

<formats>	Code	Meaning
CPI <cpi>	CI<cpi>	Characters per inch
LPI <lpi>	LI<lpi>	Lines per inch
COLOR BLACK	CO001	Color black
COLOR RED	CO002	Color red
COLOR BLUE	CO003	Color blue
COLOR GREEN	CO004	Color green
COLOR YELLOW	CO005	Color yellow
COLOR PINK	CO006	Color pink
LEFT MARGIN <lfm>	LM<lfm>	Space from the left margin
FONT <fnt>	FO<fnt>	Font
FUNCTION <code>	<code>	For directly specifying a code

There are many more print-control codes than <formats> options. Therefore, you can specify any print-control code directly using the FUNCTION option.

Use the print formats only to set formats that are either not possible or not reasonable when formatting output for the output screen (for example, size specifications or fonts). You should set all other formats as described in Formatting Options [Page 780] or Formatting the Output [Page 832]. These formats automatically apply for printing as well as for display (provided the specified printer supports them).

To find the codes supported by a certain printer, choose *Tools* → *Administration* → *Spool* → *Spool administration*. This takes you to the screen *Spool Administration* (transaction SPAD).

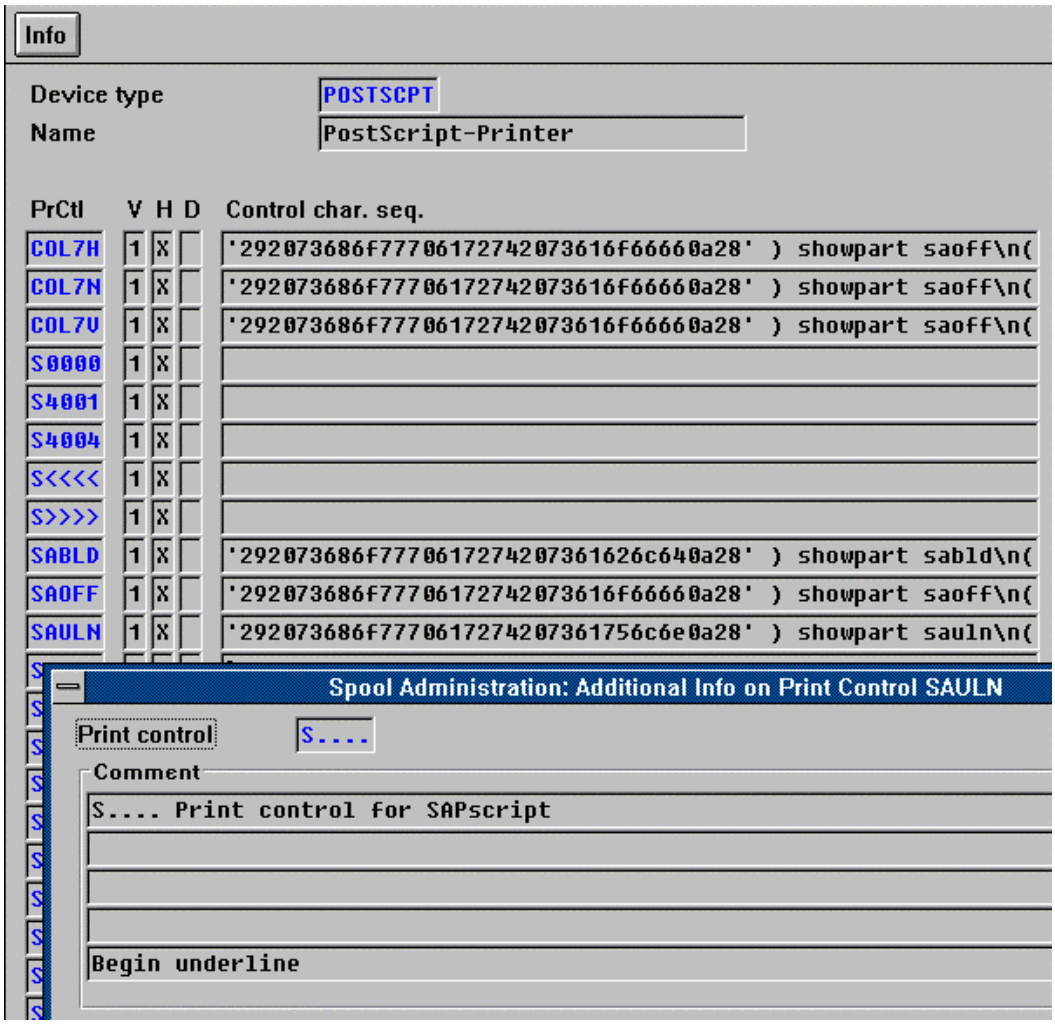


Choose the individual components to display the following information:

Choose	Information
Output devices	a list of the installed printers, including the device types
Device types	a description of the device types
Standard print control	a list of the print-control codes
Print control	an assignment of printer-specific control characters to the print-control codes for each device type
Page formats	a list of page formats
Paper types	a list of valid formats
Device initialization	an assignment of formats for each device type

Display *Print control* to find the print-control codes available for the printer you want to use. From the displayed list, select the desired device type. A section from this table (T02DD) for a Post Script printer may look like this:

Determining Left and Upper Margins



On the right, you can see whether printer-specific control characters are maintained for a print-control code. Choose *Info* to display additional information for the individual codes in a dialog window.

For more comprehensive information about spool administration, refer to the Printing Guide.

The following executable program (report) is connected to the logical database F1S.

```
REPORT SAPMZTST LINE-SIZE 60.

TABLES SPFLI.

PRINT-CONTROL FUNCTION: 'SABLD' LINE 1,
                        'SAOFF' LINE 2,
                        'SAULN' LINE 3.

GET SPFLI.
  WRITE: / SPFLI-CARRID,   SPFLI-CONNID,  SPFLI-CITYFROM,
           SPFLI-AIRPFROM, SPFLI-CITYTO,  SPFLI-AIRPTO.
```

If in table T02DD, the printer control characters for the print-control codes SABLD, SAOFF, and SAULN are defined as in the figure above, the system formats the output as follows:

```
1996/03/13              SPFLI                    1
------------------------------------------------------------
AA  0017 NEW YORK          JFK SAN FRANCISCO      SFO
AA  0064 SAN FRANCISCO     SFO NEW YORK           JFK
DL  1699 NEW YORK          JFK SAN FRANCISCO      SFO
DL  1984 SAN FRANCISCO     SFO NEW YORK           JFK
LH  0400 FRANKFURT         FRA NEW YORK          JFK

....
```

The print format for the first line is set to bold, using the print-control code SABLD. The print-control code SAOFF turns off bold style, starting from the second line. Using the print-control code SAULN, the system underlines all lines starting from line 3.

Documentation Not Available in Release 4.6C

Messages

In addition to the three kinds of user dialogs, you can also use message to communicate with users. They are mostly used for error handling during processing of other user dialogs.

Message Administration [Page 928]

Sending Messages [Page 929]

Processing Messages [Page 931]

Message Management

Nachrichtenklassen definieren [Ext.]

Messages are single texts, stored in table **T100**, that you can maintain in Transaction SE91 or by forward navigation in the ABAP Workbench. T100 has the following structure:

- Language key
- Twenty-character message class
- Message number
- Message text (up to 72 characters)

Message classes assign messages to an application (maybe to a development class, for example), and the message numbers identify the individual messages within the message class. When you call a message in ABAP, you need to specify the message class and message number. The language key is supplied by the system according to the system language environment.

For information about creating message classes and messages, refer to the Creating Messages section of the ABAP Workbench documentation

Messages

You send messages using the ABAP statement MESSAGE. The statement specifies the message class, number, and type of the message.

The message class and number are used to identify the message in table T100. The message type is one of A, E, I, S, W, or X, and defines how the ABAP runtime should process the message [Page 931].

Messages can either be displayed in modal dialog boxes, or in the status bar of the screen. How a message is processed [Page 931] depends on its type and on the context in which the MESSAGE statement occurs.

The MESSAGE Statement

The MESSAGE statement has three variants and several additions:

Using a Global Message Class

If the introductory statement [Page 1365] of a program contains the addition

... MESSAGE-ID <id>.

and a message class <id> contained in table T100, you can use the MESSAGE statement as follows:

MESSAGE <t><num> [WITH $<f_1>$ ... $<f_4>$] [RAISING <exc>].

where <t> is the single-character message type and <nnn> the three-digit message number. The system retrieves the corresponding message text from table T100 for the message class specified in the introductory statement, and displays it. The display depends on the message type and the context in which the message is sent.

Specifying the Message Statically

To specify the message class, message number, and message type statically, use the following form of the MESSAGE statement:

MESSAGE <t><nnn>(<id>) [WITH <f1> ... <f4>] [RAISING <exc>].

This statement is like the first variant, but here you specify the message class <id> within the statement. This specification overrides any message class that you may have specified at the beginning of the program.

Specifying the Message Dynamically

To specify the message class, message number, and message type dynamically, use the following form of the MESSAGE statement:

MESSAGE ID <id> TYPE <t> NUMBER <n> [WITH <f1> ... <f4>] [RAISING <exc>].

where <id>, <t>, and <n> are fields containing the message class, message number, and message type respectively. The system uses the field contents to read the appropriate message from table T100 and displays it according to the message context.

Filling Message Texts Dynamically

Message texts in table T100 can contain up to four ampersand characters as placeholders. You can replace these at runtime using the WITH addition in the MESSAGE statement:

MESSAGE ... WITH <f_1> ... <f_4>.

The contents of fields <f1> ... <f4> are then inserted sequentially into the message text in place of the placeholders.

Messages and Exceptions

Within function modules and methods, you can use the RAISING addition in the MESSAGE statement to trigger exception:

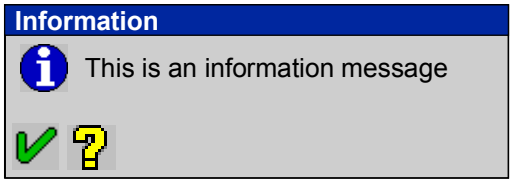
MESSAGE..... RAISING <exc>.

If the calling program does not handle the exception <exc> itself, the message is displayed, and the program continues processing in the manner appropriate to the message type and context. If the calling program handles the exception, the message is not displayed, but the exception is triggered. In this case, the message class, message number, message type, and any values of placeholders are placed in the system fields SY-MSGID, SY-MSGNO, SY-MSGTY, and SY-MSGV1 to SY-MSGV4 in the calling program.

Example

```
REPORT DEMO_MESSAGES_SIMPLE MESSAGE-ID SABAPDOCU.

MESSAGE I014.

MESSAGE S015.

WRITE text-001.
```

This simple message displays an information message in a modal dialog box:



and a success message in the status bar of the next screen (in this case, a list).

Message Processing

Nachrichten auf Listen [Page 935]

Message processing depends on the message type specified in the MESSAGE statement, and the program context in which the statement occurs.

Message Types

A	**Termination**	The message appears in a dialog box, and the program terminates. When the user has confirmed the message, control returns to the next-highest area menu.
E	**Error**	**Depending on the program context**, an error dialog appears or the program terminates.
I	**Information**	The message appears in a dialog box. Once the user has confirmed the message, the program continues immediately after the MESSAGE statement.
S	**Status**	The program continues normally after the MESSAGE statement, and the message is displayed in the status bar of the next screen.
W	**Warning**	**Depending on the program context**, an error dialog appears or the program terminates.
X	**Exit**	No message is displayed, and the program terminates with a short dump. Program terminations with a short dump normally only occur when a runtime error occurs. Message type X allows you to force a program termination. The short dump contains the message ID.

Contexts

Messages, especially those with type **E** or **W**, are processed according to the context in which they occur. The following sections summarize the most important context rules:

Messages Without Screens [Page 932]

Messages on Screens [Page 933]

Messages on Selection Screens [Page 934]

Messages on Lists

Messages in Function Modules and Methods [Page 936]

The program DEMO_MESSAGES demonstrates how the different message types are processed in different contexts.

Messages Without Screens

This context applies to all situations that do not belong to any screen processing. In ABAP programs, this includes the following processing blocks:

- PBO modules (PBO of screens)
- The selection screen event AT SELECTION-SCREEN OUTPUT (PBO of a selection screen)
- The reporting events INITIALIZATION, START-OF-SELECTION, GET, and END-OF-SELECTION
- The list events TOP-OF-PAGE and END-OF-PAGE

All other processing blocks are associated with screen processing (reacting to user input).

Message Processing

Type	Display	Processing
A	Dialog box	Program terminates, and control returns to last area menu
E	In PBO context, the same as type A, otherwise status bar	In PBO context like type A, otherwise, program terminates and control returns to point from which the program was called
I	In PBO context, the same as type S, otherwise dialog box	Program continues processing after the MESSAGE statement
S	Status bar of next screen	Program continues processing after the MESSAGE statement
W	In PBO context, the same as type S, otherwise status bar	In PBO context like type S, otherwise, program terminates and control returns to point from which the program was called
X	None	Triggers a runtime error with short dump

Messages on Screens

This context includes all situations where a screen is being processed, that is, the program is reacting to user input. In ABAP programs, this means all PAI modules.

Message Processing

Type	Display	Processing
A	Dialog box	Program terminates, and control returns to last area menu
E	Status bar	PAI processing is terminated, and control returns to the current screen. All of the screen fields for which there is a FIELD or CHAIN statement are ready for input. The user **must** enter a new value. The system then restarts PAI processing for the screen using the new values. Error messages are not possible in POH or POV processing. Instead, a runtime error occurs.
I	Dialog box	Program continues processing after the MESSAGE statement
S	Status bar of next screen	Program continues processing after the MESSAGE statement
W	Status bar	Like type E, but the user can confirm the message by pressing ENTER without having to enter new values. The system then resumes PAI processing directly after the MESSAGE statement. Warning messages are not possible in POH or POV processing. Instead, a runtime error occurs.
X	None	Triggers a runtime error with short dump

For further information about processing error messages and warnings in the PAI event, refer to Input Checks in Dialog Modules [Page 584].

Messages on Selection Screens

This context includes all situations where a selection screen is being processed, that is, the program is reacting to user input. In ABAP programs, this corresponds to the AT SELECTION-SCREEN processing blocks, except those with the OUTPUT addition.

Message Processing

Type	Display	Processing
A	Dialog box	Program terminates, and control returns to last area menu
E	Status bar	Selection screen processing terminates, and the selection screen is redisplayed. The screen fields specified through the additions to the AT SELECTION-SCREEN statement are ready for input. The user **must** enter a new value. The system then restarts the selection screen processing using the new values. You cannot use error messages with the ON HELP-REQUEST or ON VALUE-REQUEST additions. Instead, a runtime error occurs.
I	Dialog box	Program continues processing after the MESSAGE statement
S	Status bar of next screen	Program continues processing after the MESSAGE statement
W	Status bar	Like type E, but the user can confirm the message by pressing ENTER without having to enter new values. The system then resumes selection screen processing directly after the MESSAGE statement. You cannot use warning messages with the ON HELP-REQUEST or ON VALUE-REQUEST additions. Instead, a runtime error occurs.
X	None	Triggers a runtime error with short dump

For further information about processing error messages and warnings, refer to the Selection Screen Processing [Page 739] section.

Messages in Lists

This context includes all situations where a list is being processed, that is, the program is reacting to user interaction with lists. In ABAP programs, this includes the following processing blocks:

- AT LINE-SELECTION
- AT USER-COMMAND
- AT PF<nn>
- TOP-OF-PAGE DURING LINE-SELECTION

Message Processing

Type	Display	Processing
A	Dialog box	Program terminates, and control returns to last area menu
E	Status bar	Processing block terminates. Previous list levels remain displayed.
I	Dialog box	Program continues processing after the MESSAGE statement
S	Status bar of next screen	Program continues processing after the MESSAGE statement
W	Status bar	Like type E
X	None	Triggers a runtime error with short dump

For information about processing messages in lists, refer to User Actions and Detail Lists [Page 854].

Messages in Function Modules and Methods

Messages have two different functions in function modules and methods:

Normal Messages

If you use messages in function modules and methods without the RAISING addition in the MESSAGE statement, and the caller does not catch the message, the message is handled normally according to the context in which it is called within the function module or method.

Triggering Exceptions with Messages

If you use messages in function modules and methods with the addition

... RAISING <exc>

the way in which the message is handled depends on whether the calling program handles the exception <exc> or not.

- If the calling program does not handle the exception, the message is displayed and handled according to the context in which it occurs in the function module or method from which it was called.
- If the calling program handles the exception, the message is not displayed. Instead, the procedure is interrupted in accordance with the message type, and processing returns to the calling program. The contents of the message are placed in the system fields SY-MSGID, SY-MSGTY, SY-MSGNO, and SY-MSGV1 to SY-MSGV4.

Catching Message in the Calling Program

You can catch messages from function modules that are not sent using the RAISING addition in the MESSAGE statement by including the implicit exception ERROR_MESSAGE in the EXCEPTIONS list of the CALL FUNCTION statement. The following conditions apply:

- Type S, I, and W messages are ignored (but logged during background processing)
- Type E and A messages trigger the exception ERROR_MESSAGE
- Type X messages trigger the usual runtime error and short dump.

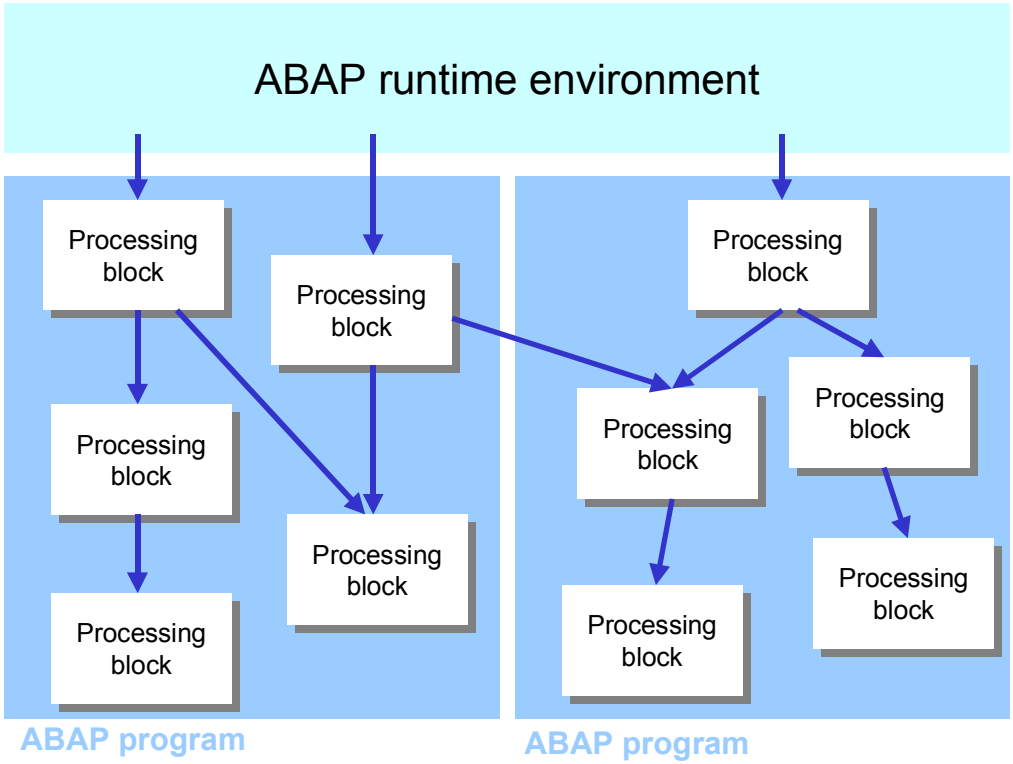
Catching messages is not currently supported for methods.

Running ABAP Programs

Quelltext-Modulen [Page 443]

ABAP programs can only be executed in an R/3 System. The R/3 Basis system contains a component called Kernel & Basis Services that provides a cross-platform runtime environment for ABAP programs (see also The Basis System [Page 21]).

Each ABAP program consists of self-contained **processing blocks**, which may occur in any order in the source code. Processing blocks are sections of programs, made up of structure blocks. They are processed sequentially. A processing block can be called either from outside the ABAP program, or from a processing block within the same ABAP program (see also Structure of ABAP Programs [Page 44]). When a processing block is called from outside the program, it can be called either by the ABAP runtime environment or another ABAP program.



To start an ABAP program, at least one of its processing blocks must be started from outside the program itself. Each ABAP program has a program type, which determines whether and how a program can be run.

Programs that can be run

These programs can be run by a user by entering the program name or a transaction code. They are they actual application programs within the R/3 System, and are explained in more depth later on.

Type 1

You can start a type 1 program by entering its program name. For this reason, they are also known as **executable programs**. When you start a type 1 program, processors are started in the runtime environment that call a series of processing blocks (event blocks) in a predefined sequence. The sequence is oriented towards reporting tasks. This is why executable programs are also known as **reports**. For further information, refer to Running Programs Directly - Reports [Page 945].

A special type of type 1 program is a **logical database**. You can start logical databases **together with reports**. A logical database contains subroutines that are called by an invisible system program in a prescribed sequence for type 1 programs. The subroutines make certain reporting functions reusable. For further information, refer to Logical Databases [Page 1163].

Type M

Type M programs can only be started using a transaction code. A transaction code starts a screen, which consists of the screen itself and its flow logic. Screen flow logic can call special processing blocks (dialog modules) in the corresponding ABAP program. Since type M programs contain mostly dialog modules, they are known as **module pools**. For further information about running module pools, refer to Dialog-controlled Program Execution - Transactions [Page 982].

Programs that Cannot Be Run Directly

These programs cannot be started directly by a user. Instead, they contain processing blocks or other source code that can only be used by an application program that is already running. They are described in more detail in a later section.

Type F

Type F programs are not executable. They serve as a container for function modules. When you call a function module from an ABAP program, the entire main program is loaded into the internal session of the current program. Since type M programs contain mainly function modules, they are known as **function groups**. Function modules [Page 480] are a type of procedure. They are described in more detail in the Procedures [Page 449] section.

Type K

Type K programs are not executable. They are container programs for global classes, and are known as **class definitions**. Classes [Page 1300] belong to ABAP Objects [Page 1291] and are described in more detail in that section.

Type J

Type J programs are not executable. They are container programs for global interfaces, and are known as **interface definitions**. Interfaces [Page 1337] belong to ABAP Objects [Page 1291] and are described in more detail in that section.

Type S

Type S programs are not executable. They are container programs for subroutines that should only be called externally. When you call a subroutine from an ABAP program, the entire main program is loaded into the internal session of the current program. Since type S programs contain mainly subroutines, they are known as **subroutine pools**. Subroutines [Page 451] are a type of procedure. They are described in more detail in the Procedures [Page 449] section.

Type I

Type I programs cannot be run directly, and contain no callable processing blocks. They are used exclusively for modularizing ABAP source code, and are included in other programs. Include programs [Page 447] are described in more detail in the Source Code Modules section.

Starting Programs in ABAP

For all programs that can be started directly, there are ABAP statements that you can use to call a program from another application program that is already running. For further information, refer to Program Calls [Page 1016].

Defining Processing Blocks

This section describes how to define the processing blocks - event blocks and dialog modules - that are called by the ABAP runtime environment.

The processing blocks that you can call using ABAP statements are called procedures. To find out more about them, refer to Modularization Techniques [Page 441].

Event Blocks [Page 941]

Dialog Modules [Page 944]

Event blocks

Event blocks are introduced by an **event keyword**. They end when the next processing block begins. The following processing block may be another event block, introduced by a different event keyword, or another processing block that is valid in the context, such as a subroutine or dialog module. Event keywords have the same name as the events to which they react.

Example for the structure of an executable program:

```
REPORT...
NODES: SPFLI, SFLIGHT.
DATA:...
INITIALIZATION.
...
AT SELECTION-SCREEN.
...
START-OF-SELECTION.
...
GET SPFLI...
..
GET SFLIGHT...
...
GET SPFLI LATE.
...
END-OF-SELECTION.
...
FORM...
...
ENDFORM.
```

The sequence in which the processing blocks occur in the program is irrelevant. The actual processing sequence is determined by the external events. However, to make your programs easier to understand, you should include the event blocks in your program in approximately the same order in which they will be called by the system. Subroutines should be placed at the end of the program.

With only two exceptions (AT SELECTION-SCREEN and GET), event blocks have **no local data area**. All declarative statements in event blocks are handled with the global data declarations in the program. You should therefore include all of your declarations at the start of the program (see also Structure of ABAP Programs [Page 44]).

Statements that are not assigned to a processing block are never executed. An exception to this are any non-declarative statements between the REPORT or PROGRAM statement and the first processing block, which are assigned to the default event START-OF-SELECTION if a program does not contain an explicit START-OF-SELECTION block, these statements form the entire START-OF-SELECTION block. If a START-OF-SELECTION keyword is already included in your program, these statements are inserted at the

beginning of this block. If the program does not contain any explicit event blocks, all non-declarative statements are assigned to the default processing block START-OF-SELECTION.

```
REPORT EVENT_TEST.

WRITE / 'Statement 1'.

FORM ROUTINE.
  WRITE / 'Subroutine'.
ENDFORM.

WRITE / 'Statement 2'.
PERFORM ROUTINE.
WRITE / 'Statement 3'.
```

This produces the following output:

Statement 1

Only the event block START-OF-SELECTION is started in this program. This block consists of the first WRITE statement.

Now we insert a START-OF-SELECTION statement in the program:

```
REPORT EVENT_TEST.

WRITE / 'Statement 1'.

FORM ROUTINE.
  WRITE / 'Subroutine'.
ENDFORM.

START-OF-SELECTION.
  WRITE / 'Statement 2'.
  PERFORM ROUTINE.
  WRITE / 'Statement 3'.
```

The output is now as follows:

```
Statement 1

Statement 2

Subroutine

Statement 3
```

In this program, the START-OF-SELECTION processing block consists of all statements except the FORM-ENDFORM block. A more readable form of the same program would look like this:

```
REPORT SAPMZTST.

START-OF-SELECTION.
  WRITE / 'Statement 1'.
  WRITE / 'Statement 2'.
  PERFORM ROUTINE.
  WRITE / 'Statement 3'.
```

```
FORM ROUTINE.
  WRITE / 'Subroutine'.
ENDFORM.
```

In this case, you could also omit the event keyword START-OF-SELECTION.

Dialog modules

Dialog modules are defined using the following pair of statements:

```
MODULE <mod> OUTPUT|[INPUT].
 ...
ENDMODULE
```

The OUTPUT addition defines the dialog module <mod> as one that may be called from the PBO event of the flow logic of any screen.

The INPUT addition defines the dialog module <mod> as one that may be called from the PAI event of the flow logic of any screen. INPUT is the standard addition. It may be omitted.

You can define two dialog modules, both called <mod>, in the same program as long as one of them has the addition OUTPUT and the other has the addition INPUT.

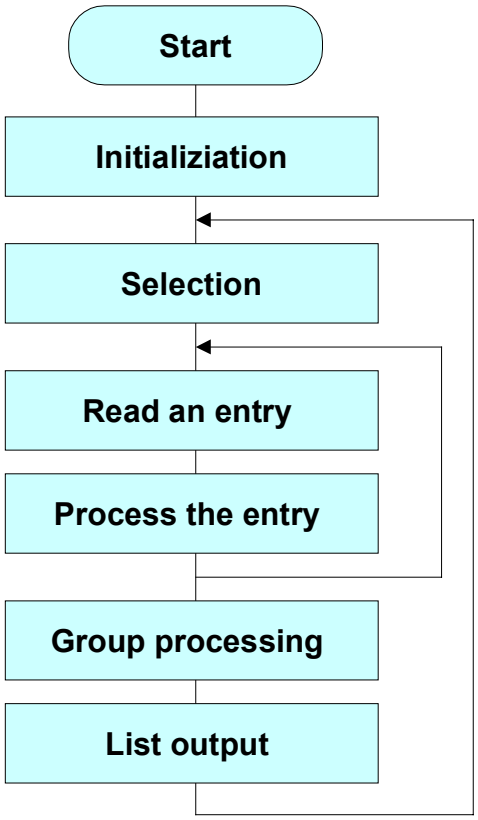
Dialog modules **do not have local data areas**. All declarative statements in dialog modules are handled with the global data declarations in the program. You should therefore include all of your declarations at the start of the program (see also Structure of ABAP Programs [Page 44]).

Running Programs Directly - Reports

Executable programs are programs with program type 1.

Users can run executable programs either in the foreground, by entering the program name in Transaction SA38 (*System* → *Services* → *Reporting*), or as a **background job**. You can only run a program in the background if there is no dialog-controlled execution [Page 982]. You do not need to assign a transaction code to an executable program, although you may if you wish, and you do not have to use the Screen Painter to create any screens for it.

When you run an executable program, the program flow is controlled by a series of processors in the runtime environment. These trigger a set sequence of events, to which you can react in corresponding processing blocks in the program. The flow an executable program conforms to the programming standard DIN 66220.



The program starts with an initialization phase, followed by a selection screen. It then retrieves, processes, and displays data. This process is best suited to reading and displaying data (reporting). Because of this, executable programs are often known as **reports**. You can either program the blocks for selecting and reading data in your program, or you can use a logical database instead.

Linking to a Logical Database [Page 947]

Event Blocks in Executable Programs [Page 952]

Linking to a Logical Database

When you create an executable program, you can specify the name of a logical database in its program attributes. A logical database [Page 1163] is a special ABAP program that combines the contents of certain database tables. You can use a logical database with any number of executable programs. If you enter an asterisk (*) for the logical database, the system uses a standard database that controls the selection screen, but does not read any data.

Running Programs With and Without Logical Databases

When you run an executable program that has a logical database linked to it, the two programs function like a single executable program, whose processing blocks are called by the runtime environment in a particular order. For further details, refer to the diagram in the section Logical Databases and Contexts [Page 60].

The main functions of a logical database are to call the selection screen and read data. If there is an appropriate logical database in the system, you no longer need to program these functions in your own executable program.

Executable programs that do not use a logical database, but define their own selection screen and read data using Open SQL statements are sometimes called SQL reports.

Some SQL reports have only one processing block - the default block START-OF-SELECTION. In this case (and only in this case), an executable program behaves like a classic sequentially-processed program. However, as soon as you process a selection screen in the program (for example, filling input fields or checking user input), you have to include further processing blocks (INITIALIZATION, AT SELECTION-SCREEN). If you use a logical database, the program is fully event-driven. The most important event used with logical databases is GET. In it, the logical database passes a database entry to the executable program.

For an overview of events and the sequence in which they are processed, refer to Event Blocks in Executable Programs [Page 952]. You can, of course, program your own database accesses using Open SQL in any event block.

The following example compares two simple executable programs that read data from the hierarchical database tables SPFLI and SFLIGHT. One does not use a logical database, the other uses the logical database F1S. Both programs generate the same list.

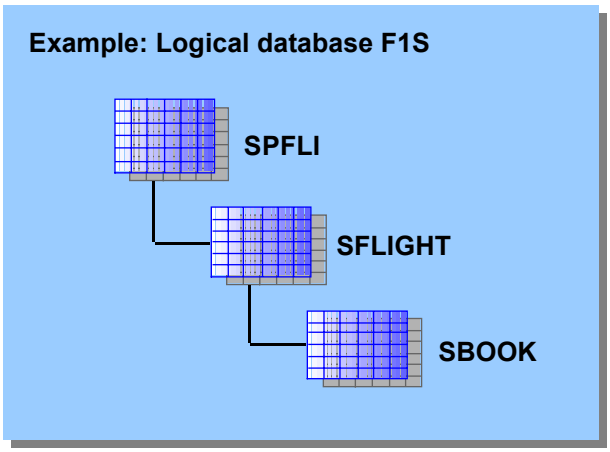
Without logical database:

```
PROGRAM READ_TABLES.

DATA: WA_SPFLI TYPE SPFLI,
      WA_SFLIGHT TYPE SFLIGHT.

SELECT-OPTIONS: SEL_CARR FOR WA_SPFLI-CARRID,
                ...

SELECT CARRID CONNID CITYFROM CITYTO
       FROM SPFLI
       INTO CORRESPONDING FIELDS OF WA_SPFLI
       WHERE CARRID IN SEL_CARR.

  WRITE: / WA_SPFLI-CARRID,
           WA_SPFLI-CONNID,
```

```
            WA_SPFLI-CITYFROM,
            WA_SPFLI-CITYTO.
  SELECT FLDATE
         FROM SFLIGHT
         INTO CORRESPONDING FIELDS OF WA_SFLIGHT
         WHERE CARRID = WA_SPFLI-CARRID
           AND CONNID = WA_SPFLI-CONNID.
    WRITE: / WA_SFLIGHT-FLDATE.
  ENDSELECT.
ENDSELECT.
```

This program does not need any explicitly-declared processing blocks. All non-declarative statements automatically belong to the default event block START-OF-SELECTION. The selection screen, work areas for data, and the SELECT statements all have to be written in the program itself.

With logical database



The logical database F1S is entered in the attributes of the executable program:

```
PROGRAM READ_TABLES.
NODES: SPFLI, SFLIGHT.
GET SPFLI FIELDS CARRID CONNID CITYFROM CITYTO.
  WRITE: / SPFLI-CARRID,
           SPFLI-CONNID,
           SPFLI-CITYFROM,
           SPFLI-CITYTO.
GET SFLIGHT FIELDS FLDATE.
  WRITE: / SFLIGHT-FLDATE.
```

If you compare this program with the one that does not use a logical database, you will see that it does not have to define a selection screen or read the data itself.

These tasks are performed by the logical database. The NODES statement declares the work areas into which the logical database places the data that it reads.

Controlling the Logical Database from the Program

When you use a logical database, you can control the selection screen and the depth to which the logical database reads data. In the NODES statement, you specify the nodes of the logical database that you want to use in the program. The statement declares table work areas for these nodes in the program. These are data objects with the same name and structure as the corresponding node. The selection screen of the logical database only contains input fields for the nodes that you have declared in the NODES statement.

You control the depth to which the logical database reads data by defining GET event blocks. You do not have to program a GET event block for every node in the NODES statement. The logical database always reads the necessary data, that is, at least the key fields, for all nodes along the path from the root node to the lowest-level node for which a GET event block is defined. If you have not defined a particular node for processing, the logical database ignores it and all of its subordinate nodes.

Each GET event block knows both the fields of the current node and the fields that it has read from all of the nodes higher in the hierarchy along the current read path. At the end of a hierarchy level, the logical database resets the table work area in the program to HEX 00.

You can also change the depth to which the logical database is read by terminating a GET event block using the EXIT or CHECK statement. When you terminate a GET event, the logical database ignores any subordinate nodes in the hierarchy.

To control the amount of data read in a GET event block, use a field list in the GET statement. This only works if the relevant node in the logical database has been defined for field selections.

Example of read depth using logical database F1S:

```
NODES: SPFLI, SFLIGHT, SBOOK.
  ...
  GET SPFLI.
```

The logical database does not read any data from the table SFLIGHT or SBOOK.

```
NODES: SPFLI, SBOOK.
  ...
  GET SBOOK.
```

The logical database reads data from the tables SPFLI and SFLIGHT, since they are on the access path for the node SBOOK. However, you cannot access data from SFLIGHT in the program.

Advantages of Using Logical Databases

Logical databases save you having to define a selection screen and read data from the database in every program. The program does not have to specify how to retrieve the data, but instead only has to process it and display it on the screen.

An executable program can only work with one logical database, but each logical database can be used by several programs. This offers considerable advantages over integrating the database accesses with SELECT statements into each executable program. It means that you only have to code identical access paths once. The same applies to coding authorization checks.

When you use logical databases, most executable programs benefit from having

- an easy-to-use and standard user interface

- check functions, which check that user input is complete, correct, and plausible
- meaningful data selection
- central authorization checks for database accesses
- good read access performance (for example, with views) while retaining the hierarchical data view determined by the application logic.

Even though you are using central logical databases, the program itself remains flexible because:

- You can still create your own selection screens for each program
- You can code your own functions in any event block in the program. For example, you may want to write user dialogs for further authorization or plausibility checks on the selection screen.

Programming Logical Databases

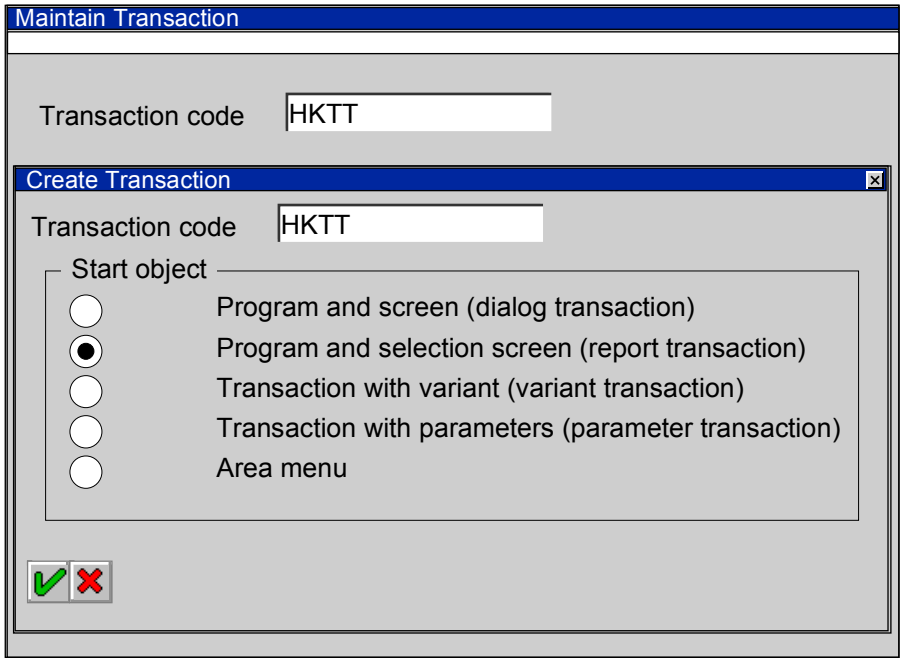
To create, edit, or display a logical database, use the Logical Database Builder in the ABAP Workbench (Transaction SLDB, or choose *Tools* → *ABAP Workbench*, followed by *Development* → *Programming environment* → *Logical databases*. For further information about programming logical databases, refer to the section Logical Databases [Page 1163].

Report Transactions

You can assign a transaction code to any executable program. This allows users to start it as they would an normal transaction [Page 982].

To define a transaction code:

1. From the ABAP Workbench, choose *Development* → *Other tools* → *Transactions*. The *Maintain Transactions* screen appears.
2. Enter a transaction code and choose *Create*.
3. In the dialog box, choose *Program and selection screen* (report transaction).



   It is important to choose *Report transaction* to ensure that the executable program is started by the same processors in the ABAP runtime environment as when you start it directly from the ABAP Editor. If you choose Dialog transaction from the above screen, the program has to be controlled using screen logic, like a module pool.

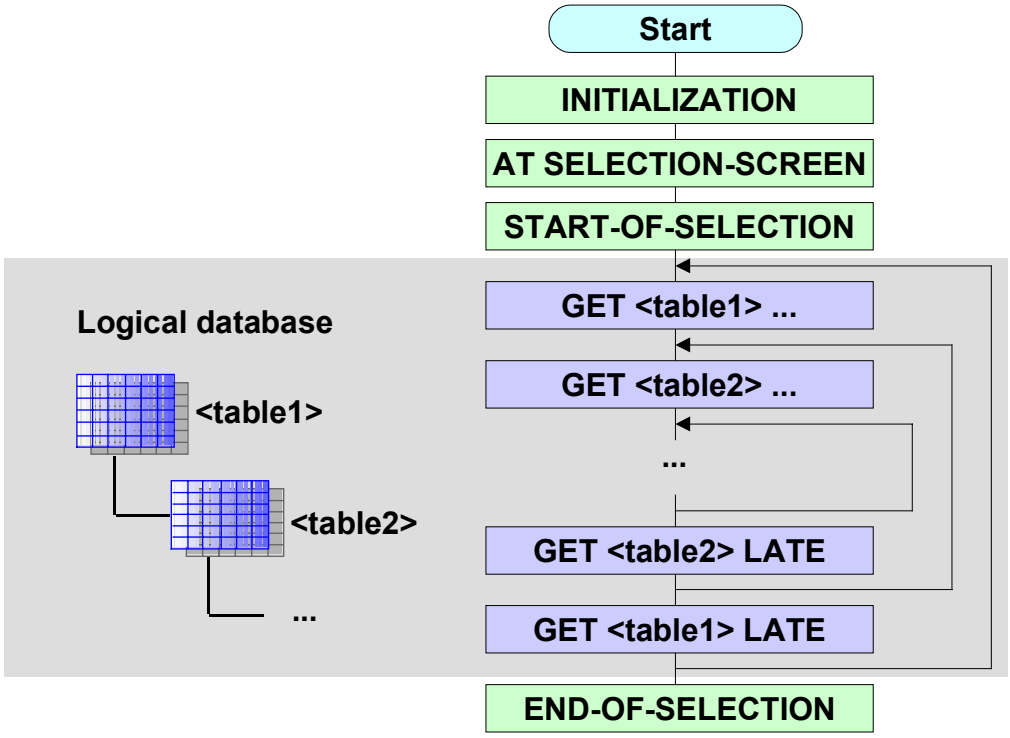
4. On the next screen, enter a transaction text, the program, and the selection screen.
5. Save the transaction code and assign it to the relevant development class.

When you define a transaction code, you can use one of the selection screens [Page 681] defined in the program as the initial screen. The standard selection screen is proposed as a default, but you can overwrite it.

For an example of a report transaction, refer to Calling User-Defined Selection Screens [Page 726].

Event Blocks in Executable Programs

When you run an executable program, the program flow is controlled by the external events in the ABAP runtime environment. The following diagram shows the sequence of the events:



The events in the gray box are only processed if you have entered a logical database in the program attributes. The AT SELECTION-SCREEN event is only processed if a selection screen is defined in the program or the logical database linked to the program. The other events occur when any executable program is run. (See also Processing Blocks in ABAP Programs [Page 49].)

As well as these events, there are others that can, as in other ABAP programs, occur when a list is created (TOP-OF-PAGE, END-OF-PAGE), and in interactive lists (AT LINE-SELECTION, AT USER-COMMAND). For more information about these events, refer to Lists [Page 771].

If you want to handle an event, you must define the corresponding event block in your program. If you do not define the event block, there is no reaction to the event.

Description of Events [Page 953]

Leaving Event Blocks [Page 966].

Description of Events

This section describes in more detail the events that occur when you run an executable program.

The following events occur when you run a typical executable program with a logical database:

Event	Occurs
INITIALIZATION [Page 954]	Before the standard selection screen is displayed
AT SELECTION-SCREEN [Page 956]	After user input on a selection screen has been processed, but while the selection screen is still active
START-OF-SELECTION [Page 957]	After the standard selection screen has been processed, before data is read from the logical database
GET <table> [Page 958]	After the logical database has read an entry from the node <table>
GET <table> LATE [Page 961]	After all of the nodes of the logical database have been processed that are below <table> in the database hierarchy
END-OF-SELECTION [Page 963]	After all data has been read by the logical database

List processor events:

Event	Occurs
TOP-OF-PAGE	In list processing when a new page starts
END-OF-PAGE	In list processing when a page ends
AT LINE-SELECTION	When the user triggers the predefined function code PICK
AT PF<nn>	When the user triggers the predefined function code PF<nn>
AT USER-COMMAND	When the user triggers a function code defined in the program

For more information about these events, refer to Lists [Page 771].

INITIALIZATION

SUBMIT [Page 1018]

AT SELECTION-SCREEN OUTPUT [Page 743]

AT SELECTION-SCREEN OUTPUT [Page 743]

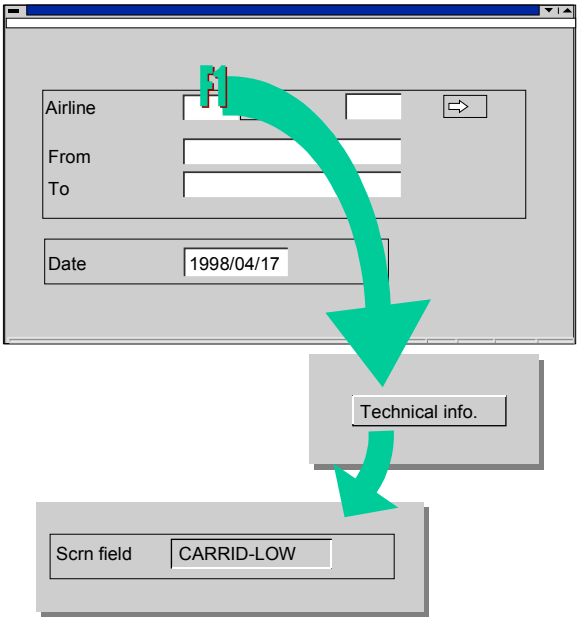
This event occurs before the standard selection screen is called. You can use it, for example, to initialize the input fields of the standard selection screen. This is the only possible way to change the default values of parameters or selection criteria defined in logical databases. To change a selection criterion, you must fill at least the components <seltab>-SIGN, <seltab>-OPTION, and <seltab>-LOW of the selection table <seltab>, otherwise it remains undefined.

If you want to initialize input fields of the logical database, you need to find out the names of the fields. To do this for the logical database SAPDB<ldb>, use Transaction SLDB or choose *Tools* →
ABAP Workbench, followed by *Development* → *Programming environ.* → *Logical databases*. You can also display the technical information for the required field on the selection screen. To do this, call the F1 help for the required field and then choose *Technical info*. In the field *Scrn Field* of the following dialog box, you then see the name of the field used in the program.

The following program is connected to the logical database F1S.

```
REPORT EVENT_DEMO.
PARAMETERS DATUM TYPE SY-DATUM DEFAULT SY-DATUM.
NODES SPFLI.
```

When you start the program, the selection screen appears:

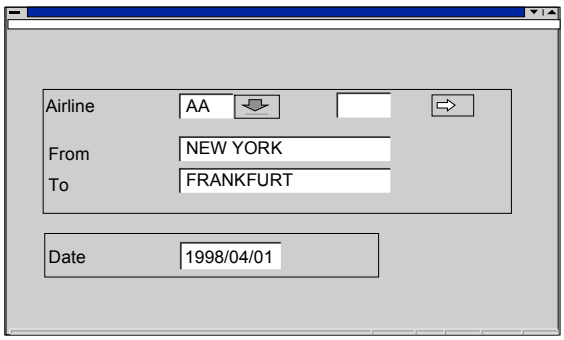


Only the parameter *DATUM* is defined in the program itself. All of the other input fields are defined in the logical database F1S.

When you call the F1 help for the first input field for *Airline* and then choose *Technical info.* the field name CARRID-LOW appears in the *Scrn field* field. This is the component of the selection table that corresponds to the input field. From this, you see that the name of the selection criterion is CARRID. In the same procedure as described above, you find that the parameters of the input fields *From* and *To* are named CITY_FR and CITY_TO.
Suppose we now change the program as follows:

```
REPORT EVENT_DEMO.
PARAMETERS DATUM TYPE SY-DATUM DEFAULT SY-DATUM.
NODES SPFLI.
INITIALIZATION.
  CITY_FR = 'NEW YORK'.
  CITY_TO = 'FRANKFURT'.
  CARRID-SIGN   = 'I'.
  CARRID-OPTION = 'EQ'.
  CARRID-LOW    = 'AA'.
  APPEND CARRID.
  DATUM+6(2) = '01'.
```

The selection screen is now filled with default values as follows:



AT SELECTION-SCREEN

The event AT SELECTION-SCREEN is the basic form of a whole series of events that occur while the selection screen is being processed.

The standard selection screen in an executable program or in the logical database linked to it is automatically called between the INITIALIZATION and START-OF-SELECTION events. When you call the selection screen, and when users interact with it, the ABAP runtime environment generates selection screen events, which occur between INITIALIZATION and START-OF-SELECTION.

You can define event blocks for these events in your program to change the selection screen or process user input.

For further information about selection screen events, refer to Processing Selection Screens [Page 739].

START-OF-SELECTION

This event occurs after the selection screen has been processed and before data is read using the logical database. You can use it to prepare for reading data and creating the list by, for example, setting values for internal fields and writing introductory notes on the output list.

In an executable program, any non-declarative statements that occur between the REPORT or PROGRAM statement and the first processing block are also processed in the START-OF-SELECTION block. See also Defining Event Blocks [Page 941].

The following program is connected to the logical database F1S.

```
REPORT EVENT_DEMO.

NODES SPFLI.

...

AT SELECTION-SCREEN.

  ...

START-OF-SELECTION.

  WRITE: / 'List of Flights' COLOR COL_HEADING,
         / 'Created by', SY-USERID, 'on', SY-DATUM.
  ULINE.

GET SPFLI.

  ...
```

GET

This is the most important event for executable programs that use a logical database. It occurs when the logical database has read a line from the node <table> and made it available to the program in the work area declared using the statement NODES <table>.

When you define the corresponding event block in the program, you can specify a field list if the logical database supports field selection for this node:

GET <table> [FIELDS <f_1> <f_2>...].

You can process the data in the work area in this event block. For example, you can write it directly to a list, or store it in a sequential dataset (internal table or extract) so that you can process it later.

The logical database reads **all** columns from **all** nodes that are not designated for field selection in the logical database and which are superior to <table> on the access path of the logical database. This works independently of whether you have defined GET event blocks for these nodes or not. However, you can only access the data of the nodes for which you have declared a work area in the NODES statement.

At the end of a hierarchy level of the logical database, all of the fields in the work area <table> are set to the value Hexadecimal 00. If you are using extracts, there is a special sort rule for fields with the content hexadecimal 00. For further information, refer to Sorting Extract Datasets [Page 340].

Performance can be much better for tables that are designated for field selection in the logical database. If there are nodes of this type above <table> in the hierarchy of the logical database for which there are no GET event blocks, the data for all columns is only read for the nodes for which there is a NODES statement in the program. For nodes without a NODES statement, only the key fields are read. The logical database needs the key fields to construct an access path.

You can use the FIELDS option to specify explicitly the columns of a node that the logical database should read. With the FIELDS option, the logical database program reads only the fields <f_1> <f_2> ... and the key fields from the database table <table>. However, the node <table> must have been designated for field selection in the logical database.

Using FIELDS can result in much better response times than when the logical database has to read all of the columns of the node.

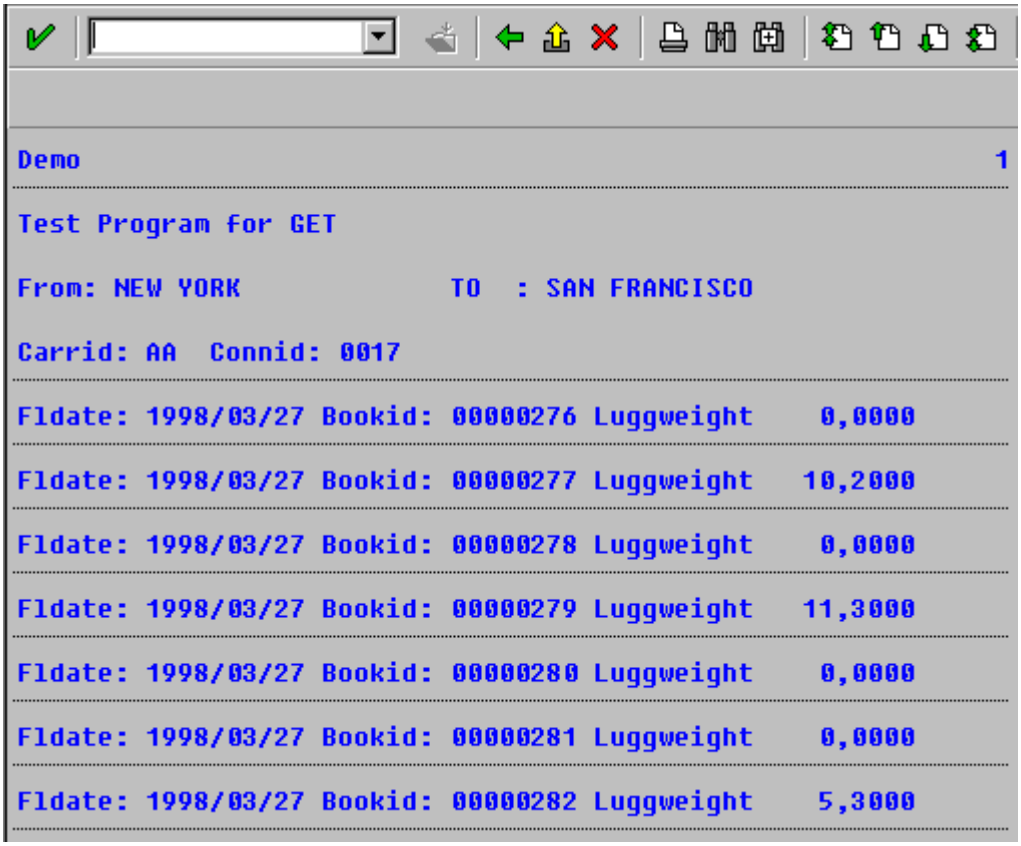
All fields of the node <table> that are not key fields and are not listed after FIELDS, are not read by the logical database. The contents of the corresponding components of the table work area <table> are set to hexadecimal 00. This means that they are also set to hex 00 during the GET events of the nodes below <table> in the hierarchy. You should therefore not use these fields in your program or call subroutines that work with them. If you use the GET event block to fill an extract dataset, remember that they have a special sort rule for fields with the contents hexadecimal 00.

The following program is connected to the logical database F1S.

```
REPORT EVENT_DEMO.

NODES: SPFLI, SFLIGHT, SBOOK.

START-OF-SELECTION.
  WRITE 'Test Program for GET'.
```

```
GET SPFLI.
  SKIP.
  WRITE: / 'From:', SPFLI-CITYFROM,
           'TO  :', SPFLI-CITYTO.

GET SFLIGHT.
  SKIP.
  WRITE: / 'Carrid:', SFLIGHT-CARRID,
           'Connid:', SFLIGHT-CONNID.
  ULINE.

GET SBOOK.
  WRITE: / 'Fldate:',    SFLIGHT-FLDATE,
           'Bookid:',    SBOOK-BOOKID,
           'Luggweight', SBOOK-LUGGWEIGHT.
  ULINE.
```

The table work area SFLIGHT is also used in the event block for GET SBOOK. Depending on what you enter on the selection screen, the beginning of the list display might look like this:



In the logical database F1S, the nodes SFLIGHT and SBOOK are designated for field selection. This means that you can specify a field list in their GET event blocks:

```
REPORT EVENT_DEMO.
```

```
NODES: SFLIGHT, SBOOK.
GET SFLIGHT FIELDS CARRID CONNID.
 ...
GET SBOOK FIELDS BOOKID.
 ...
GET SFLIGHT LATE FIELDS PLANETYPE.
 ...
```

In this case, the logical database reads the following fields:

- MANDT, CARRID, CONNID, FLDATE, and PLANETYPE from SFLIGHT
- MANDT, CARRID, CONNID, FLDATE, and BOOKID from SBOOK

The system reads the fields MANDT and FLDATE from SFLIGHT, even though they are not specified in the field list, since they belong to the table key.

Only the key fields of SBOOK are read.

GET … LATE

This event is triggered when all of the data records for a node of the logical database have been read.

When you define the corresponding event block in the program, you can, as with GET [Page 958], specify a field list if the logical database supports field selection for this node:

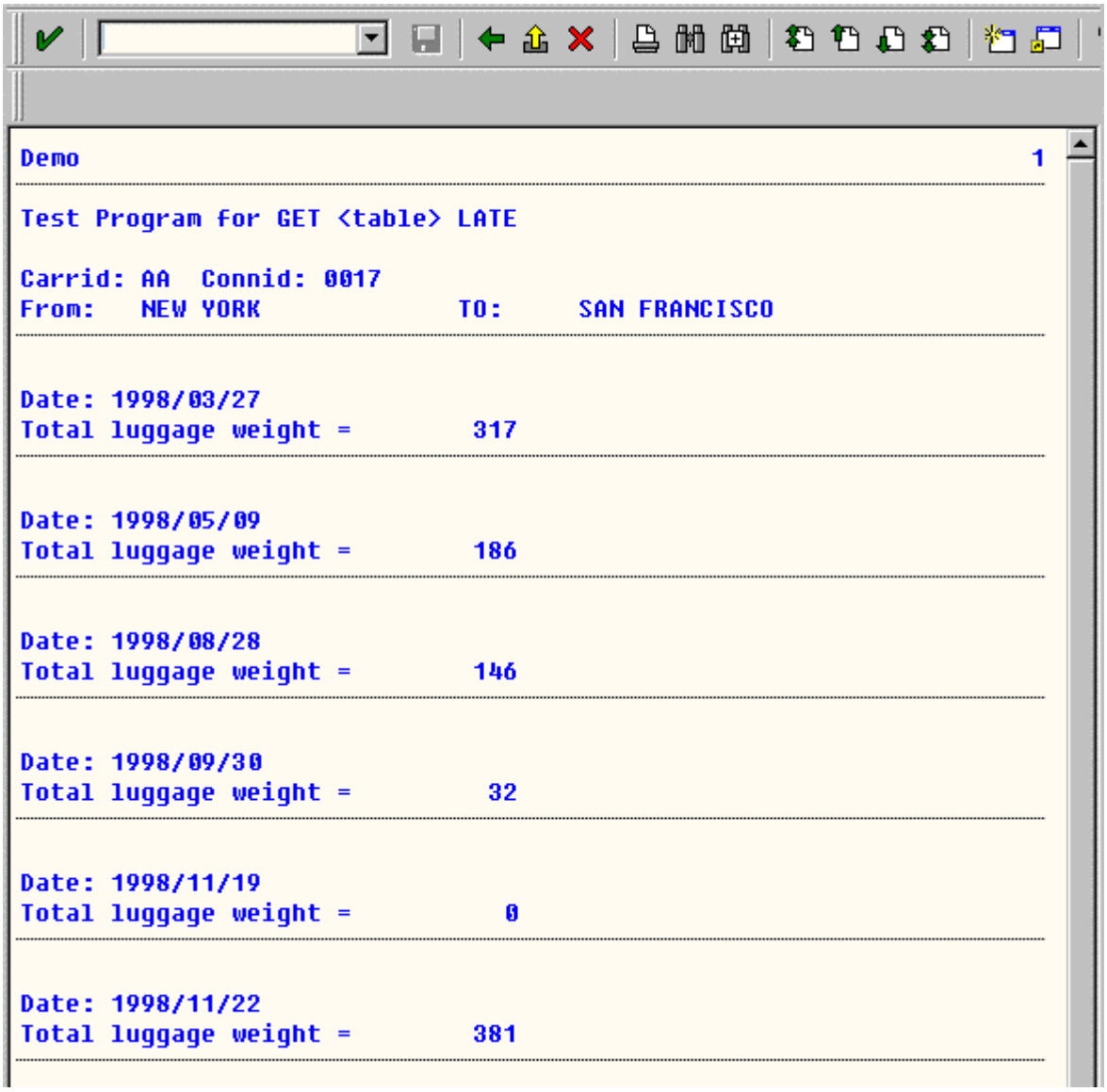
GET <table> LATE [FIELDS <f_1> <f_2>...].

You can use the event block for processing steps that should occur at the end of the block, like, for example, calculating statistics.

The following program is connected to the logical database F1S.

```
REPORT EVENT_DEMO.

NODES: SPFLI, SFLIGHT, SBOOK.

DATA WEIGHT TYPE I VALUE 0.

START-OF-SELECTION.

  WRITE 'Test Program for GET <table> LATE'.

GET SPFLI.

  SKIP.
  WRITE: / 'Carrid:', SPFLI-CARRID,
           'Connid:', SPFLI-CONNID,
         / 'From:  ', SPFLI-CITYFROM,
           'To:    ', SPFLI-CITYTO.
  ULINE.

GET SFLIGHT.

  SKIP.
  WRITE: / 'Date:', SFLIGHT-FLDATE.

GET SBOOK.

  WEIGHT = WEIGHT + SBOOK-LUGGWEIGHT.

GET SFLIGHT LATE.

  WRITE: / 'Total luggage weight =', WEIGHT.
  ULINE.
  WEIGHT = 0.
```

The total luggage weight is calculated for each flight in the event GET SBOOK, and then displayed in the list and reset in the event GET SFLIGHT LATE. Depending on the values you enter on the selection screen, the beginning of the list might look like this:


Demo 1
Test Program for GET <table> LATE
Carrid: AA Connid: 0017
From: NEW YORK TO: SAN FRANCISCO
Date: 1998/03/27
Total luggage weight = 317
Date: 1998/05/09
Total luggage weight = 186
Date: 1998/08/28
Total luggage weight = 146
Date: 1998/09/30
Total luggage weight = 32
Date: 1998/11/19
Total luggage weight = 0
Date: 1998/11/22
Total luggage weight = 381

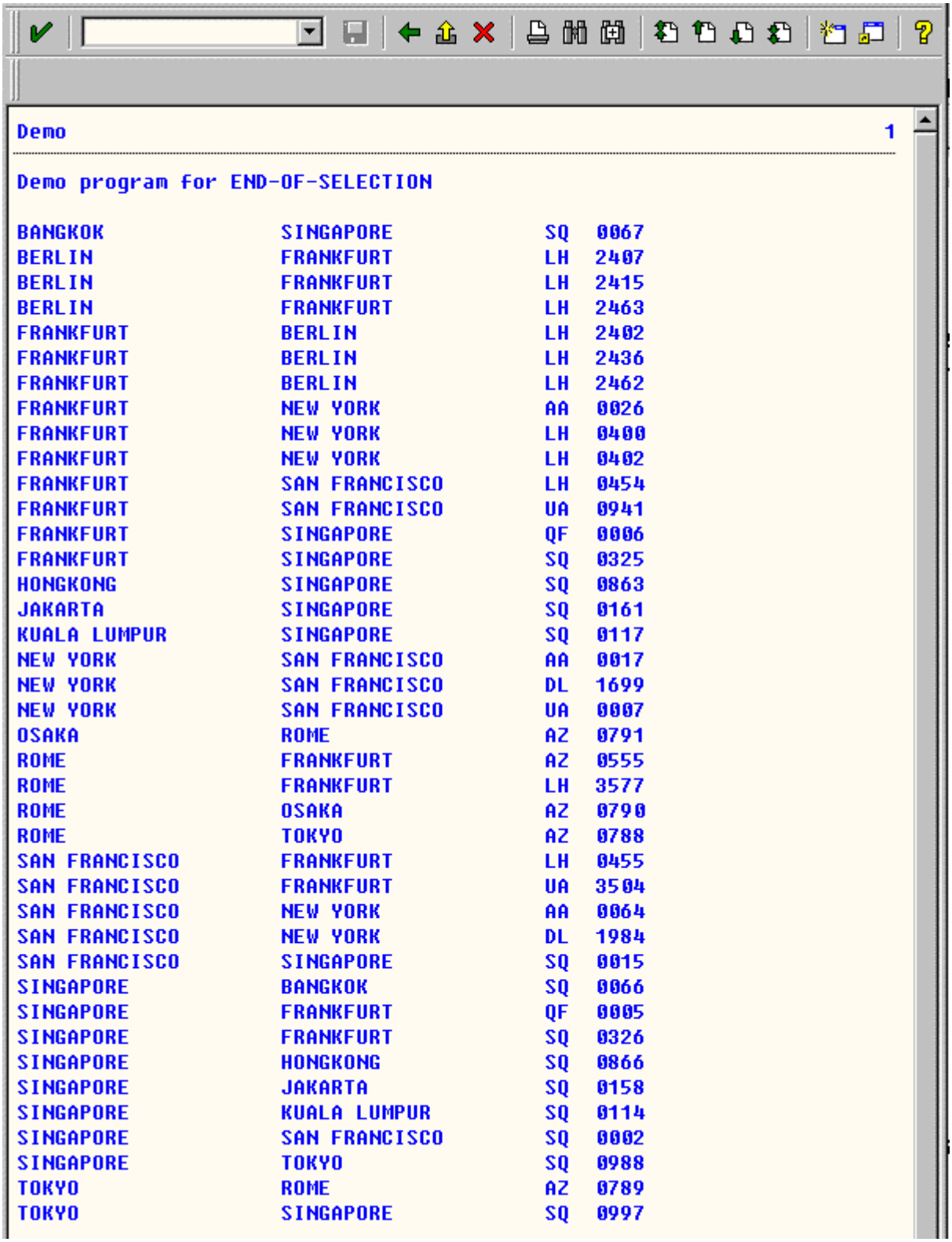
END-OF-SELECTION

This is the last of the events called by the runtime environment to occur. It is triggered after all of the data has been read from the logical database, and before the list processor is started. You can use the corresponding event block to process and format the data that the program has stored in internal tables or extracts during the various GET events.

The following program is connected to the logical database F1S.

```
REPORT EVENT_DEMO.

NODES SPFLI.

DATA: SPFLI_TAB TYPE SORTED TABLE OF SPFLI
                WITH UNIQUE KEY CITYFROM CITYTO CARRID CONNID,
      SPFLI_LINE TYPE SPFLI.

START-OF-SELECTION.

   WRITE 'Demo program for END-OF-SELECTION'.
   SKIP.

GET SPFLI FIELDS CARRID CONNID CITYFROM CITYTO.

  MOVE-CORRESPONDING SPFLI TO SPFLI_LINE.
  INSERT SPFLI_LINE INTO TABLE SPFLI_TAB.

END-OF-SELECTION.

  LOOP AT SPFLI_TAB INTO SPFLI_LINE.
    WRITE: / SPFLI_LINE-CITYFROM,
             SPFLI_LINE-CITYTO,
             SPFLI_LINE-CARRID,
             SPFLI_LINE-CONNID.
  ENDLOOP.
```

This program fills a sorted table with data from the logical database in the GET SPFLI event, and displays them in a list in the END-OF-SELECTIOn event. Depending on what you enter on the selection screen, the beginning of the list display might look like this:



Demo 1

Demo program for END-OF-SELECTION

BANGKOK	SINGAPORE	SQ	0067
BERLIN	FRANKFURT	LH	2407
BERLIN	FRANKFURT	LH	2415
BERLIN	FRANKFURT	LH	2463
FRANKFURT	BERLIN	LH	2402
FRANKFURT	BERLIN	LH	2436
FRANKFURT	BERLIN	LH	2462
FRANKFURT	NEW YORK	AA	0026
FRANKFURT	NEW YORK	LH	0400
FRANKFURT	NEW YORK	LH	0402
FRANKFURT	SAN FRANCISCO	LH	0454
FRANKFURT	SAN FRANCISCO	UA	0941
FRANKFURT	SINGAPORE	QF	0006
FRANKFURT	SINGAPORE	SQ	0325
HONGKONG	SINGAPORE	SQ	0863
JAKARTA	SINGAPORE	SQ	0161
KUALA LUMPUR	SINGAPORE	SQ	0117
NEW YORK	SAN FRANCISCO	AA	0017
NEW YORK	SAN FRANCISCO	DL	1699
NEW YORK	SAN FRANCISCO	UA	0007
OSAKA	ROME	AZ	0791
ROME	FRANKFURT	AZ	0555
ROME	FRANKFURT	LH	3577
ROME	OSAKA	AZ	0790
ROME	TOKYO	AZ	0788
SAN FRANCISCO	FRANKFURT	LH	0455
SAN FRANCISCO	FRANKFURT	UA	3504
SAN FRANCISCO	NEW YORK	AA	0064
SAN FRANCISCO	NEW YORK	DL	1984
SAN FRANCISCO	SINGAPORE	SQ	0015
SINGAPORE	BANGKOK	SQ	0066
SINGAPORE	FRANKFURT	QF	0005
SINGAPORE	FRANKFURT	SQ	0326
SINGAPORE	HONGKONG	SQ	0866
SINGAPORE	JAKARTA	SQ	0158
SINGAPORE	KUALA LUMPUR	SQ	0114
SINGAPORE	SAN FRANCISCO	SQ	0002
SINGAPORE	TOKYO	SQ	0988
TOKYO	ROME	AZ	0789
TOKYO	SINGAPORE	SQ	0997

Leaving Event Blocks

ABAP contains a series of statements that allow you to leave an event block. They return control to the runtime environment. The subsequent sequence of events in the runtime environment depends on which statement you use and the event block in which you use it.

Leaving Event Blocks Using STOP [Page 967]

Leaving Event Blocks Using EXIT [Page 970]

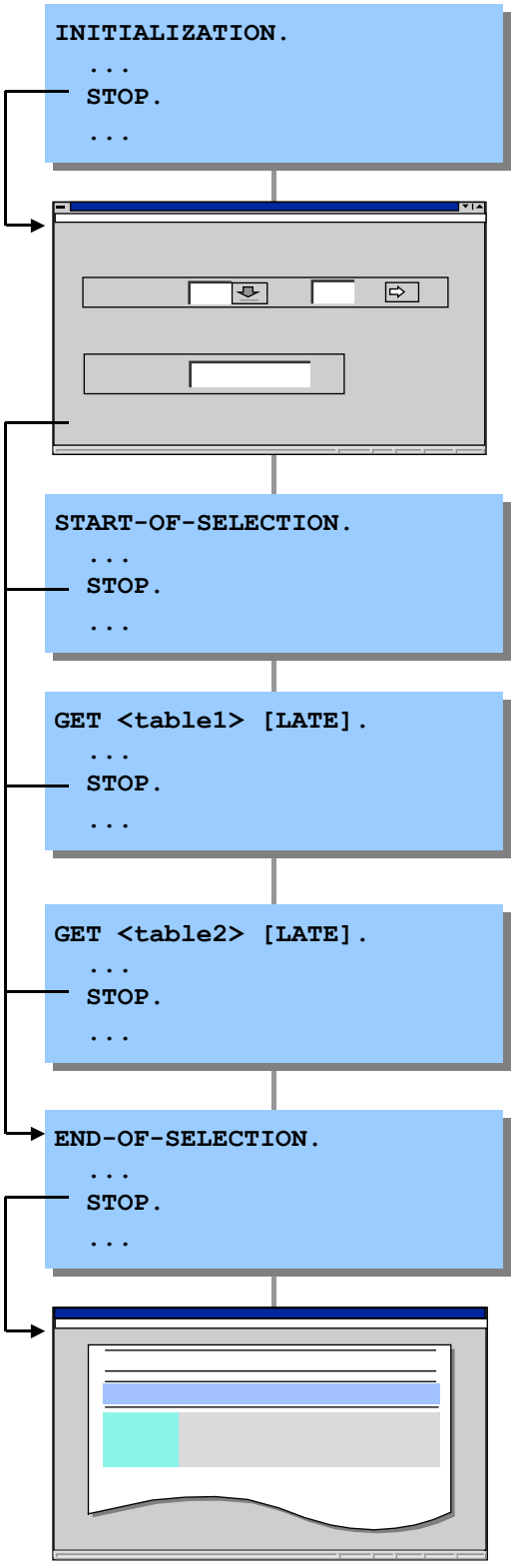
Leaving Event Blocks Using CHECK [Page 974]

Leaving a GET Event Block Using REJECT [Page 979]

Leaving Event Blocks Using STOP

If you use the STOP statement within an event block, the system stops processing the block immediately. The ABAP runtime environment triggers the next event according to the following diagram:

STOP statement



Before and during selection screen processing, the next event in the prescribed sequence is always called. From the AT SELECTION-SCREEN event onwards, the system always jumps from a STOP statement directly to the END-OF-SELECTION statement. Once the corresponding event block has been processed, the system displays the list.

The following program is connected to the logical database F1S.

```
REPORT EVENT_TEST.

NODES: SPFLI, SFLIGHT, SBOOK.

START-OF-SELECTION.
   WRITE 'Test program for STOP'.

GET SBOOK.
   WRITE: 'Bookid', SBOOK-BOOKID.
   STOP.

END-OF-SELECTION.
   WRITE: / 'End of Selection'.
```

This produces the following output:

Test Program for STOP

Bookid 00010001

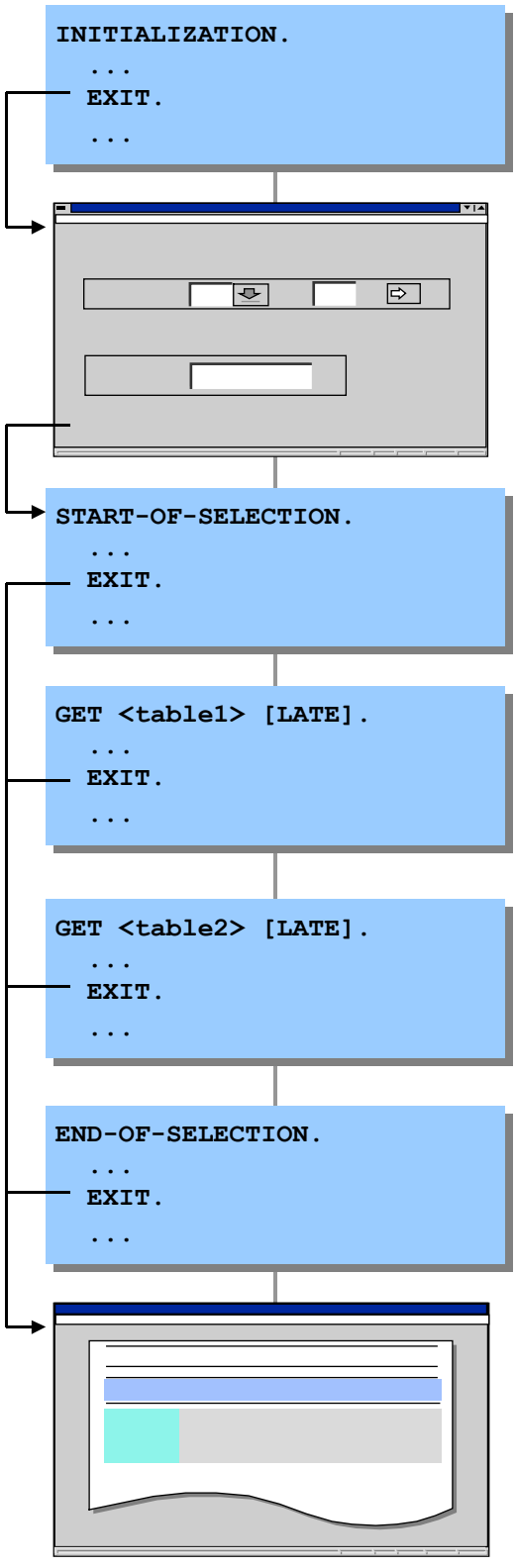
End of Selection

As soon as the first line of SBOOK has been read, the system calls the END-OF-SELECTION event block.

Leaving Event Blocks Using EXIT

If you use the STOP statement within an event block but **not in a loop**, the system stops processing the block immediately. The ABAP runtime environment triggers the next event according to the following diagram:

EXIT statement



Leaving Event Blocks

Before and during selection screen processing, the next event in the prescribed sequence is always called. From the START-OF-SELECTION event onwards, the system starts the list processor directly when the EXIT statement occurs, and displays the list.

If the EXIT statement occurs in a loop using DO, WHILE, or LOOP, it is the loop that terminates, **not** the processing block.

The following program is connected to the logical database F1S.

```
REPORT EVENT_TEST.
NODES: SPFLI, SFLIGHT, SBOOK.
START-OF-SELECTION.
   WRITE 'Test Program for EXIT'.
GET SBOOK.
   WRITE: 'Bookid', SBOOK-BOOKID.
   EXIT.
END-OF-SELECTION.
   WRITE: / 'End of selection'.
```

This produces the following output:
Test Program for EXIT
Bookid 00010001
After the first line of SBOOK has been read, the list is displayed immediately.

The following program is connected to the logical database F1S.

```
REPORT EVENT_TEST.

NODES: SPFLI, SFLIGHT, SBOOK.

DATA FLAG.

AT SELECTION-SCREEN.
  IF CARRID-LOW IS INITIAL.
    FLAG = 'X'.
    EXIT.
  ENDIF.
  ........

START-OF-SELECTION.
  IF FLAG = 'X'.
    WRITE / 'No selection made for CARRID'.
    EXIT.
  ENDIF.

GET SPFLI.
  .........

GET SFLIGHT.
  .........

GET SBOOK.
  .........

END-OF-SELECTION.
   WRITE / 'End of Selection'.
```

If the user does not enter a value for CARRID-LOW, the output appears as follows:

```
No selection made for CARRID
```

After the first EXIT statement, the next event, namely the START-OF-SELECTION block, is processed. After the second EXIT statement, the output list is displayed.

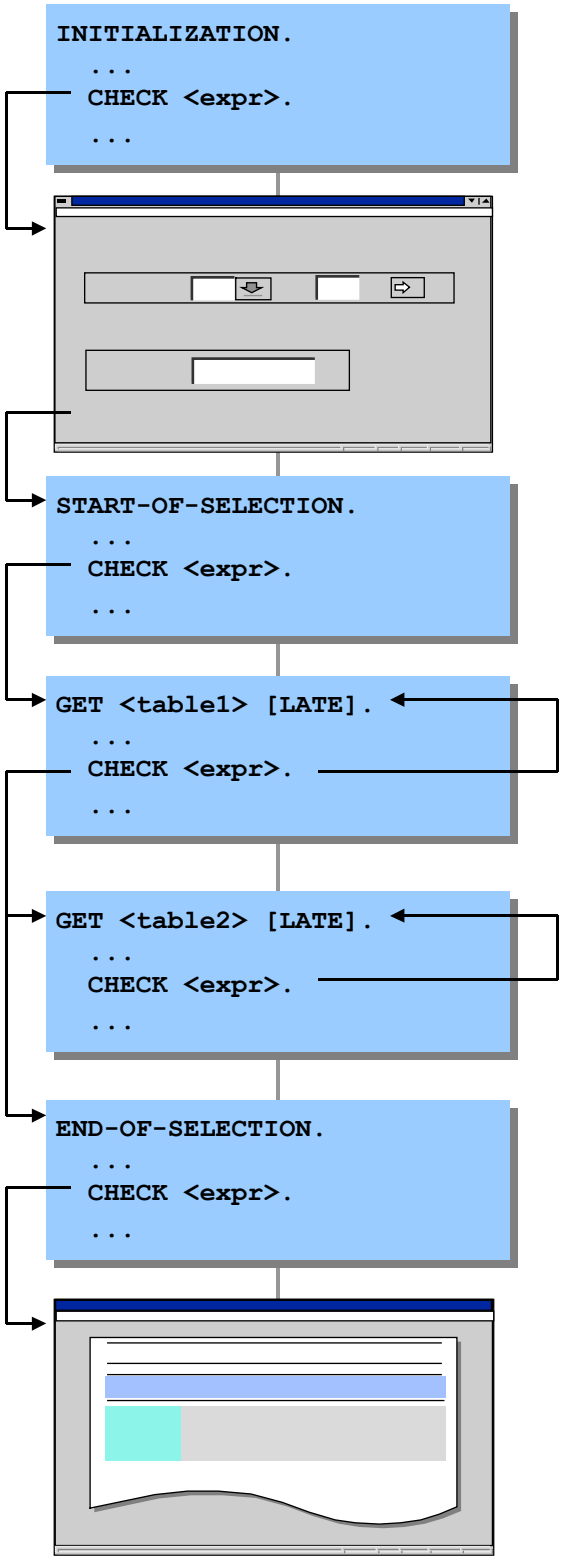
Leaving Event Blocks Using CHECK

If you use the CHECK <expr> statement within an event block but **not within a loop**, and the condition <expr> is not fulfilled, the system exits the processing block immediately.

<expr> can be any logical expression or the name of a selection table. If you specify a selection table and the contents of the corresponding table work are do not fulfill the condition in the selection table, it is the same as a false logical expression.

The ABAP runtime environment triggers the next event according to the following diagram:

CHECK statement



Leaving Event Blocks

The next event in the prescribed sequence is always called.

If the CHECK statement occurs in a loop using DO, WHILE, or LOOP, it is the loop that terminates, **not** the processing block.

Within a GET event block, this means the next GET event at the **same hierarchical level**. When it leaves the event block, the logical database reads the next line of the current node, or the next-highest node if it has already reached the end of the hierarchy level. Nodes that are lower down in the hierarchical structure of the logical database are not processed.

Inside GET events, you can use an extra variant of the CHECK statement:

CHECK SELECT-OPTIONS.

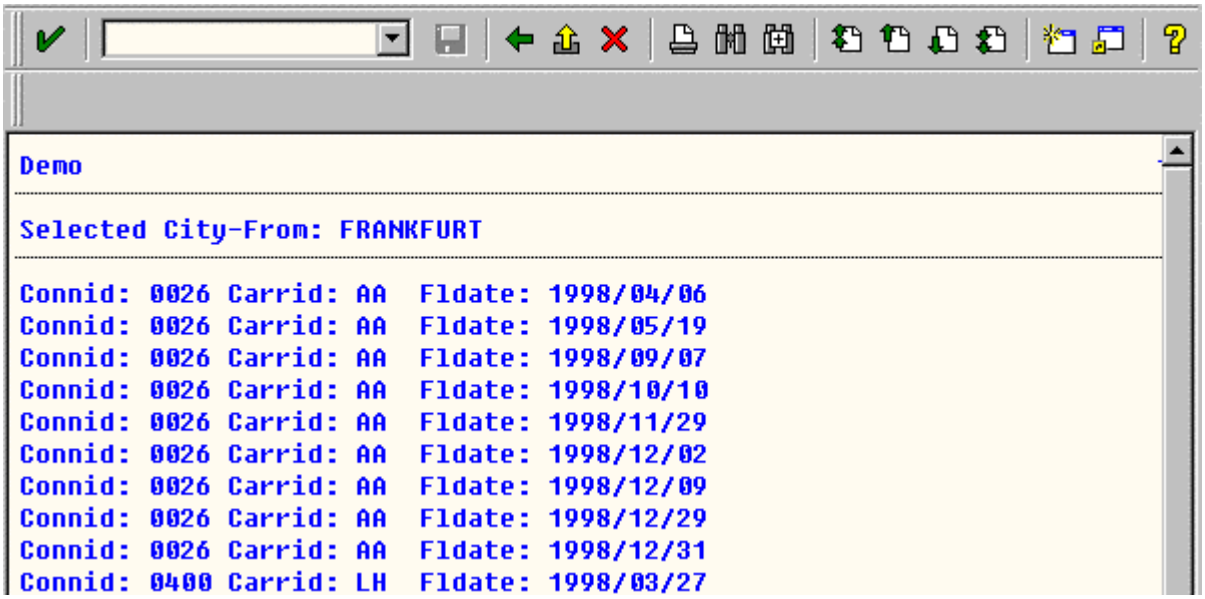
This statement checks the contents of the table work area of the current node against **all** selection tables linked to that node.

Note that CHECK statements for checking database contents in GET events are only processed **after** the data has been read from the logical database. For performance reasons, you should therefore avoid using checks of this kind. Instead, try to check before the data is read, for example, by using dynamic selections.

The following program is connected to the logical database F1S.

```
REPORT EVENT_DEMO.
NODES: SPFLI, SFLIGHT, SBOOK.
START-OF-SELECTION.
  CHECK CITY_FR NE ' '.
  WRITE: / 'Selected City-From:', CITY_FR.
  ULINE.
  CHECK CITY_TO NE ' '.
  WRITE: / ' Selected City-To:', CITY_TO.
  ULINE.
GET SFLIGHT.
  WRITE: / 'Connid:', SFLIGHT-CONNID,
         'Carrid:', SFLIGHT-CARRID,
         'Fldate:', SFLIGHT-FLDATE.
```

If the user enters “Frankfurt” for CITY_FR, but nothing for CITY_TO, the beginning of the list would look like this:

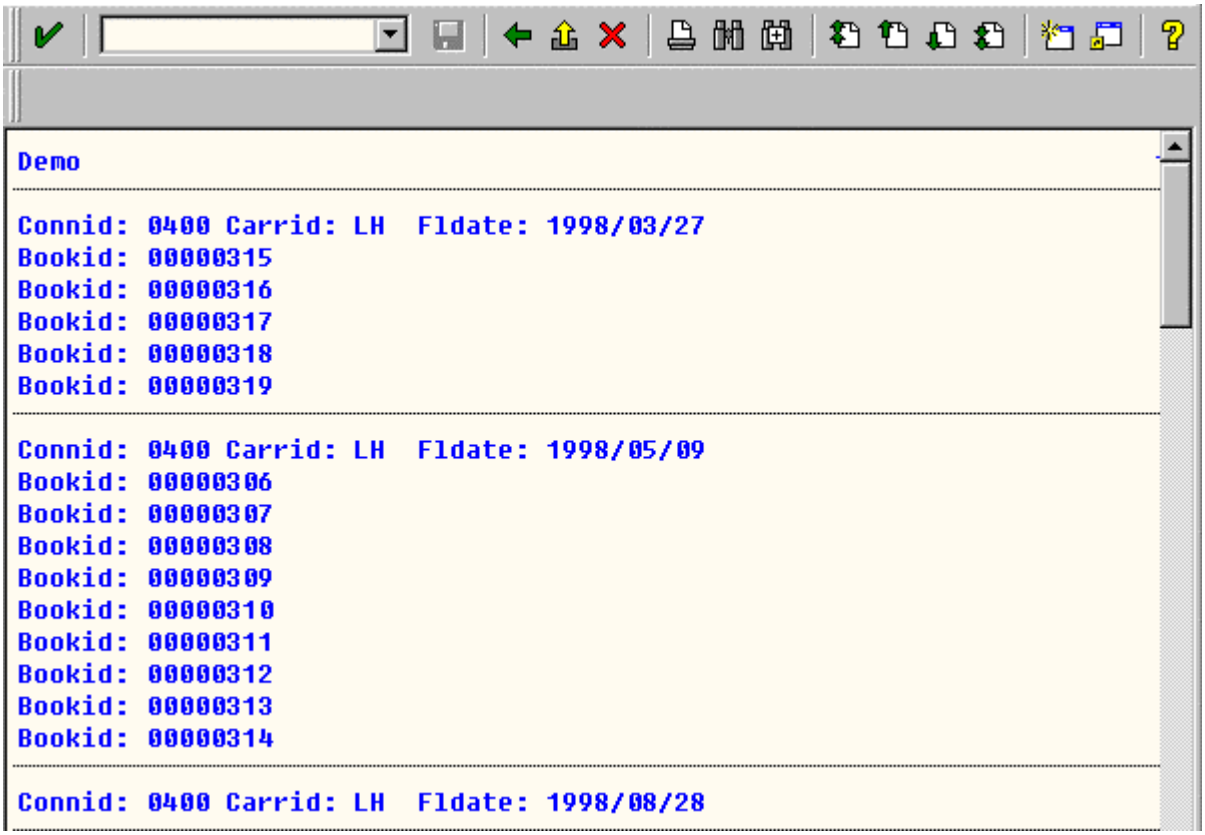


After the second CHECK statement, the system leaves the START-OF-SELECTION block and triggers the event GET SFLIGHT.

The following program is connected to the logical database F1S.

```
REPORT EVENT_DEMO.
NODES: SPFLI, SFLIGHT, SBOOK.
GET SFLIGHT.
  CHECK SFLIGHT-CARRID EQ 'LH'.
  WRITE: / 'Connid:', SFLIGHT-CONNID,
           'Carrid:', SFLIGHT-CARRID,
           'Fldate:', SFLIGHT-FLDATE.
GET SBOOK.
  CHECK SBOOK-BOOKID LT 00000320.
  WRITE: / 'Bookid:', SBOOK-BOOKID.
GET SFLIGHT LATE.
  ULINE.
```

This produces the following output list:

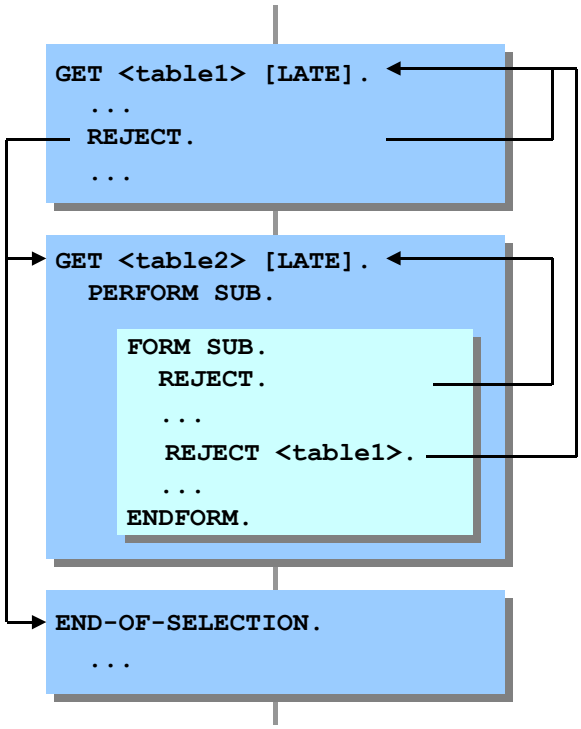


In the example above, all lines of the node SFLIGHT and, if SFLIGHT-CARRID is "LH", all lines of the node SBOOK, must be read. For performance reasons, you should **avoid programming selections as above**. Instead, use the selections of the logical database.

Leaving a GET Event Block Using REJECT

The REJECT statement was specially developed for leaving GET event blocks. Unlike CHECK and EXIT, REJECT always refers to the current GET event block. If CHECK and EXIT occur in a loop, they refer to the loop, and in a subroutine, they always refer to the subroutine. The REJECT statement, on the other hand, allows you to exit a GET event block directly from a loop or a subroutine.

REJECT statement



The statement

REJECT [<dbtab>].

always terminates the processing of the current line of the node of the logical database. Without the optional <dbtab>, the logical database automatically reads the next line of the same node, and the next GET event at the same hierarchy level is triggered. If you use the optional <dbtab>, the logical database reads the next line of the node <dbtab>. The node <dbtab> must occur above the current node in the logical database hierarchy.

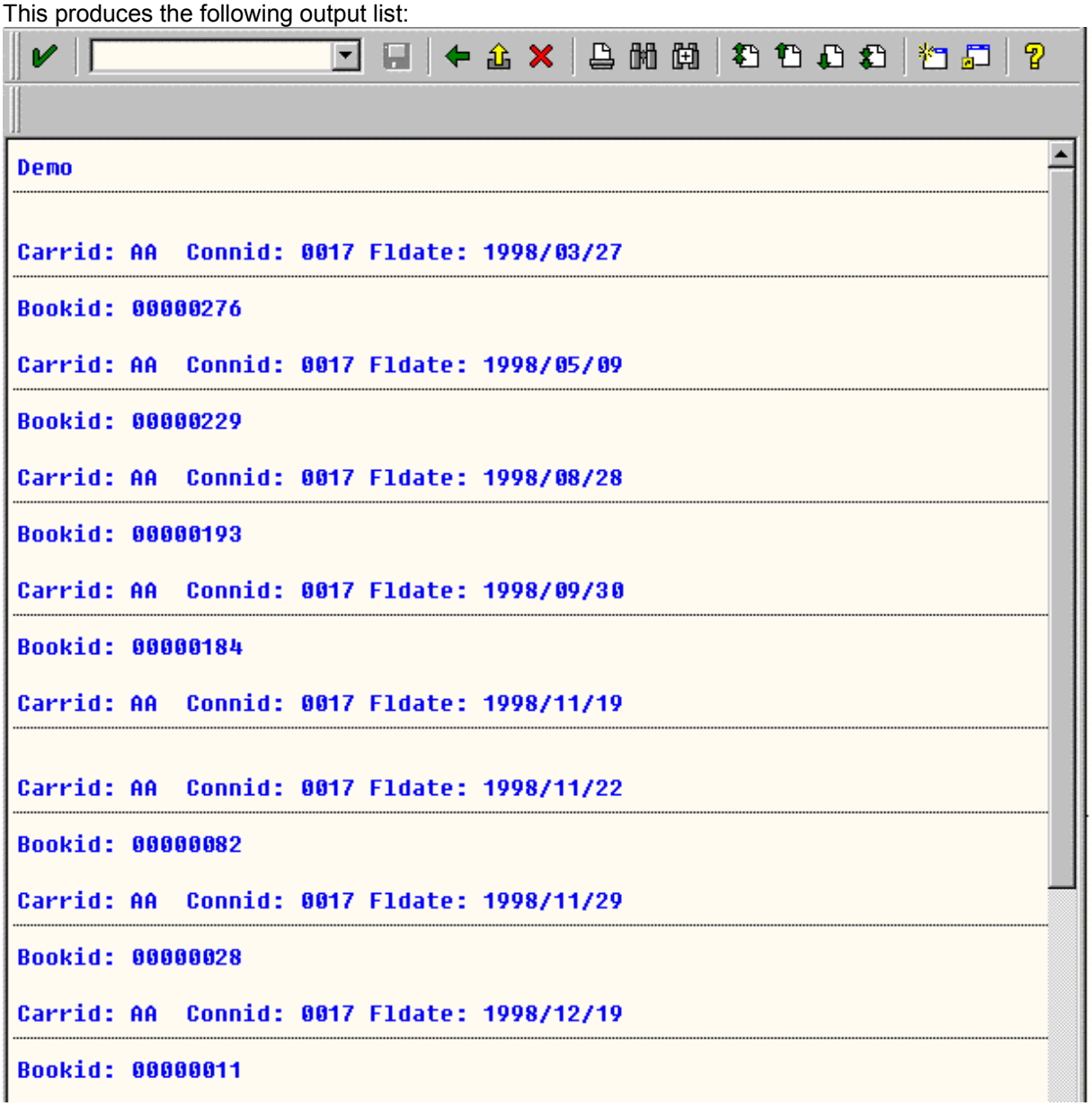
The following program is connected to the logical database F1S.

```
REPORT EVENT_DEMO.
NODES: SPFLI, SFLIGHT, SBOOK.
GET SFLIGHT.
  SKIP.
  WRITE: / 'Carrid:', SFLIGHT-CARRID,
```

Leaving Event Blocks

```
            'Connid:', SFLIGHT-CONNID,
            'Fldate:', SFLIGHT-FLDATE.
  ULINE.
GET SBOOK.
  PERFORM SUB.
FORM SUB.
  WRITE: / 'Bookid:', SBOOK-BOOKID.
  REJECT 'SFLIGHT'.
ENDFORM.
```

This program reads and the first bookings for each flight, since the logical database reads the next line of SFLIGHT after the REJECT statement.
This produces the following output list:



Dialog-Driven Programs: Transactions

Transaktionspflege [Page 995]

Dynprofolgen [Page 1000]

In a dialog-driven program, the program flow is controlled by a series of user dialogs. Dialog-driven programs are typically started using **transaction codes**, which specify the first screen of the program. This initial screen allows users to enter or request information. The screen flow logic then reacts to the user input by calling various modules of ABAP processing logic. It then moves on to the next screen. The corresponding ABAP processing logic might contain statements for displaying data or updating the database.

Suppose a travel agent wants to book a flight. The agent enters the corresponding data on the screen. The system either confirms the desired request, that is, the agent can book the flight and the customer travels on the desired day on the reserved seat to the chosen destination, or the system displays the information that the flight is already booked up.

To fulfill such requirements, a dialog program must offer:

- a user-friendly user interface
- format and consistency checks for the data entered by the user
- an easy way of correcting wrong entries
- access to data by storing it in the database.

ABAP offers a variety of tools and language elements to meet the requirements stated above in the dialog programs.

Dialog Programs: Overview [Page 983]

Example Program

Dialog Programs: Overview

GUI-Status [Page 548]

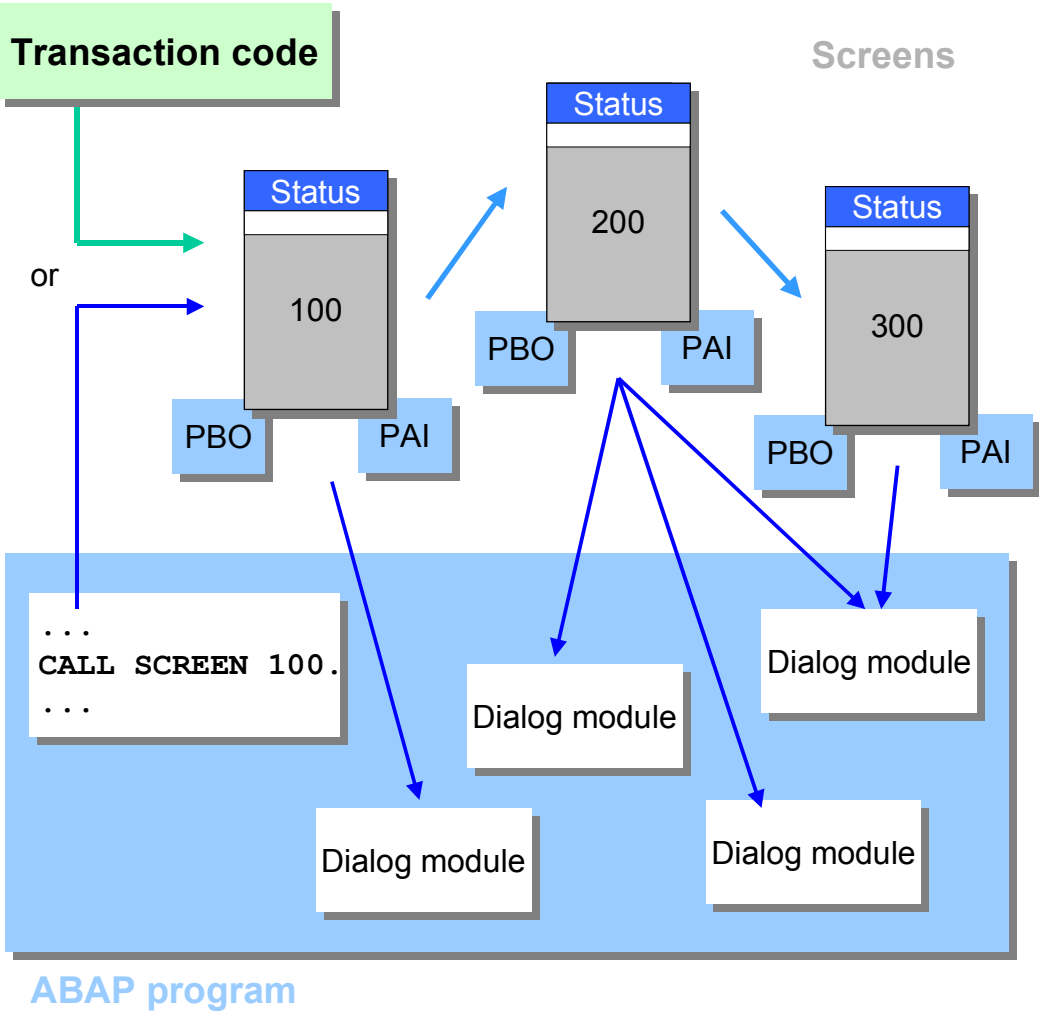
Dialog-driven programs, or any program started using a transaction code, are known as SAP transactions, or just transactions. The term “transaction” is used in several different contexts in the IT world. In OLTP (Online Transaction Processing), where several users are working in one system in dialog mode, the term “transaction” stands for a user request. In conjunction with database updates [Page 1260], it means a change in state in the database.

Programs with type M can only be started using a transaction code, in which an initial screen is defined. Programs with type 1 can be started either using a transaction code, or by entering the program name in one of the transactions SE38 or SA38. Screens call dialog modules in the associated ABAP program from their flow logic. Type M programs serve principally as containers for dialog modules, and are therefore known as **module pools**. Type 1 programs, or function modules can also switch to dialog mode by calling screens using the CALL SCREEN statement. The program code of the corresponding executable program or function pool must then contain the corresponding dialog modules.

Programs that are partially or wholly dialog-driven **cannot be executed in the background**. They are therefore sometimes referred to as **dialog programs**.

Components of a Dialog Program

A dialog-driven program consists of the following basic components:



- Transaction code

  The transaction code starts a screen sequence. You create transaction codes in the Repository Browser in the ABAP Workbench or using Transaction SE93. A transaction code is linked to an ABAP program and an initial screen. As well as using a transaction code, you can start a screen sequence from any ABAP program using the CALL SCREEN statement.

- Screens

  Each dialog in an SAP system is controlled by one or more screens [Page 524]. These screens consist of a screen mask and its flow logic. Since the flow logic influences the program flow, screens are sometimes referred to as “dynamic programs”. You create screens using the Screen Painter in the ABAP Workbench. Each screen belongs to an ABAP program.

  The screen has a layout that determines the positions of input/output fields and other graphical elements such as checkboxes and radio buttons. The flow logic consists of two parts:

– Process Before Output (PBO). This defines the processing that takes place before the screen is displayed.

– Process After Input (PAI). This defines the processing that takes place after the user has chosen a function on the screen.

All of the screens that you call within an ABAP program must belong to that program. The screens belonging to a program are numbered. For each screen, the system stores the number of the screen which is normally displayed next. This screen sequence [Page 1000] can be either linear or cyclic. From within a screen chain, you can even call another screen chain and, after processing it, return to the original chain. You can also override the statically-defined next screen from within the dialog modules of the ABAP program.

- GUI status

  Each screen has a GUI status. This controls the **menu bars**, **standard toolbar**, and **application toolbar**, with which the user can choose functions in the application. Like screens, GUI statuses are independent components of an ABAP program. You create them in the ABAP Workbench using the Menu Painter.

- ABAP Program

  Each screen and GUI status in the R/3 System belongs to one ABAP program. The ABAP program contains the dialog modules that are called by the screen flow logic, and also process the user input from the GUI status. ABAP programs that use screens are also known as dialog programs. In a module pool (type M program); the first processing block to be called is always a dialog module. However, you can also use screens in other ABAP programs, such as executable programs or function modules. The first processing block is then called differently; for example, by the runtime environment or a procedure call. The screen sequence is then started using the CALL SCREEN statement.

  Dialog modules are split into PBO modules and PAI modules. Dialog modules called in the PBO event are used to prepare the screen, for example by setting context-specific field contents or by suppressing fields from the display that are not needed. Dialog modules called in the PAI event are used to check the user input and to trigger appropriate dialog steps, such as the update task.

Passing Data Between ABAP Programs and Screens

How are fields from ABAP programs displayed on the screen? And how is user input on the screen passed back to the ABAP program? Unlike in list programming, you **cannot** write field data to the screen using the WRITE statement. Instead, the system transfers the data by comparing the names of screen fields with the names of the ABAP fields in the program. If it finds a pair of matching names, the data is transferred between the screen and the ABAP program. This happens immediately before and immediately after displaying the screen.

Field Attributes

For all screen fields of a dialog screen, field attributes are defined in the Screen Painter. If a field name in the screen corresponds to the name of an ABAP Dictionary field, the system automatically establishes a reference between these two fields. Thus, a large number of field attributes for the screen are automatically copied from the ABAP Dictionary. The field attributes together with data element and domain of the assigned Dictionary field form the basis for the standard functions the screen executes in a dialog (automatic format check for screen fields, automatic value range check, online help, and so on).

Error Dialogs

Another task of the screen processor is to conduct error dialogs. Checking the input data is carried out either automatically using check tables of the ABAP Dictionary or by the ABAP program itself. The screen processor includes the error message into the received screen and returns the screen to the user. The message may be context-sensitive, that is, the system replaces placeholders in the message text with current field contents. In addition, only fields whose contents is related to the error and for which a correction may solve the error can accept input. See also Messages on Screens [Page 936].

Data Consistency

To keep data consistent within complex applications, ABAP offers techniques for optimizing database updates that operate independent of the underlying database and correspond to the special requests of dialog programming. See also Programming Database Updates [Page 1260].

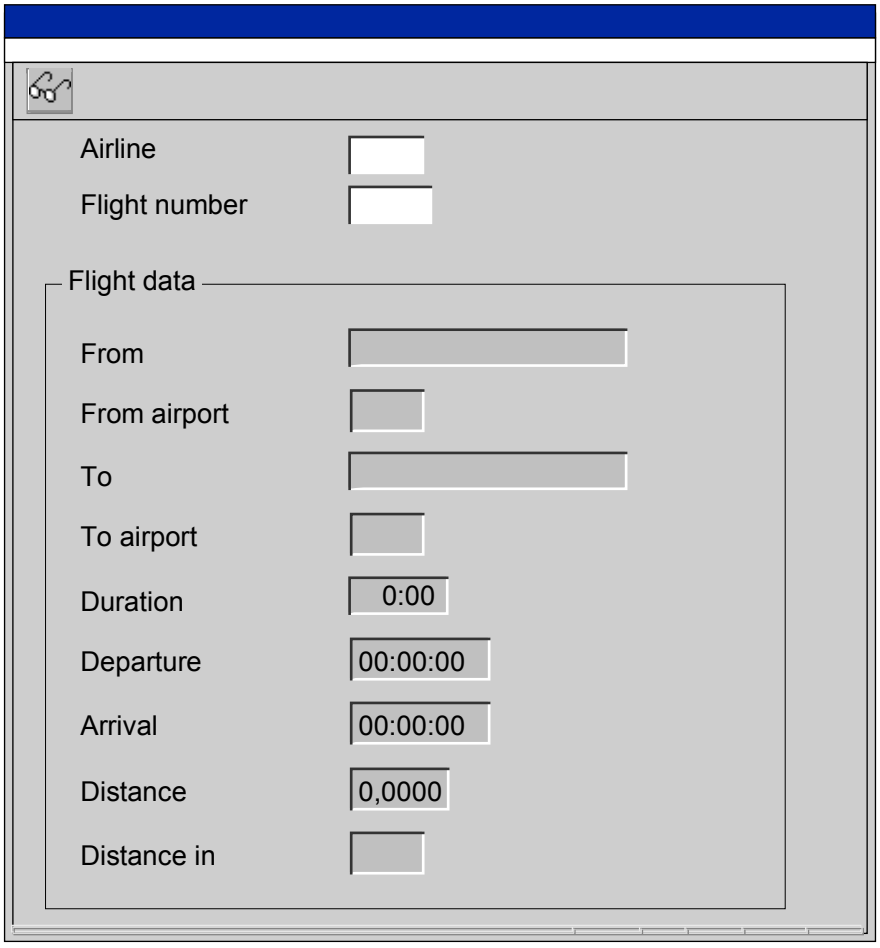
Sample Transaction

Kontextmenü [Page 639]

This example illustrates the concept behind dialog-driven transactions.

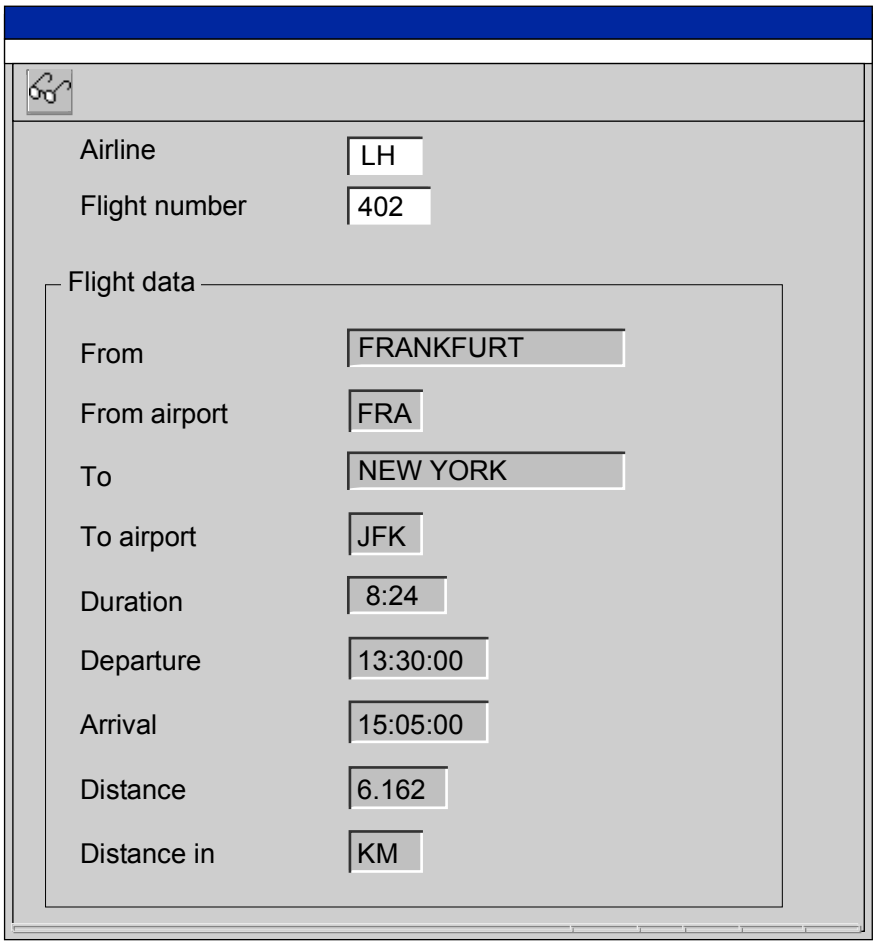
Features

Transaction TZ10 (development class SDWA) is delivered with the system. This transaction consists of a single dialog screen. The user can enter the ID of an airline company and a flight number to request flight information:



If the user chooses *Display*, the system retrieves the requested data from the database and displays it:

Sample Transaction



Structure

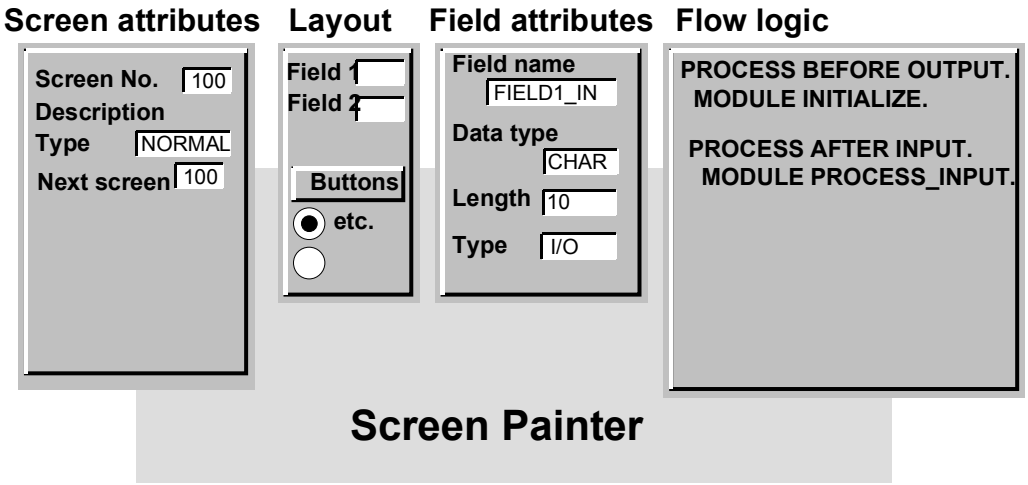
The structure of transaction TZ10 is described below.

All of its components belong to the program SAPMTZ10. You can display them in the Repository Browser. You can also choose *System* → *Status* from within the transaction, and then double-click one of the listed components to switch to the appropriate tool in the ABAP Workbench.

Screen

Each screen contains fields used to display or request information. Fields can be text strings, input or output fields, radio buttons, checkboxes, or pushbuttons. The screen of Transaction TZ10 contains only texts and input/output fields.

A screen consists of several components:



- **Screen attributes:** Contain the screen number, number of the next screen, and other attributes.
- **Screen layout:** Positions of the texts, fields, pushbuttons, and so on for a screen.
- **Field attributes:** Definition of the attributes of the individual fields on a screen.
- **Flow logic**: Calls the ABAP modules for a screen.

You create all of the components of a screen using the Screen Painter [Ext.]. To start the Screen Painter, create a screen in the Repository Browser or double-click an existing one. The Repository Browser calls the Screen Painter, in which you can enter the flow logic of the screen you want to create or change. The flow logic editor also contains pushbuttons that you can use to switch to the layout editor, the field list, or the screen attributes screen.

Screen Attributes

From the user's point of view, a transaction is a sequence of screens, displayed one after another. How is the sequence determined? The transactions's attributes determine the first screen to be displayed. The attributes of the individual screens determine which screen is displayed after the current screen. You can also set the number of the subsequent screen dynamically from within the ABAP program.

For our example, the screen attributes need not be changed, since no subsequent screen is called.

Layout

To start the layout editor in the Screen Painter, choose *Fullscreen*. Here you can determine the layout of the screen. For Transaction TZ10, the desired fields can be copied from table SPFLI of the ABAP Dictionary. For further information, refer to the Screen Painter [Ext.] documentation.

Field Attributes

In the *element list*, you can display or change the attributes of each field on the screen. (Input/output fields, required fields, whether the possible entries button is displayed, whether the

Sample Transaction

field is invisible, and so on.)
The fields *Airline* (SPFLI-CARRID) and *Flight number* (SPFLI-CONNID) are defined as input/output fields. All other fields are used to display the flight data only.

Flow Logic

The flow logic code of a dialog screen consists of a few statements that syntactically resemble ABAP statements. You cannot use flow logic keywords in ABAP programs, or ABAP statements in screen flow logic. The flow logic is a component of the screen. You enter it in the flow logic editor of the Screen Painter.

The flow control for the screen in Transaction TZ10 looks like this:

```
PROCESS BEFORE OUTPUT.
  MODULE SET_STATUS_0100.
*
PROCESS AFTER INPUT
  MODULE USER_COMMAND_0100.
```

The PROCESS statement introduces the flow logic for the two events PBO and PAI. The MODULE statements each call one dialog module in the associated ABAP program. In this example, there is only one MODULE for each event PBO and PAI. However, the flow logic can, of course, contain more statements, calling more than one dialog module in each event. You can also call the same dialog module from more than one screen.

The flow logic syntax contains only a few statements. The most important are MODULE, FIELD, CHAIN, LOOP, and CALL SUBSCREEN. For information about flow logic syntax, choose *Utilities → Help on...* from the flow logic editor. A dialog box appears, in which you select *Flow logic keyword* and then enter the keyword for which you want more information.

ABAP Program

In our example, the ABAP program has type M, that is, it is a module pool. When you create a type M program in the Repository Browser, the ABAP Workbench automatically organizes the program code into a series of include programs. If your ABAP program observes the naming convention SAPM<name>, the hierarchy tree in the Repository Browser allows you to create the following include programs:

- **Global fields**: Global data declarations in the include M<name>TOP. This data is visible in all modules within the program.
- **PBO modules**: Dialog modules in the includes M<name>O<nn>, which are called before a screen is displayed.
- **PAI modules**: Dialog modules in the includes M<name>I<nn>, which are called after user actions on screens.
- **Subroutines**: Subroutines within the program, stored in the includes M<name>F<nn>. These can be called from anywhere in the program.
- **List events:** Event blocks for list processor events, stored in the includes M<name>E<nn>. These occur during list processing.

Include programs can contain several processing blocks. These normally all have the same type (for example, only PBO modules or only PAI modules). However, you could create a separate include program for each processing block, or combine various types of processing block in a single include program.

If you follow the working method suggested by the ABAP Workbench, the source code of your main program is empty apart from a series of INCLUDE statements that incorporate the individual includes into your program:

```
*&---------------------------------------------------------------------*
*& Module pool       SAPMTZ10                                          *
*&                                                                     *
*&---------------------------------------------------------------------*
*&                                                                     *
*&    Display data from table SPFLI                                    *
*&                                                                     *
*&---------------------------------------------------------------------*

* Global data
INCLUDE MTZ10TOP.

* PAI Modules
INCLUDE MTZ10I01.

* PBO Modules
INCLUDE MTZ10O01.
```

In the example program, the include programs look like this:

```
*&---------------------------------------------------------------------*
*& Module pool   SAPMTZ10                                              *
*&      FUNCTION: Display data from Table SPFLI                        *
*&                                                                     *
*&---------------------------------------------------------------------*
*---------------------------------------------------------------------*
*  INCLUDE MTZ10TOP (This is the TOP include:                         *
*      the TOP module contains global data declarations)              *
*---------------------------------------------------------------------*
PROGRAM SAPMTZ10.
  TABLES: SPFLI.

  DATA OK_CODE(4).
*---------------------------------------------------------------------*
*  INCLUDE MTZ10O01 (This is a PBO include.)                          *
*---------------------------------------------------------------------*
*&---------------------------------------------------------------------*
*&   Module STATUS_0100
*&---------------------------------------------------------------------*
*  Specify GUI status and title for screen 100                        *
*---------------------------------------------------------------------*
MODULE STATUS_0100.
 SET PF-STATUS 'TZ0100'.
  SET TITLEBAR '100'.
ENDMODULE.
*---------------------------------------------------------------------*
*  INCLUDE MTZ10I01 (This is a PAI include.)                          *
*---------------------------------------------------------------------*
```

Sample Transaction

```
*&---------------------------------------------------------------------*
*&   Module  USER_COMMAND_0100 INPUT
*&---------------------------------------------------------------------*
*    Retrieve data from SPFLI or leave transaction      *
*----------------------------------------------------------------------*
MODULE USER_COMMAND_0100 INPUT.
  CASE OK_CODE.
    WHEN 'SHOW'.
      CLEAR OK_CODE.
      SELECT SINGLE * FROM SPFLI WHERE CARRID = SPFLI-CARRID
               AND  CONNID = SPFLI-CONNID.
    WHEN SPACE.
    WHEN OTHERS.
      CLEAR OK_CODE.
      SET SCREEN 0
            LEAVE SCREEN.
 ENDCASE.
ENDMODULE.
```

In the include program MTZ10TOP, the TABLES statement creates a table work area for the database table SPFLI. The table work area serves as an interface for passing data between the screen and the ABAP program, since the input/output fields on the screen were created with the same ABAP Dictionary reference. The OK_CODE field is used to receive function codes from the identically-named screen field.

In the include program MTZ10O1, the PBO module for screen 100 sets the GUI status STATUS_0100 and the title 100. This GUI status and title apply to all subsequent screens until you set new ones. It is important that you set a GUI status, since the system otherwise uses an empty status that does not allow the user to leave the program.

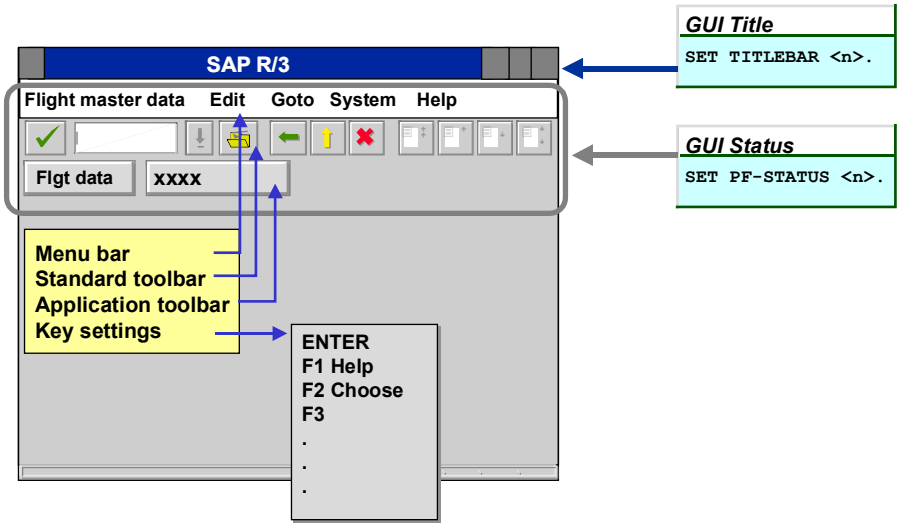
In the include program MTZ10I01, the PAI module USER_COMMAND_0100 checks which pushbutton the user chose (CASE OK_CODE.). The *Display* pushbutton has the function code ‘SHOW’. If the user chooses this function, the program reads the entries from database table SPFLI that correspond to the details entered by the user. In the WHERE condition of the SELECT statement, the system compares the fields SPFLI-CARRID and SPFLI-CONNID (filled in on the screen by the user) with the key fields CARRID and CONNID of database table SPFLI.

Before the screen is next displayed, the data from the table work area is passed to the corresponding screen fields, and therefore appears on the screen.

GUI Status and GUI Title

The GUI status and GUI title are interface elements of screens. You create both of them using the Menu Painter [Ext.] in the ABAP Workbench.

Interface elements



A GUI status is a collection of interactive interface elements for a screen. To apply such a set of elements to a screen, you use the ABAP statement SET PF-STATUS to link the GUI status to the screen. Typically, the GUI status for a screen of a transaction contains all possible functions. Each of these functions send a function code to the PAI modules of the current screen when the user chooses them by choosing a menu entry, pushbutton, or function key.

The GUI title is the screen title displayed in the title bar of the window. The GUI title does not belong to a GUI status - you must set it separately using the ABAP statement SET TITLEBAR.

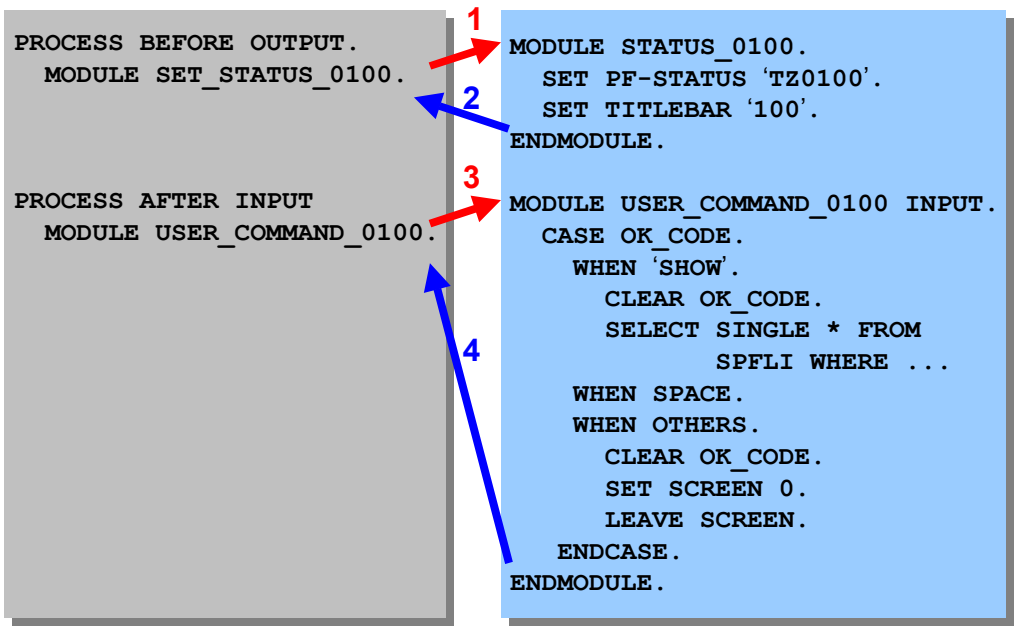
Interaction between Screens and ABAP Programs

In its most simple form, a transaction is a collection of screens and ABAP routines, controlled and executed by the runtime environment. The runtime environment processes the screens in sequence, and calls the corresponding ABAP processing modules.

The runtime environment comprises the screen processor and ABAP processor (see Work Processes [Page 32]). For each screen, the screen processor executes the flow logic, from which the corresponding ABAP processing is called. The control alternates between the screen processor and ABAP processor, and the flow of the application program changes accordingly between screen flow logic and ABAP processing logic. In the same way that we talk about events in ABAP programs, we can also distinguish two events in screen processing - PBO and PAI. The runtime environment (screen processor in this case), triggers the events (for example, PAI after a user action on the screen).

The sequence of events for Transaction TZ10, for example, looks like this:

Sample Transaction



1. In the screen event PBO, the statement MODULE STATUS_0100 calls the corresponding dialog module and passes control to the ABAP processor.
2. After processing the module STATUS_0100, control returns to the flow logic. The screen processor displays the screen.
3. The PAI event is triggered by a user action on the screen. The statement MODULE USER_COMMAND_0100 calls the corresponding dialog module and passes control to the ABAP processor.
4. After processing the module STATUS_0100, control returns to the screen processor. Since the screen has the static next screen 100 (defined in the screen attributes), the program flow begins again at step 1.

Each time control passes between the two processors, the system transfers data between identically-named fields in the program and screen. When control passes from the flow logic to the processing logic, the global ABAP variables are filled with the contents of any identically-named screen fields. Data is transferred in the opposite direction when control passes from the processing logic back to the flow logic. Each screen has a field with type OK that contains the function code of the action that the user chose. To read this value in your ABAP programs, you need a **global** field with the same name in your ABAP program.

Maintaining Transactions

In the ABAP Workbench, you can maintain transaction codes using the Object Navigator or by choosing *Development* → *Other tools* → *Transactions.*

To create a transaction code:

1. Enter a transaction code (up to 20 characters).
2. Choose *Create*.
3. A dialog box appears. Enter a short text and choose the transaction type.

   The transaction type can be as follows:

   Dialog Transaction [Page 996]

   Report Transaction [Page 997]
   Variant Transaction [Page 998]
   Parameter Transaction [Page 999]

Dialog Transactions

In a dialog transaction, the flow of the program is determined by a sequence of screens. The screens that are called within a transaction should belong to a single ABAP program, usually a module pool (type M program).

To create a dialog transaction, use the *Transaction Maintenance* transaction (SE93). Once you have entered a transaction code and short description, choose transaction type *Program and screen (dialog transaction)*.

On the next screen, enter data as required.

The transaction code of a dialog program must be linked to the number of its initial screen. Enter this in the *Screen number* field.

You can also protect the dialog transaction against unauthorized use. To do this, enter the name of an authorization object in the corresponding field. For further information about authorizations in the R/3 System, refer to The SAP Authorization Concept [Ext.].
To enter values for the fields of the authorization object, choose *Values*.

Report Transactions

In a report transaction, you use a transaction code to start an executable program (type 1). An executable program usually has three steps - data entry (selection screen), data processing (often using a logical database), and data output (list).

To create a report transaction, use the *Transaction Maintenance* transaction (SE93). Once you have entered a transaction code and short description, choose transaction type *Program and selection screen (report transaction)*.

When you define a report transaction, you can specify the selection screen and variant with which you want it to start.

You can also protect the report transaction against unauthorized use. To do this, enter the name of an authorization object in the corresponding field. For further information about authorizations in the R/3 System, refer to The SAP Authorization Concept [Ext.].
To enter values for the fields of the authorization object, choose *Values*.

Variant Transactions

In the SAP Reference IMG, you can create transaction variants. Choose *Basis Components* → *Application Personalization* → *Tailoring of Application Transactions* → *Configure Transaction-Related Display Values for Fields* (Transaction SHD0). Transaction variants allow you to preset values for fields in a transaction, set field attributes, or hide entire screens.

To execute a transaction variant, you define a variant transaction using *the Transaction Maintenance* transaction (SE93).

Once you have entered a transaction code and short description, choose transaction type *Transaction with variant (Variant transaction)*.

To define a variant, enter the name of the transaction and the name of the variant. You can then use the new transaction code to start the special variant of the transaction.

Parameter Transaction

Parameter transactions allow you to preassign values to fields on the initial screen of a transaction.

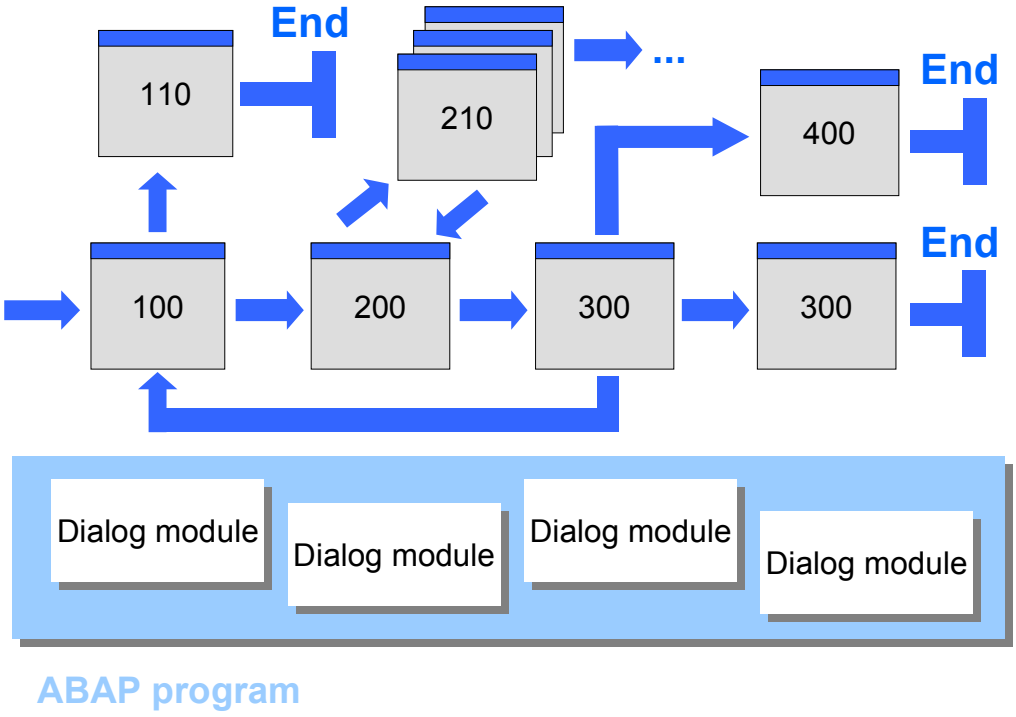
To create a parameter transaction, use the *Transaction Maintenance* transaction (SE93). Once you have entered a transaction code and short description, choose transaction type *Transaction with parameters (Parameter transaction)*.

You can hide the initial screen of a parameter transaction if you have specified values for all of its fields.

Screen Sequences

For the user, an application program consists of a series of screens that are displayed one after the other. The major difference between the program flow of an executable program and a dialog program is that in a dialog program, you can program screens to appear in any sequence you want. In executable programs, the screen sequence is controlled by events, which occur in a fixed order. In a dialog program, the programmer is free to program any sequence of screens, and the user can affect the program flow by his or her actions. However, it is still possible to call a freely-defined screen sequence within an executable program and thus to branch into a form of dialog program.

Screens always belong to an ABAP program. The dialog modules in an ABAP program can only be called from screens in the same program. The system field SY-DYNNR always contains the number of the current screen. You can create any number of complex screen sequences from the screens in a **single program**. For example, you can navigate from one screen to any other screen in the same program, or repeat part of a sequence any number of times.



To start a screen sequence, you have to call its first screen. You can do this either by starting a transaction (the first screen is contained in the transaction definition), or by using the CALL SCREEN statement in an ABAP program. When you call a screen sequence using the CALL SCREEN statement, you nest it within the screen sequence that was already running at the time.

The actual sequence of screens is defined by setting the next screen for each screen in the chain. The attributes of **every** screen in the Screen Painter contain a statically-defined next screen. This forms a **static screen sequence**. However, ABAP allows you to overwrite the statically-defined next screen within a processing block, allowing you to define dynamic screen sequences. These may depend on user interaction with the program.

You can also run a screen sequence without displaying all of the screens. The SUPPRESS DIALOG statement allows you to prevent a screen from being displayed. You include it in a PBO module of the screen concerned. When you use SUPPRESS DIALOG, the entire PBO and PAI logic is processed, even though the screen itself is not displayed. Suppressing a screen is useful when you need to display a list within a screen sequence (refer to Switching Between Screens and Lists [Page 895]).

Screen sequences always end when they reach the next screen number zero. For this reason you cannot create screens with number 0. Instead, you use 0 to terminate a screen sequence. When a screen sequence ends, the system returns to the point from which the first screen in the sequence was called. If you started the screen sequence using a transaction code, this is the point from which you started the transaction. If you started it using CALL SCREEN, it is the point in the ABAP program where the statement occurred.

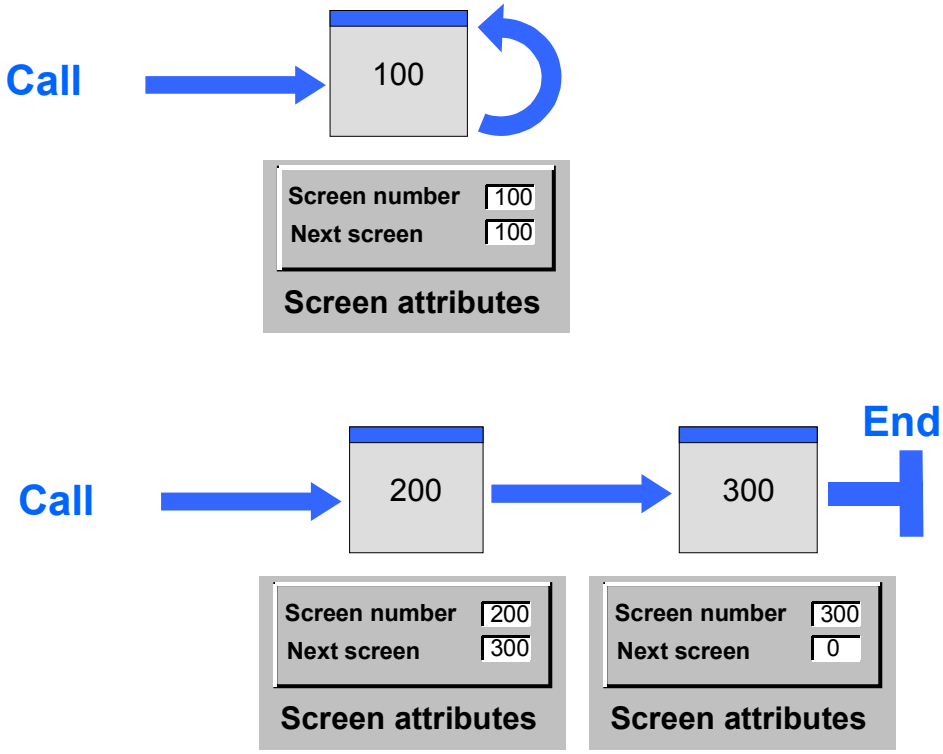
The following sections describe in more detail how to define screen sequences.

Static Next Screen

All screens have a *Next screen* attribute, containing the number of the screen that is statically-called after the current screen. However, the static attribute is always bypassed if you specify a next screen dynamically [Page 1004] while the screen is being processed.

The default next screen in the Screen Painter is always the number of the screen itself. If you use the default value and do not overwrite it dynamically, the screen always calls itself recursively (see the example transaction [Page 987]), and the screen sequence consists of a single screen.

If the next screen specified is zero (or no value at all), and you do not override it dynamically, the current screen is the **last in the screen sequence**. After the screen has been processed, control returns to the point from which the screen sequence was called. If the screen sequence was embedded [Page 1007], the system returns to the previous screen sequence containing the relevant CALL SCREEN statement. If the screen sequence was not embedded, the application program terminates.



Statically-defined screen sequence

Dynamic Next Screen

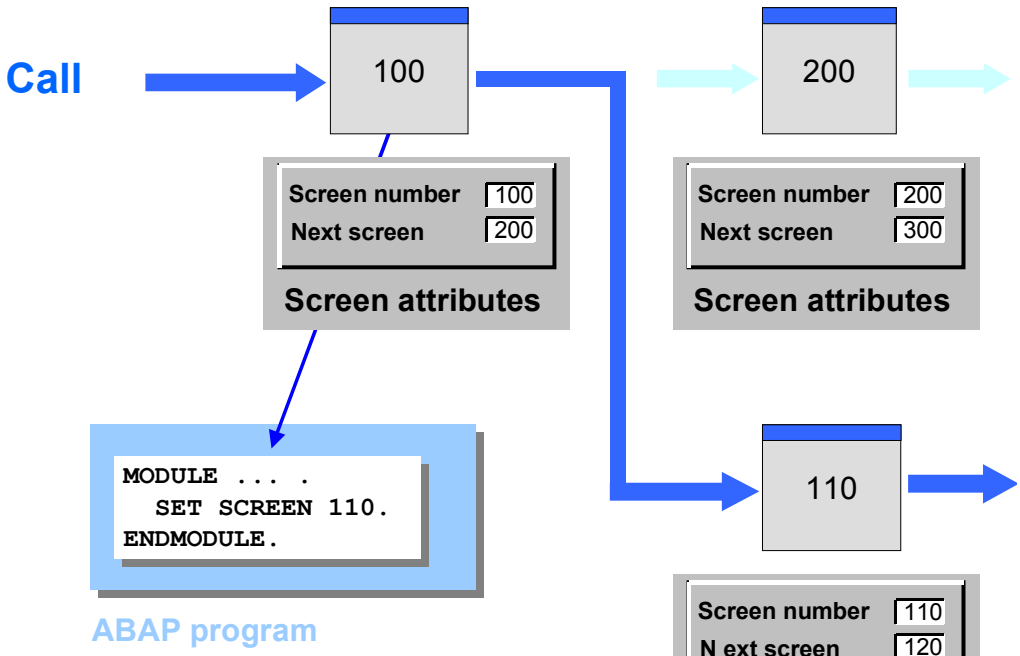
Every screen has a static next screen [Page 1002] attribute that specifies the next screen to be called as long as it is not overwritten dynamically. In the processing logic, that is, a dialog module called by the screen, you can use the SET SCREEN statement to overwrite the statically-defined next screen dynamically:

SET SCREEN <next screen>.

This statement defines a new next screen for the current program run. You can also specify the number of the next screen as a field containing a screen number. The statically-defined next screen is ignored. However, this only overrides the static screen sequence temporarily. The static value for the next screen, as defined in the Screen Painter, is always retained.

If you specify the value 0 for <next screen>, the current screen becomes the **last in the screen chain**. After the screen has been processed, control returns to the point from which the screen sequence was called. If the screen sequence was embedded [Page 1007], the system returns to the previous screen sequence containing the relevant CALL SCREEN statement. If the screen sequence was not embedded, the application program terminates.

The SET SCREEN **does not** interrupt the current screen processing. To leave a screen, use the ABAP statement LEAVE [Page 1006].



Dynamic screen sequencing

Leaving a Screen from a Program

In a program, you can use one of the two following ABAP statements to leave a screen:

LEAVE SCREEN.

or

LEAVE TO SCREEN <next screen>.

The LEAVE SCREEN statement ends the current screen and calls the subsequent screen. The next screen is either the static next screen [Page 1002] or a dynamic next screen [Page 1004]. In the second case, you must override the static next screen using the SET SCREEN statement before the LEAVE SCREEN statement.

The LEAVE TO SCREEN statement exits the current screen and calls the dynamic next screen, which you specify as part of the statement. The LEAVE TO SCREEN statement is no more than a contraction of the two statements

SET SCREEN <next screen>.
LEAVE SCREEN.

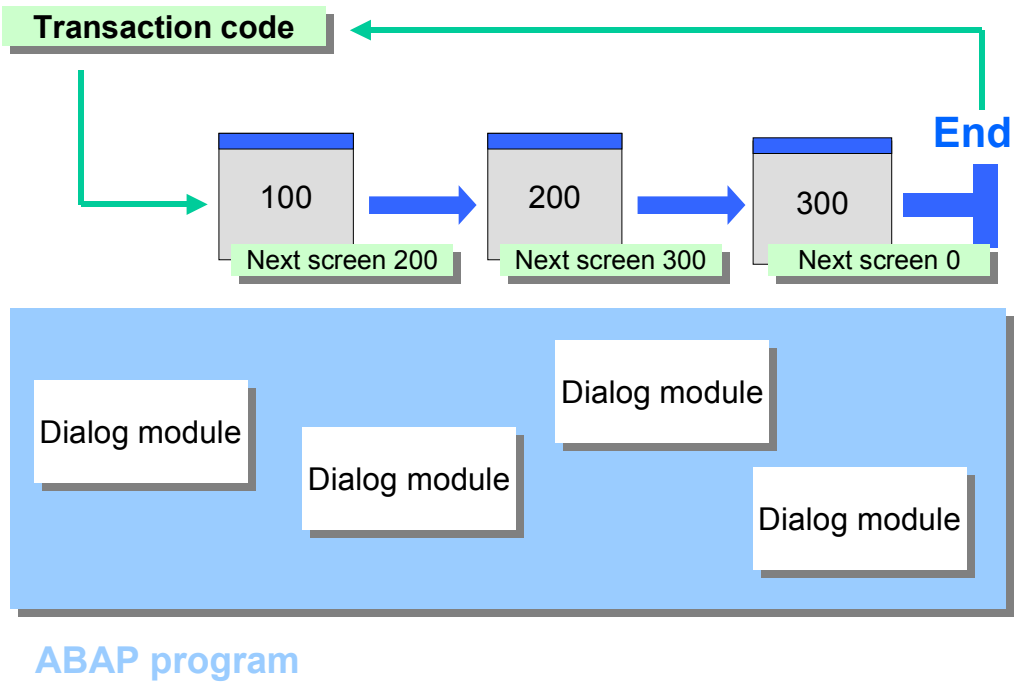
These statements **do not** end the screen sequence. They merely branch to another screen in the same sequence. The screen sequence only ends when you leave to next screen 0.

Starting a Screen Sequence

There are two ways of calling a sequence of screens. Starting a sequence from an ABAP program allows you to insert a sequence of screens into the existing program flow.

Using a Transaction Code

When you use a transaction code to start a screen sequence, the corresponding ABAP program is automatically loaded as well. The processing logic of the program is controlled by the screen flow logic. On reaching the end of the screen sequence (next screen 0), the entire program terminates, and control returns to the point from which the transaction was called.

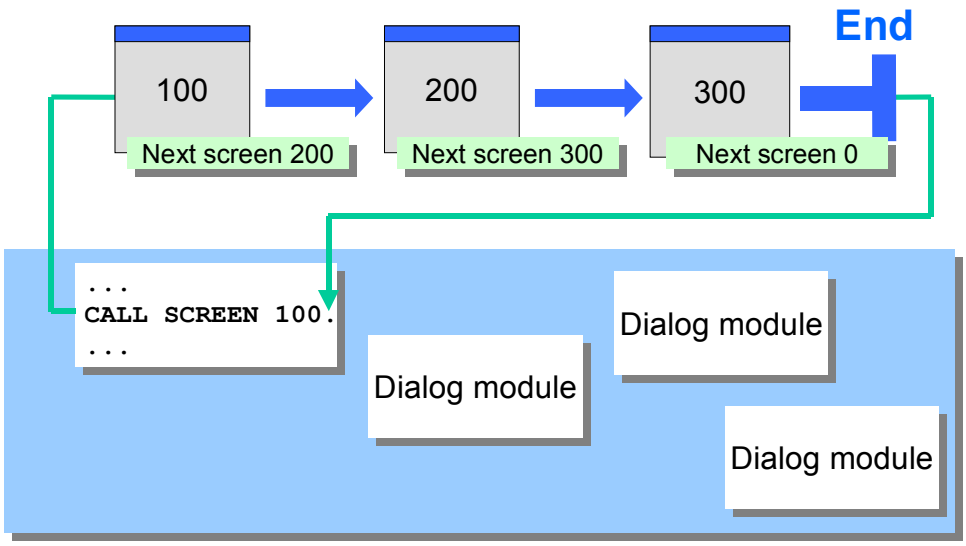


From an ABAP Program

You can start a screen sequence from an ABAP program using the

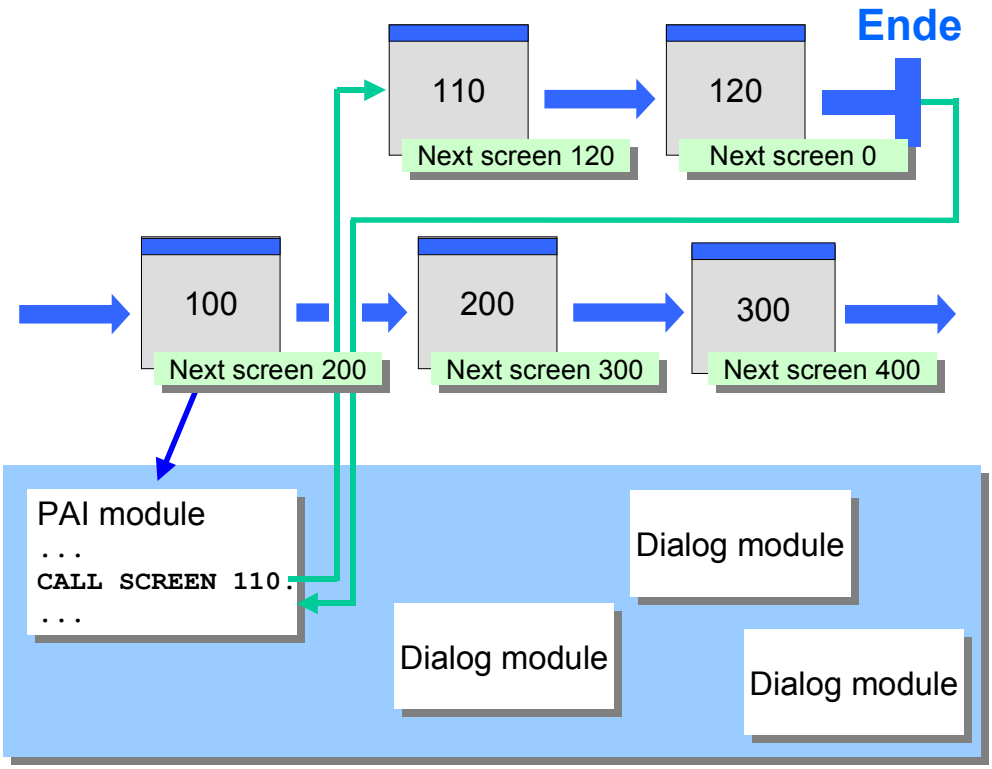
CALL SCREEN <dynnr>.

statement. At the end of the screen sequence (next screen 0), the program continues processing directly after the CALL SCREEN statement.



Inserting Screen Sequences

If you use the CALL SCREEN statement within a screen sequence that is already running, the new screen sequence is inserted into the current sequence.



This works as a kind of stack, since the CALL SCREEN statement interrupts the current screen sequence and starts a new one. When you start a screen sequence with a transaction code, you can stack up to 50 other screen sequences on top of it. However, you should not exceed 40 sequences, since help and error dialogs also use internally-implemented screen sequences.

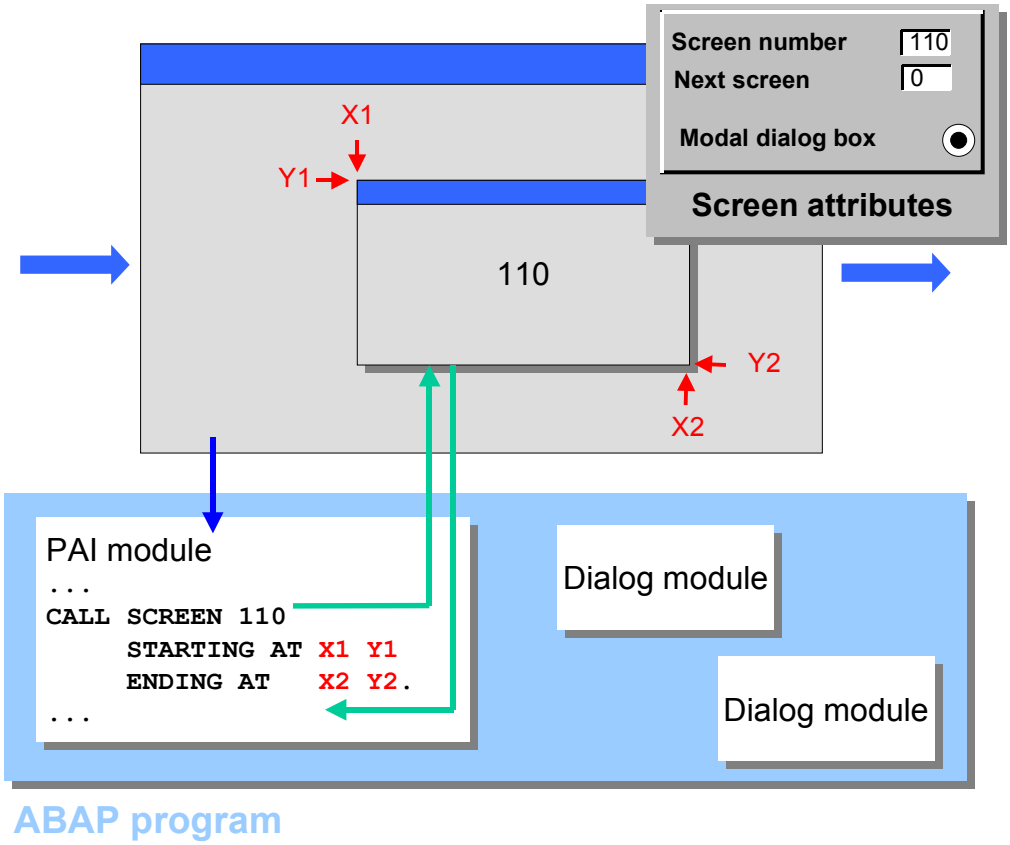
Since during an R/3 terminal session you are always in a screen sequence of some kind, each CALL SCREEN statement leads to one screen sequence being inserted into another. For example, if you use the statement in an executable program, the screen sequence is normally inserted into the normal screen sequence for executable programs, that is, selection screen → list.

Calling Modal Dialog Boxes

Calling a single screen is a special case of embedding a screen sequence [Page 1007]. If you want to prevent the called screen from covering the current screen completely, you can use the CALL SCREEN statement with the STARTING AT and ENDING AT options:

```
CALL SCREEN <scrn>
    STARTING AT <X1> <Y1>
    ENDING AT   <X2> <Y2>.
```

This calls the screen number <scrn> as a modal dialog box. When the screen is displayed, the screen that had called it is visible but inactive. The STARTING AT and ENDING AT additions specify the top left-hand and bottom right-hand corners of the dialog box respectively. In the screen attributes of screen <scrn>, you must set the *Modal dialog box* attribute. This defines how the interface elements are positioned on the screen.

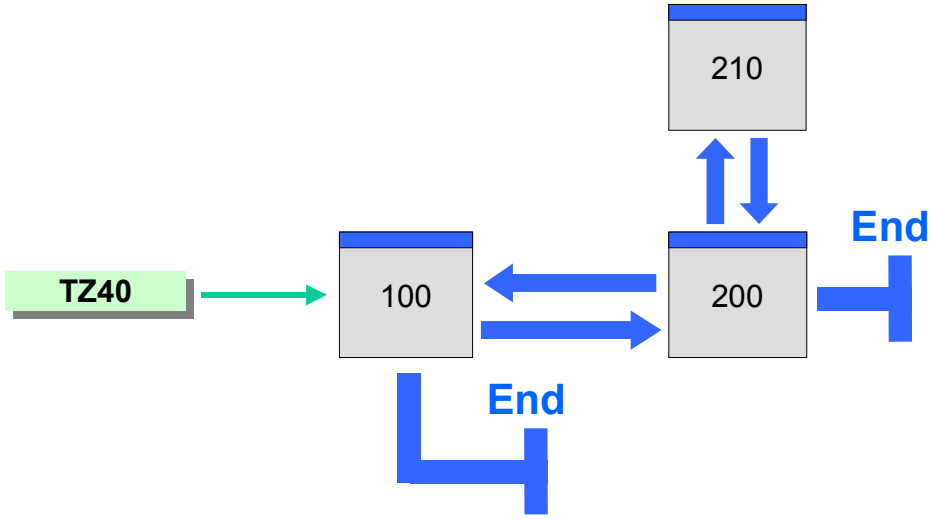


Screen Sequences: Example Transaction

Transaction TZ40 is an example of screen sequences. This transaction is in development class SDWA and is delivered with the system. TZ40 lets users display and change flight data.

Features

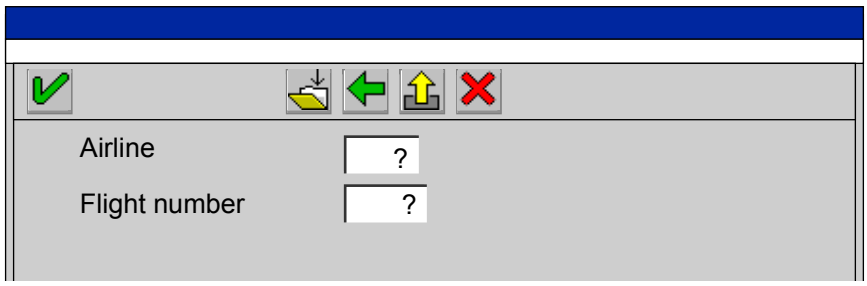
TZ40 uses three screens. Screens 100 and 200 form a sequence, while screen 210 is a modal dialog box and is only called under certain conditions. The possible screen sequences can be represented as follows:



The user working in the system will see the following sequence:

- **Screen 100:**

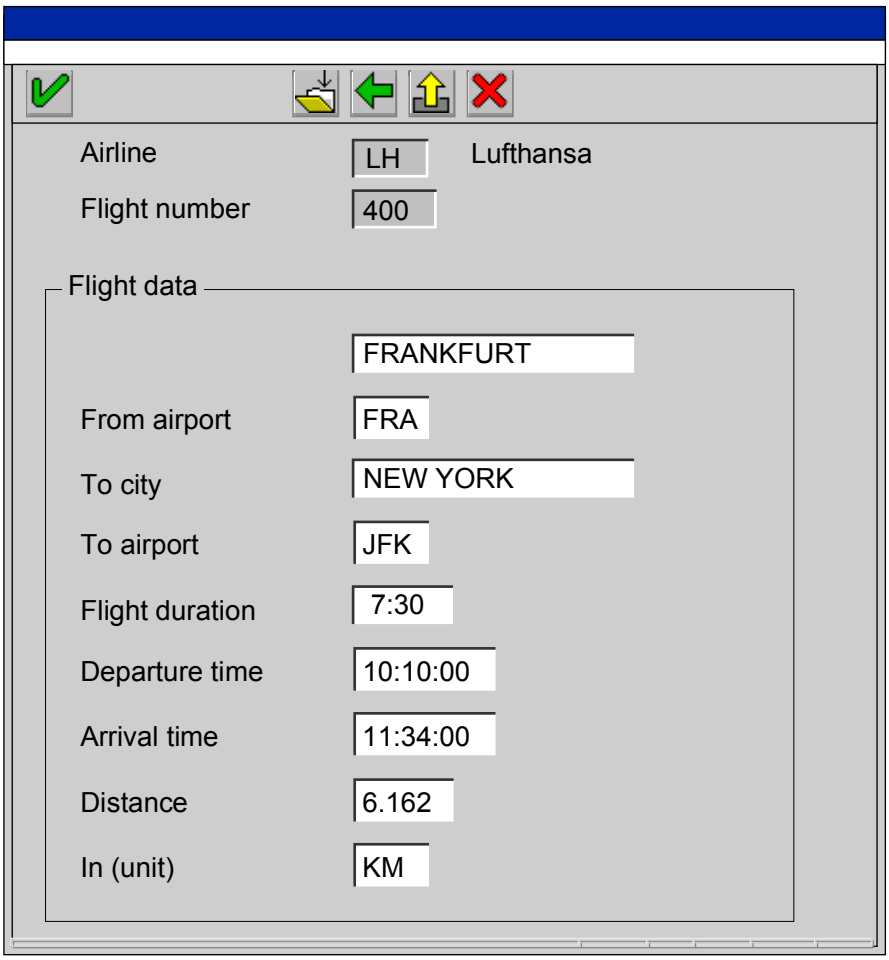
  The user enters an airline and flight number, and either presses ENTER to request the flight data or ends the transaction.



- **Screen 200:**

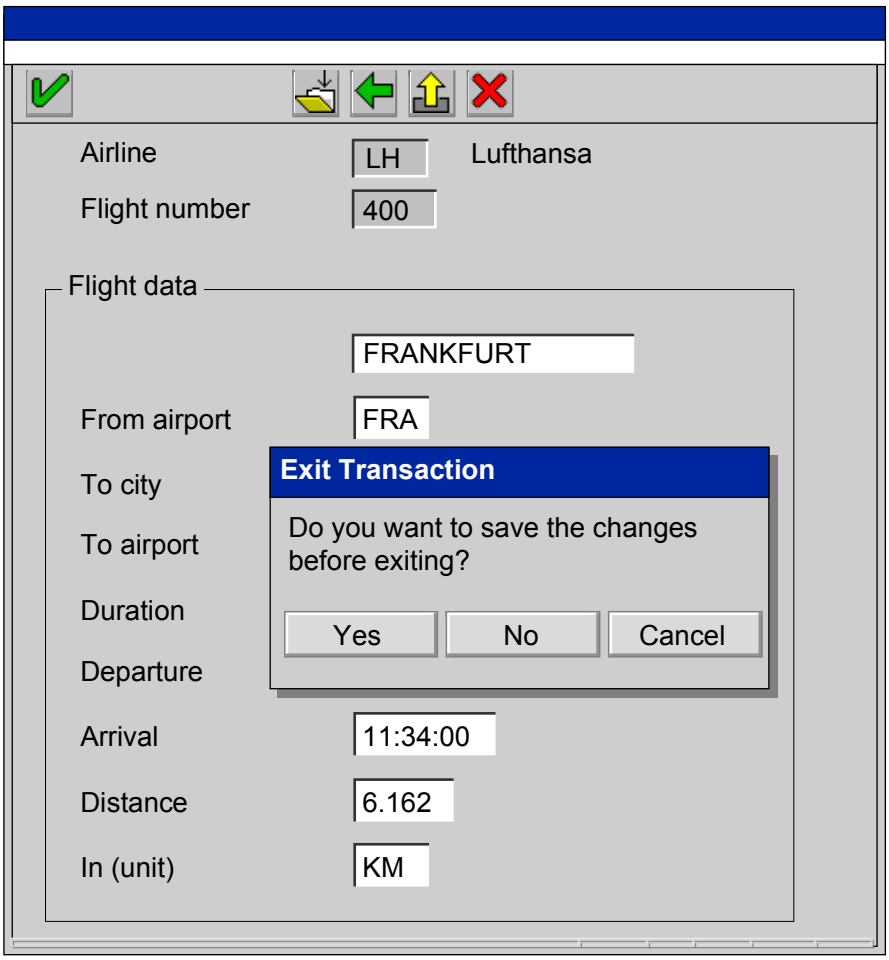
  The system displays complete details about the flight, in input fields. The user types over the display to enter the changes.

Screen Sequences: Example Transaction



- **Screen 210:**

  The modal dialog box is only displayed if you try to leave screen 200 using *Back* or *Exit* without saving your changes. It allows you either to save or cancel the changes.



This transaction provides a good example of how screen sequences are implemented. Let us look at screen 200 to see how the modal dialog box is called.

Screen Flow Logic

When handling a BACK or EXIT function code, the PAI module must check whether flight details have changed since the screen display or last save. If this is the case, screen 210 is called as a modal dialog box. The following parts of the flow logic of screen 200 are as follows:

```
*---------------------------------------------------------------*
*   Screen 200: Flow logic                                      *
*&---------------------------------------------------------------*
PROCESS AFTER INPUT.
  MODULE EXIT_0200 AT EXIT-COMMAND.
  ...
  MODULE USER_COMMAND_0200.
```

Two dialog modules are called in the PAI event.

Screen Sequences: Example Transaction

ABAP Coding

The user interface of Transaction TZ40 has three functions for leaving a screen - *Back*, *Exit*, and *Cancel*. On screen 200, the user should only be able to leave the screen directly using *Cancel*. The function code is processed in the module EXIT_200. The next screen is set dynamically to 100, and screen 200 is terminated immediately with the LEAVE SCREEN statement.

```
*&---------------------------------------------------------------------*
*&      Module  EXIT_0200  INPUT
*&---------------------------------------------------------------------*
MODULE EXIT_0200 INPUT.
  CASE OK_CODE.
    WHEN 'CANC'.
      CLEAR OK_CODE.
      SET SCREEN 100. LEAVE SCREEN.
  ENDCASE.
ENDMODULE.
```

The remaining function codes for screen 200 are processed in the module USER_COMMAND_200.

- The SAVE function triggers an update of the database.
- The EXIT and BACK functions trigger calls to the SAFETY_CHECK routine. This subroutine checks to see whether there is unsaved data on the screen, and, if required, calls screen 210.

```
*&---------------------------------------------------------------------*
*&      Module   USER_COMMAND_0200  INPUT
*&---------------------------------------------------------------------*
MODULE USER_COMMAND_0200 INPUT.
  CASE OK_CODE.
    WHEN 'SAVE'.
      UPDATE SPFLI.
      IF SY-SUBRC = 0.
        MESSAGE S001 WITH SPFLI-CARRID SPFLI-CONNID.
      ELSE.
        MESSAGE A002 WITH SPFLI-CARRID SPFLI-CONNID.
      ENDIF.
      CLEAR OK_CODE.
    WHEN 'EXIT'.
      CLEAR OK_CODE.
      PERFORM SAFETY_CHECK USING RCODE.
      IF RCODE = 'EXIT'. SET SCREEN 0. LEAVE SCREEN. ENDIF.
    WHEN 'BACK'.
      CLEAR OK_CODE.
      PERFORM SAFETY_CHECK USING RCODE.
      IF RCODE = 'EXIT'. SET SCREEN 100. LEAVE SCREEN. ENDIF.
  ENDCASE.
ENDMODULE.
```

If the user chooses *Exit* (function code EXIT), the screen sequence is terminated dynamically using SET SCREEN 0, and the transaction ends. If the user chooses *Back* (function code BACK), the next screen is changed dynamically to 100 using the statement SET SCREEN 100.

The subroutine SAFETY_CHECK checks the current screen values against the old values. If the values match, no save is needed, and the routine terminates.

```
*---------------------------------------------------------------*
*   Subroutine SAFETY_CHECK                                     *
*---------------------------------------------------------------*
FORM SAFETY_CHECK USING RCODE.
  LOCAL OK_CODE.
  RCODE = 'EXIT'.
  CHECK SPFLI NE OLD_SPFLI.
  CLEAR OK_CODE.
  CALL SCREEN 210 STARTING AT 10 5.
  CASE OK_CODE.
    WHEN 'SAVE'. UPDATE SPFLI.
    WHEN 'EXIT'.
    WHEN 'CANC'. CLEAR SPFLI.
  ENDCASE.
ENDFORM.
```

If the values differ, SAFETY_CHECK calls the modal dialog box 210. This asks the user if he wants to save, and returns the answer (SAVE, EXIT and CANC) to the field OK_CODE. The static next screen for screen 210 is 210. However, the processing logic (module USER_COMMAND_210) always sets the next screen dynamically to 0, which returns control to the subroutine.

Calling Programs

If you need to program an extensive application, one single program will become very complex. To make the program easier to read, it is often reasonable to divide the required functions among several programs.

As well as external modularization [Page 441], in which you store procedures in special non-executable ABAP programs like function groups, you can also call independent programs from within an ABAP program.

The following ABAP statements allow you to start an executable program or transaction. You can either exit the calling program, or have the system return to it when the called program finishes running.

	Type 1 Program	Transaction
Call (without returning)	SUBMIT	LEAVE TO TRANSACTION
Call and return	SUBMIT AND RETURN	CALL TRANSACTION

You can use these statements in any ABAP program. For example, while processing a user action in the output list of an executable program, you might call a transaction whose initial screen is filled with data from the selected list line.

An interesting remark at this point is that **every time** you run an executable program, a SUBMIT statement occurs. When you enter the program name in a transaction like SE38 or SA38 and choose *Execute*, a SUBMIT statement occurs in the transaction. It is therefore a technical attribute of a type 1 program that they are called using SUBMIT, although their principal characteristic from a user's point of view is that they are started in the foreground.

Memory Organization in Program Calls

The first ABAP program in a session on the application server opens its own internal session (roll area) within the main session. All externally-called procedures [Page 449] run in the same internal session as the calling program, that is, the main program and working data of the procedure are loaded into the same memory area in the internal session.

When you call an executable program or a transaction, the system opens a new internal session for each program. Here, there are two possible cases: If the second program does not return control to the calling program when it has finished running, the called program replaces the calling program in the internal session. The contents of the memory of the calling program are deleted. If the second program does return control to the calling program when it has finished running, the session of the called program is not deleted. Instead, it becomes inactive, and its memory contents are placed on a stack. The system can open up to 9 further internal sessions in external program calls. See also Memory Structures of an ABAP Program [Page 66].

As well as executable programs and transactions, dialog modules [Page 1028] also open a new internal session. Dialog modules were previously used for modularizing screen sequences.

Program Calls and SAP LUWs

An SAP LUW [Page 1265] is a logical unit consisting of dialog steps, whose changes are written to the database in a single database LUW. There are various bundling techniques that you can use to ensure that all of the database updates belonging to an SAP LUW are made in the same single database LUW.

Externally-called procedures [Page 449] do not open a new SAP LUW.

However, when you start a new executable program or transaction, a new SAP LUW starts. Database updates belonging to these programs are collected in their own database LUW. If the new program does not return control to the calling program, the SAP LUW of the old program concludes when the new program is called. If, on the other hand, the new program does return control to the calling program, the new SAP LUW runs parallel to the SAP LUW of the calling program.

No new SAP LUW is opened when you call a dialog module. Bundling techniques in a dialog module add the database updates to the database LUW of the calling program. You may sometimes need to call a transaction that runs in the same SAP LUW as the calling program. One technique for doing this is to use the existing transaction as a dialog module. To do this, you need to create a new dialog module with the same main program and initial screen as the transaction. Transactions that are used both as transactions and as dialog modules must be programmed to obey certain rules. For further information, refer to Calling Screen Sequences [Page 1028].

The fact that an external program shares (or doesn't share) the SAP LUW with its caller has special consequences if the program calls update-task functions or uses COMMIT WORK. For further information, refer to Special LUW Considerations [Page 1286].

Calling Executable Programs

You can call executable programs from other ABAP programs using the following statement:

SUBMIT <rep>|(<field>) [AND RETURN] [<options>].

You can either specify the name of the program you want to call statically by entering the program name in the code of the calling program, or dynamically by specifying the name of a field (in parentheses) containing the name of the program. If the system cannot find the specified executable program when trying to execute the SUBMIT statement, a runtime error occurs.

If you omit the AND RETURN addition, all data and list levels of the calling program (the entire internal session) are deleted. After the called executable program has finished, control returns to the level from which you started the calling program.

If you use AND RETURN, the system stores the data of the calling executable program and returns to the calling after processing the called program. The system resumes executing the calling program at the statement following the call.

The SUBMIT statement has a set of additions <options> for passing data to the called program and specifying various other processing options. Some of them are described in the following sections:

Filling the Selection Screen of a Called Program [Page 1019]

Affecting Lists in Called Programs [Page 1023]

Program Statements to Leave a Called Program [Page 1025].

Filling the Selection Screen of a Called Program

When you start an executable program, the standard selection screen normally appears, containing the selection criteria and parameters of both the logical database connected to the program and of the program itself (see Direct Execution - Reports [Page 945]). When you start an executable program using SUBMIT, there are various additions that you can use to fill the input fields on the selection screen:

```
SUBMIT... [VIA SELECTION-SCREEN]
          [USING SELECTION-SET <var>]
          [WITH <sel> <criterion>]
          [WITH FREE SELECTIONS <freesel>]
          [WITH SELECTION-TABLE <rspar>].
```

These options have the following effects:

- VIA SELECTION-SCREEN

  The selection screen of the called executable program (report) appears. If you transfer values to the program using one or more of the other options, the corresponding input fields in the selections screen are filled. The user can change these values. By default, the system does not display a selection screen after SUBMIT.

- USING SELECTION-SET <var>

  This option tells the system to start the called program with the variant <var>.

- WITH <sel> <criterion>

  Use this option to fill individual elements <sel> of the selection screen (selection tables and parameters). Use one of the elements <criterion>:

– <op> <f> [SIGN <s>], for single value selection

  If <sel> is a selection criterion, use <op> to fill the OPTION field, <f> to fill the LOW field, and <s> to fill the SIGN field of the selection table <sel> in the called program.

  If <sel> is a parameter, you can use any operator for <op>. The parameter <sel> is always filled with <f>.

– [NOT] BETWEEN $<f_1>$ AND $<f_2>$ [SIGN <s>], for interval selection

  $<f_1>$ is transferred into the LOW field, $<f_2>$ into the HIGH field, and <s> into the SIGN field of the selection table <sel> in the called program. If you omit the NOT option, the system places the value BT into the OPTION field; if you use NOT, the system fills OPTION with NB.

– IN <seltab>, transferring a selection table

  This addition fills the selection table <sel> in the called program with the values of the table <seltab> in the calling program. Table <seltab> must have the structure of a selection table. Use the RANGES statement to create selection tables.

- WITH FREE SELECTION <freesel>, user dialog for dynamic selections

  To use this option, both calling and called programs must be connected to a logical database that supports dynamic selections. In the calling program, use the function modules FREE_SELECTIONS_INIT and FREE_SELECTIONS_DIALOG. They allow the user to enter dynamic selections on a selection screen. One export parameter of these

function modules has structure RSDS_TEXPR from the RSDS type group. Transfer the values of this export parameter by means of the internal table <freesel> of the same structure to the called report.

- WITH SELECTION-TABLE <rspar>, dynamic transfer of values

  You need an internal table <rspar> with the Dictionary structure RSPARAMS. The table then consists of the following six fields:

– SELNAME (type C, length 8) for the name of the selection criterion or parameter

– KIND (type C, length 1) for the selection type (S for selection criterion, P for parameter)

– SIGN, OPTION, LOW, HIGH as in a normal selection table, except that LOW and HIGH both have type C and length 45.

  This table can be filled dynamically in the calling program with all of the required values for the selection screen of the called program. If the name of a selection criterion appears more than once, the system creates a multiple-line selection table for that criterion in the called program. If the name of a parameter appears more than once, the system uses the last value. Note that LOW and HIGH have type C, so that the system executes type conversions to the criteria of the called program. This is important for date fields, for example. Before your program is used in a live context, you should check it using the VIA SELECTION-SCREEN addition.

Except for WITH SELECTION-TABLE, you can use any of the above options several times and in any combination within a SUBMIT statement. In particular, you can use the WITH <sel> option several times for one single criterion <sel>. In the called program, the system appends the corresponding lines to the selection tables used. For parameters, it uses the last value specified. The only combination possible for the WITH SELECTION-TABLE option is USING SELECTION-SET.

If the input fields on the selection screen are linked to SPA/GPA parameters, you can also use this technique to pass values to the selection screen (see Passing Data Between Programs [Page 1032]).

The following executable program (report) creates a selection screen containing the parameter PARAMET and the selection criterion SELECTO:

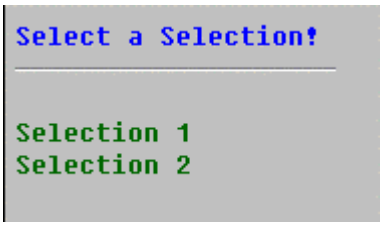
```
REPORT REP1.
DATA NUMBER TYPE I.
PARAMETERS     PARAMET(14).
SELECT-OPTIONS  SELECTO FOR NUMBER.
```

The program REP1 is called by the following program using various parameters:

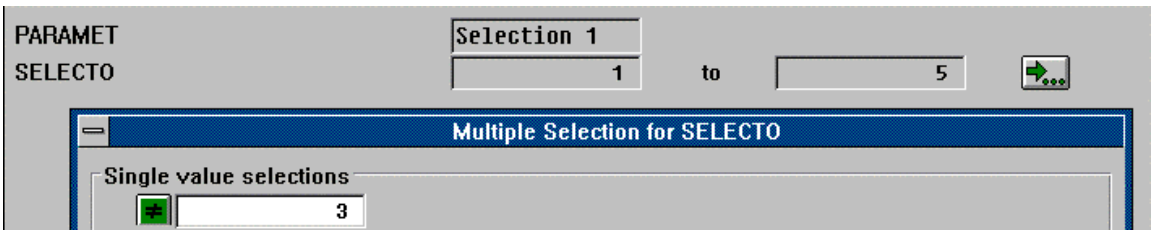
```
REPORT REP2 NO STANDARD PAGE HEADING.

DATA: INT TYPE I,
      RSPAR LIKE RSPARAMS OCCURS 10 WITH HEADER LINE.

RANGES SELTAB FOR INT.

WRITE: 'Select a Selection!',
     / '--------------------'.
SKIP.
```

```
FORMAT HOTSPOT COLOR 5 INVERSE ON.
WRITE: 'Selection 1',
    / 'Selection 2'.

AT LINE-SELECTION.
  CASE SY-LILLI.
    WHEN 4.
      SELTAB-SIGN = 'I'. SELTAB-OPTION = 'BT'.
      SELTAB-LOW  = 1.   SELTAB-HIGH   = 5.
      APPEND SELTAB.
      SUBMIT REP1 VIA SELECTION-SCREEN
                  WITH PARAMET EQ 'Selection 1'
                  WITH SELECTO IN SELTAB
                  WITH SELECTO NE 3
                  AND RETURN.
    WHEN 5.
      RSPAR-SELNAME = 'SELECTO'. RSPAR-KIND = 'S'.
      RSPAR-SIGN = 'E'. RSPAR-OPTION = 'BT'.
      RSPAR-LOW  = 14.  RSPAR-HIGH = 17.
      APPEND RSPAR.
      RSPAR-SELNAME = 'PARAMET'. RSPAR-KIND = 'P'.
      RSPAR-LOW  = 'Selection 2'.
      APPEND RSPAR.
      RSPAR-SELNAME = 'SELECTO'. RSPAR-KIND = 'S'.
      RSPAR-SIGN = 'I'. RSPAR-OPTION = 'GT'.
      RSPAR-LOW  = 10.
      APPEND RSPAR.
      SUBMIT REP1 VIA SELECTION-SCREEN
                  WITH SELECTION-TABLE RSPAR
                  AND RETURN.
  ENDCASE.
```

When you start the program, the following basic list appears:

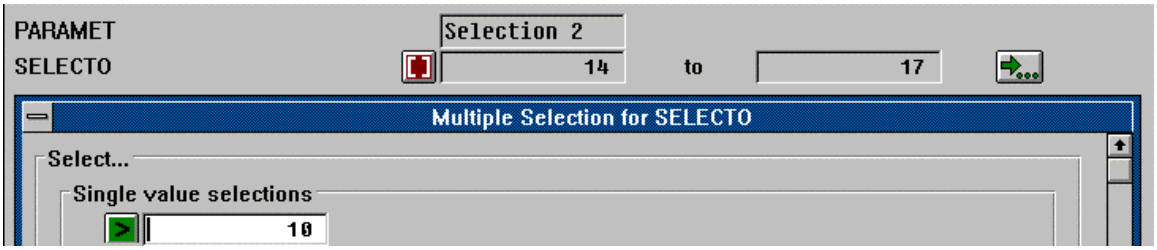


After clicking once on the first hotspot, the selection screen of SAPMZTS1 looks like this:



After clicking once on the second hotspot, the selection screen of SAPMZTS1 looks like this:

Filling the Selection Screen of a Called Program



For both calls of REP1, the system transfers values that lead to two-line selection tables SELECTO. The second line appears in the respective dialog box *Multiple Selection for SELECTO*. Without the VIA SELECTION-SCREEN option of the SUBMIT statement, PARAMET and SELECTO would be filled accordingly in REP1, but they would not be displayed.

Affecting Lists in Called Programs

When you call an ABAP program, you can modify its lists, send them to a spool file instead of the screen, or store them in ABAP memory.

Modifying the List Structure

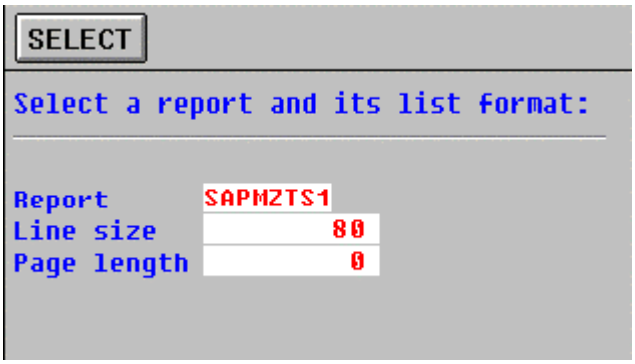
You can modify the list structure of a called program by using the following additions in the SUBMIT statement:

SUBMIT... [LINE-SIZE <width>] [LINE-COUNT <length>].

If the called program contains **no** such options in the REPORT statement, the system formats the lists of the called program according to the options in the SUBMIT statement. If, on the other hand, the REPORT statement in the called program does specify a list structure, the additions in the SUBMIT statement are ignored. For further information about these additions, refer to Defining Your Own List Structure [Page 795].

```
REPORT STARTER NO STANDARD PAGE HEADING.

DATA: NAME(8) VALUE 'SAPMZTS1',
            WID TYPE I VALUE 80,
            LEN TYPE I VALUE 0.

SET PF-STATUS 'SELECT'.

WRITE: 'Select a report and its list format:',
    / '-------------------------------------'.
SKIP.

WRITE: 'Report     ', NAME INPUT ON,
    / 'Line size  ', WID  INPUT ON,
    / 'Page length', LEN  INPUT ON.

AT USER-COMMAND.

  CASE SY-UCOMM.
    WHEN 'SELE'.
      READ LINE: 4 FIELD VALUE NAME,
                 5 FIELD VALUE WID,
                 6 FIELD VALUE LEN.
      SUBMIT (NAME) LINE-SIZE WID LINE-COUNT LEN AND RETURN.
  ENDCASE.
```

You can use this program to start programs with user-defined list formatting. On the basic list, the user can enter a report name and the desired list width and length by overwriting the default values:



In the AT USER-COMMAND event, the system reads the values and starts the specified executable program using SUBMIT. If the REPORT statement in the called program has no LINE-SIZE or LINE-COUNT specification of its own, its lists are displayed using the values WID and LEN. At the end of the called program, you can change the input values on the basic list and start a new program.

Printing Lists

You can send a list from a called program directly to the spool system instead of displaying it on the screen. To do this, use the TO SAP-SPOOL addition in the SUBMIT statement:

SUBMIT... TO SAP-SPOOL <print-parameters>.

For details of the parameters that you can use here, refer to Printing Lists [Page 904].

Saving Lists

Instead of displaying a list on the screen, you can store it in ABAP memory using the EXPORTING LIST TO MEMORY addition in the SUBMIT statement:

```
SUBMIT... AND RETURN
        EXPORTING LIST TO MEMORY.
```

This statement stores the list in ABAP memory, allowing the calling program to access it once the called program has finished. You must use the AND RETURN addition in this case. You cannot use the additions EXPORTING LIST TO MEMORY and TO SAP-SPOOL together.

The function group SLST provides function modules for accessing the saved list, including for example:

- LIST_FROM_MEMORY
- WRITE_LIST
- DISPLAY_LIST

Program Statements to Leave a Called Program

Usually, the user exits a program you called using SUBMIT ... AND RETURN by choosing F3 or F15 from list level 0 of the called report.

However, if you need to execute further statements before returning to the called program (for example, to place data in ABAP memory using the EXPORT statement), you need to modify the user interface of the called program. For example, you can define your own function code for the Back function and process it in the AT USER-COMMAND event. After you have written your additional statements, you can leave the called program using the LEAVE statement.

Control then returns to the point from which the program was called.

```
REPORT REP1 NO STANDARD PAGE HEADING.

DATA: ITAB TYPE I OCCURS 10,
      NUM TYPE I.

SUBMIT REP2 AND RETURN.

IMPORT ITAB FROM MEMORY ID 'HK'.

LOOP AT ITAB INTO NUM.
  WRITE / NUM.
ENDLOOP.

TOP-OF-PAGE.
WRITE 'Report 1'.
ULINE.
```

This program calls the following executable program (report):

```
REPORT REP2 NO STANDARD PAGE HEADING.

DATA: NUMBER TYPE I,
      ITAB TYPE I OCCURS 10.

SET PF-STATUS 'MYBACK'.

DO 5 TIMES.
  NUMBER = SY-INDEX.
  APPEND NUMBER TO ITAB.
  WRITE / NUMBER.
ENDDO.

TOP-OF-PAGE.
WRITE 'Report 2'.
ULINE.

AT USER-COMMAND.
  CASE SY-UCOMM.
    WHEN 'MBCK'.
      EXPORT ITAB TO MEMORY ID 'HK'.
      LEAVE.
  ENDCASE.
```

In the status MYBACK, the function code MBCK is assigned to the function keys `F3` and `F15`:

If the user chooses *Back* from the interface MYBACK, the system transfers table ITAB into ABAP memory and then leaves SAPMZTS1. In SAPMZTST, it reads table ITAB again.

Calling Transactions

If a program has a transaction code, there are two ways of starting it from another program.

If you do not want to return to the calling program at the end of the new transaction, use the statement:

LEAVE TO TRANSACTION <tcod> [AND SKIP FIRST SCREEN].

This statement ends the calling program and starts transaction <tcod>. This deletes the call stack (internal sessions) of all previous programs. At the end of the transaction, the system returns to the area menu from which the original program in the call stack was started.

If, on the other hand, you do not want to return to the calling program at the end of the new transaction, use the statement:

CALL TRANSACTION <tcod> [AND SKIP FIRST SCREEN] [USING <itab>].

This statement saves the data of the calling program, and starts transaction <tcod>. At the end of the transaction, the system returns to the statement following the call in the calling report. If the LEAVE statement occurs within the called transaction, the transaction ends and control returns to the program in which the call occurred.

You can use a variable to specify the transaction <tcod>. This allows you to call transactions statically as well as dynamically.

The addition AND SKIP FIRST SCREEN allows you to prevent the initial screen of the new transaction from being displayed. The first screen to be displayed is then the specified *Next screen* in the screen attributes of the initial screen. If the first screen calls itself as the next screen, you cannot skip it.

Furthermore, the AND SKIP FIRST SCREEN option works only if all mandatory fields on the initial screen of the new transaction are filled completely and correctly with input values from SPA/GPA parameters [Page 1033].

The USING ITAB addition in the CALL TRANSACTION statement allows you to pass an internal table <itab> to the new transaction. <itab> has the form of a batch input table. For further information about batch input tables, refer to Importing Data With Batch Input [Ext.].

Calling Screen Sequences as Modules

If you want to encapsulate a screen sequence with all of its flow logic and application logic, you must ensure that all of the screens in the sequence are linked to a single ABAP program. The ABAP program contains all of the dialog modules called by the screens in the sequence.

It is not currently possible to assign screens to a single procedure [Page 449] (subroutine, function module, or method) - they instead belong to the main program of the procedure.

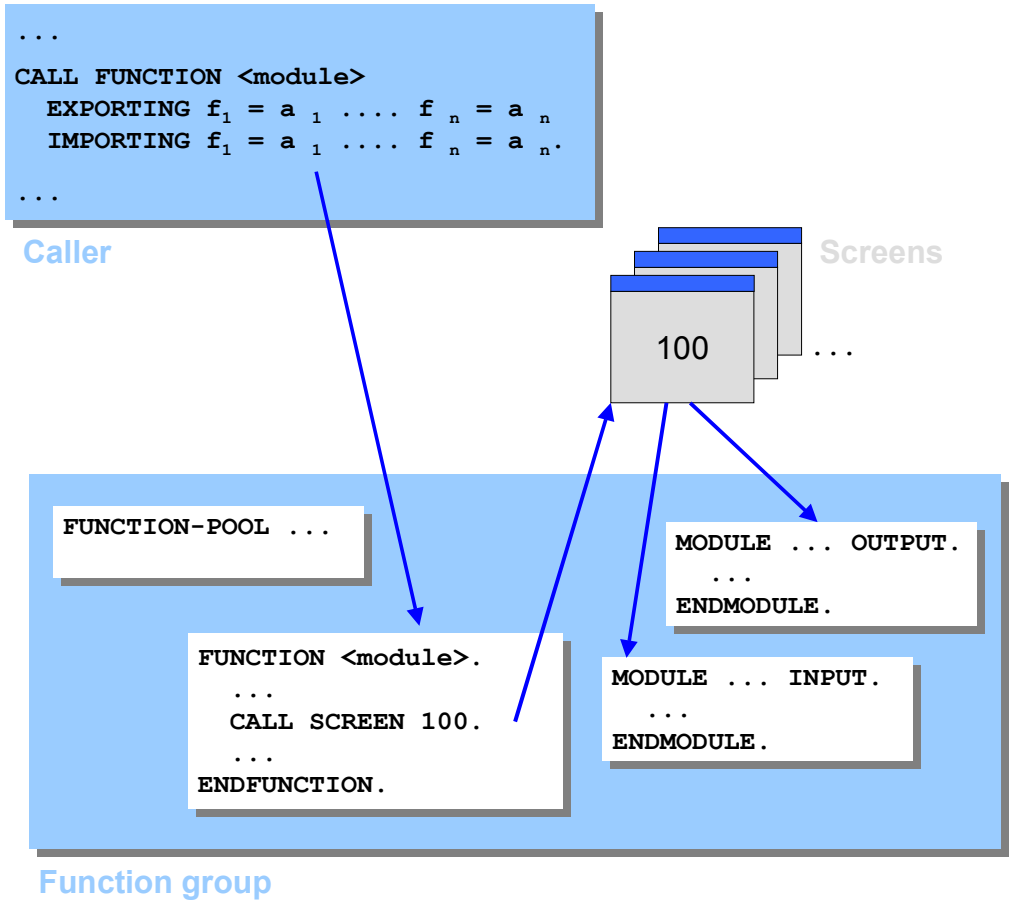
When you call a new main program in order to use a screen sequence (for example, using CALL TRANSACTION), a new SAP LUW begins. This may be an undesired effect, especially when you are trying to modularize a transaction that updates the database.

An alternative to this is to use an **external procedure**, whose main program is linked to a screen sequence, and which contains the corresponding dialog modules. Procedures run in the same SAP LUW and internal session as their calling program.

Using Function Modules

Function modules [Page 480] and their function groups are often used to define and call screen sequences that you want to run in the same SAP LUW as their calling program.

To encapsulate a screen sequence in a function module, you use the CALL SCREEN statement in the function module source code. The called screen and all of its subsequent screens belong to the function group to which the function module belongs. The dialog modules that you call from the screen flow logic are defined in the main program of the function module. The Function Builder assists you in defining dialog modules by automatically creating the appropriate include programs.



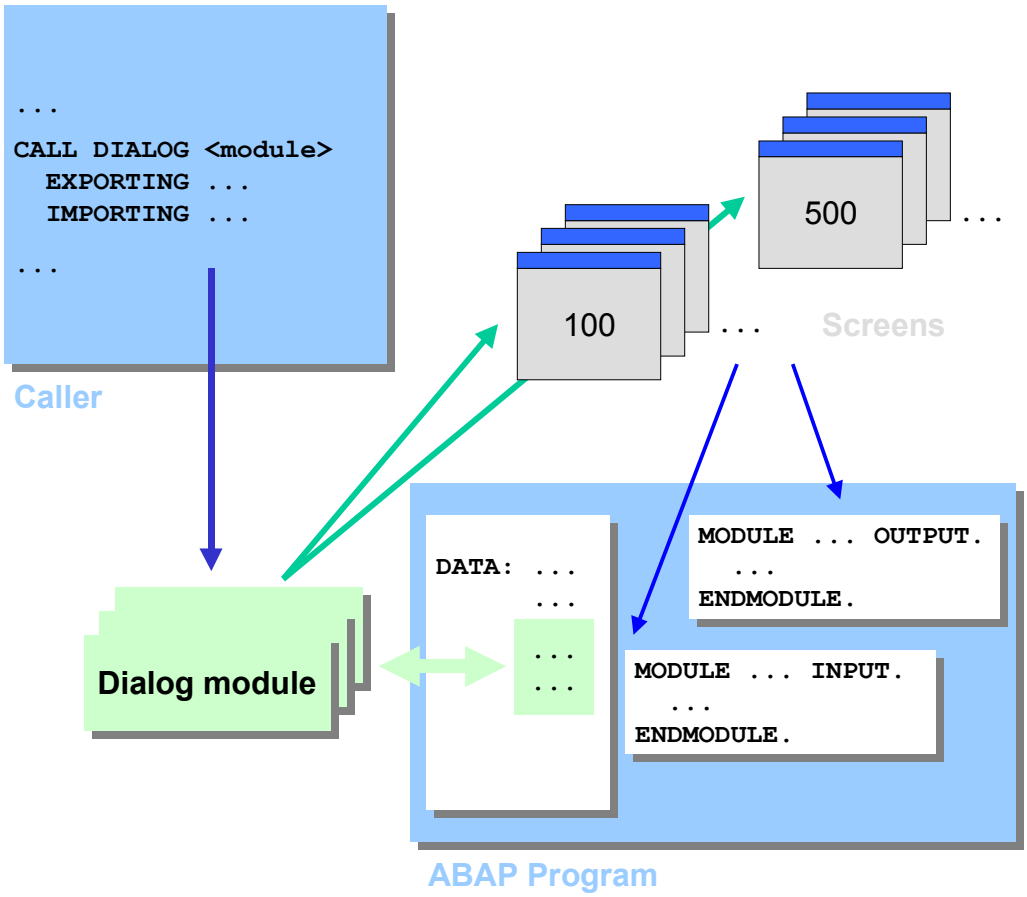
A function group like the one above contains only one function module and one screen sequence. However, you can also have several function modules with several independent screen sequences in the same function group.

Using Dialog Modules

Dialog modules were the predecessors of function modules. They are now obsolete, and you should no longer create new ones. However, for the sake of completeness, this section contains a brief description of their structure and function.

Dialog modules are objects that, like transaction codes, are linked to a screen sequence in an ABAP program by means of an initial screen. When you call a dialog module, the system calls the corresponding ABAP program and the initial screen of the dialog module. Dialog modules can also have an interface. Before you can do this, you must have declared the parameters as **global data** in the ABAP program. Dialog modules are created and administered in the ABAP Workbench using a tool (Transaction SE35) similar to the Function Builder. You can assign more than one dialog module to the same ABAP program.

Calling Screen Sequences as Modules



You call dialog modules using the CALL DIALOG statement:

CALL DIALOG <dialog>
[AND SKIP FIRST SCREEN]
[EXPORTING $f_1 = a_1 .... f_n = a_n$]
[IMPORTING $f_1 = a_1 .... f_n = a_n$]
[USING itab].

CALL DIALOG has the same syntax as CALL TRANSACTION [Page 1027]. You can also pass values to the global data of the called program using the interface.

Although a dialog module occupies a new internal session, it runs in the same SAP LUW as the calling program. However, since they open a new session, they are less efficient than function modules in performance terms. Nevertheless, they are the only means of switching internal session without starting a new SAP LUW. You can only run an entire transaction within the SAP LUW of a calling program by linking the initial screen of the transaction with a dialog module.

If you want to run screen sequences and dialog modules in the SAP LUW of the calling program and as a transaction in their own SAP LUW, you must ensure that all database update tasks can be executed without compromising data integrity. The following special conditions apply:

- Dialog modules inherit locks [Page 1270] set by the calling program.

If the screen sequence is called as a dialog module, the system assumes that a lock for a particular object already exists. When you start the screen sequence as a transaction, you must set your own locks. To find out at runtime if a program is running as a called program, use the system variable SY-CALLD.

- Dialog modules inherit update [Page 1276] keys from the calling program.
- When the screen sequence is running as a dialog module, the system ignores COMMIT WORK statements.

  The COMMIT WORK statement can therefore be used in a screen sequence that is used both as a transaction and as a dialog module. It is ignored if the screen sequence is running as a dialog module. Instead, all updates requested in the screen sequence are processed at the next COMMIT WORK in the calling program.
- Update function modules called using the IN UPDATE TASK addition are not started if the screen sequence is running as a dialog module.

  You must ensure that any function modules called with IN UPDATE TASK can be delayed until the next COMMIT WORK in the calling program.
- PERFORM ON COMMIT routines are not executed in the dialog module.

  You must ensure that any subroutines called using ON COMMIT can be delayed until the next COMMIT WORK in the calling program. Remember that the global data of the dialog module is destroyed along with the internal session when control returns to the calling program. Consequently, subroutines called using PERFORM ON COMMIT must not use this global data.
- If you want to run the screen sequence as a transaction, you must ensure that the input parameters of the interface contain reasonable default values.

Passing Data Between Programs

There are two ways of passing data to a called program:

Passing Data Using Internal Memory Areas

There are two cross-program memory areas to which ABAP programs have access (refer to the diagram in Memory Structures of an ABAP Program [Page 66]) that you can use to pass data between programs.

SAP Memory

SAP memory is a memory area to which all main sessions within a SAPgui have access. You can use SAP memory either to pass data from one program to another within a session, or to pass data from one session to another. Application programs that use SAP memory must do so using SPA/GPA parameters (also known as SET/GET parameters). These parameters can be set either for a particular user or for a particular program using the SET PARAMETER statement. Other ABAP programs can then retrieve the set parameters using the GET PARAMETER statement. The most frequent use of SPA/GPA parameters is to fill input fields on screens (see below).

ABAP Memory

ABAP memory is a memory area that all ABAP programs within the same internal session can access using the EXPORT and IMPORT statements. Data within this area remains intact during a whole sequence of program calls. To pass data to a program which you are calling, the data needs to be placed in ABAP memory before the call is made. The internal session of the called program then replaces that of the calling program. The program called can then read from the ABAP memory. If control is then returned to the program which made the initial call, the same process operates in reverse. For further information, refer to Data Clusters in ABAP Memory [Page 363].

Filling Input Fields on an Initial Screen

Most programs that you call from other programs have their own initial screen that the user must fill with values. For an executable program, this is normally the selection screen. The SUBMIT statement has a series of additions that you can use to fill the input fields of the called program:

Filling the Selection Screen of a Called Program [Page 1019]

You cannot fill the input fields of a screen using additions in the calling statement. Instead, you can use SPA/GPA parameters. For further information, refer to Filling an Initial Screen Using SPA/GPA Parameters [Page 1033].

Filling an Initial Screen using SPA/GPA Parameters

To fill the input fields of a called transaction with data from the calling program, you can use the SPA/GPA technique. SPA/GPA parameters are values that the system stores in the global, user-specific SAP memory. SAP memory allows you to pass values between programs. A user can access the values stored in the SAP memory during one terminal session for all parallel sessions. Each SPA/GPA parameter is identified by a 20-character code. You can maintain them in the Repository Browser in the ABAP Workbench. The values in SPA/GPA parameters are user-specific.

ABAP programs can access the parameters using the SET PARAMETER and GET PARAMETER statements.

To fill one, use:

SET PARAMETER ID <pid> FIELD <f>.

This statement saves the contents of field <f> under the ID <pid> in the SAP memory. The code <pid> can be up to 20 characters long. If there was already a value stored under <pid>, this statement overwrites it. If the ID <pid> does not exist, double-click <pid> in the ABAP Editor to create a new parameter object.

To read an SPA/GPA parameter, use:

GET PARAMETER ID <pid> FIELD <f>.

This statement fills the value stored under the ID <pid> into the variable <f>. If the system does not find a value for <pid> in the SAP memory, it sets SY-SUBRC to 4, otherwise to 0.

To fill the initial screen of a program using SPA/GPA parameters, you normally only need the SET PARAMETER statement.

The relevant fields must each be linked to an SPA/GPA parameter.

On a selection screen, you link fields to parameters using the MEMORY ID addition in the PARAMETERS or SELECT-OPTIONS statement. If you specify an SPA/GPA parameter ID when you declare a parameter or selection option, the corresponding input field is linked to that input field.

On a screen, you link fields to parameters in the Screen Painter. When you define the field attributes of an input field, you can enter the name of an SPA/GPA parameter in the *Parameter ID* field in the screen attributes. The SET parameter and GET parameter checkboxes allow you to specify whether the field should be filled from the corresponding SPA/GPA parameter in the PBO event, and whether the SPA/GPA parameter should be filled with the value from the screen in the PAI event.

When an input field is linked to an SPA/GPA parameter, it is initialized with the current value of the parameter each time the screen is displayed. This is the reason why fields on screens in the R/3 System often already contain values when you call them more than once.

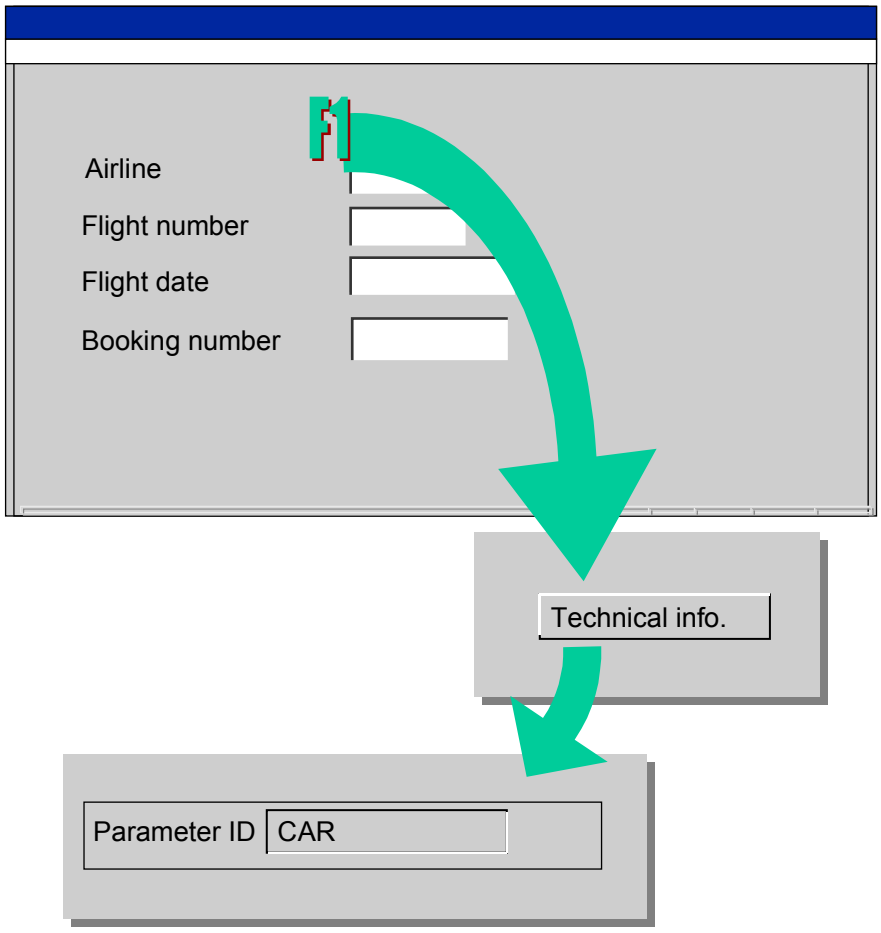
When you call programs, you can use SPA/GPA parameters with no additional programming overhead if, for example, you need to fill obligatory fields on the initial screen of the called program. The system simply transfers the values from the parameters into the input fields of the called program.

However, you can control the contents of the parameters from your program by using the SET PARAMETER statement before the actual program call. This technique is particularly useful if you want to skip the initial screen of the called program and that screen contains obligatory fields.

Filling an Initial Screen using SPA/GPA Parameters

If you want to set SPA/GPA parameters before a program call, you need to know which parameters are linked to which fields on the initial screen. A simple way of doing this is to start the program that you want to call, place the cursor on the input fields, and choose F1 followed by *Technical info*. The *Parameter ID* field contains the name of the corresponding SPA/GPA parameter. Alternatively, you can look at the screen definition in the Screen Painter.

The technical information for the first input field of the booking transaction TCG2 looks like this:



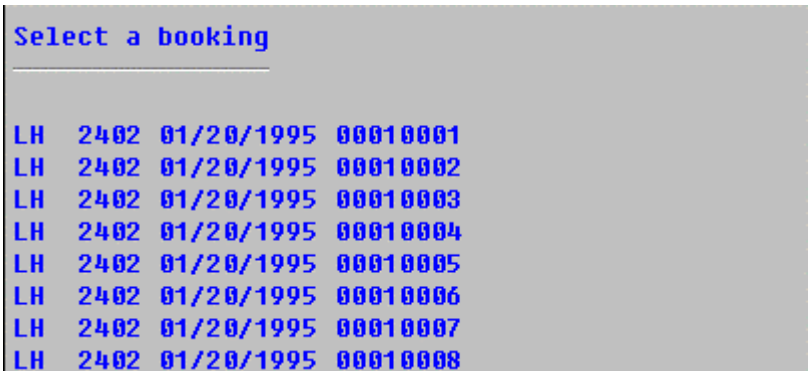
The SPA/GPA parameter for the input field *Company* has the ID CAR. Use this method to find the IDs CON, DAY, and BOK for the other input fields.

The following executable program is connected to the logical database F1S and calls an update transaction:

```
REPORT BOOKINGS NO STANDARD PAGE HEADING.

TABLES SBOOK.

START-OF-SELECTION.
  WRITE: 'Select a booking',
```

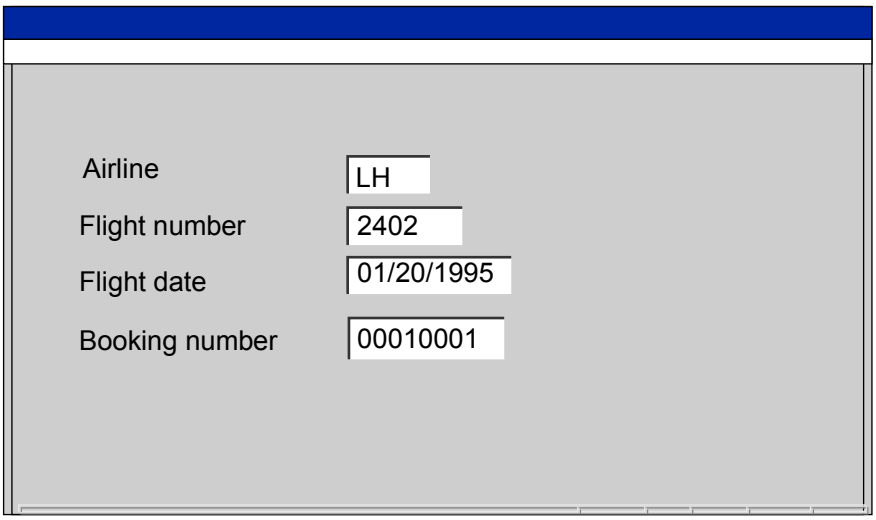
```
      / '----------------'.
  SKIP.

GET SBOOK.
  WRITE: SBOOK-CARRID, SBOOK-CONNID,
         SBOOK-FLDATE, SBOOK-BOOKID.
  HIDE:  SBOOK-CARRID, SBOOK-CONNID,
         SBOOK-FLDATE, SBOOK-BOOKID.

AT LINE-SELECTION.
  SET PARAMETER ID: 'CAR' FIELD SBOOK-CARRID,
                    'CON' FIELD SBOOK-CONNID,
                    'DAY' FIELD SBOOK-FLDATE,
                    'BOK' FIELD SBOOK-BOOKID.
  CALL TRANSACTION 'BOOK'.
```

The basic list of the program shows fields from the database table SBOOK according to the user entries on the selection screen. These data are also stored in the HIDE areas of each line.



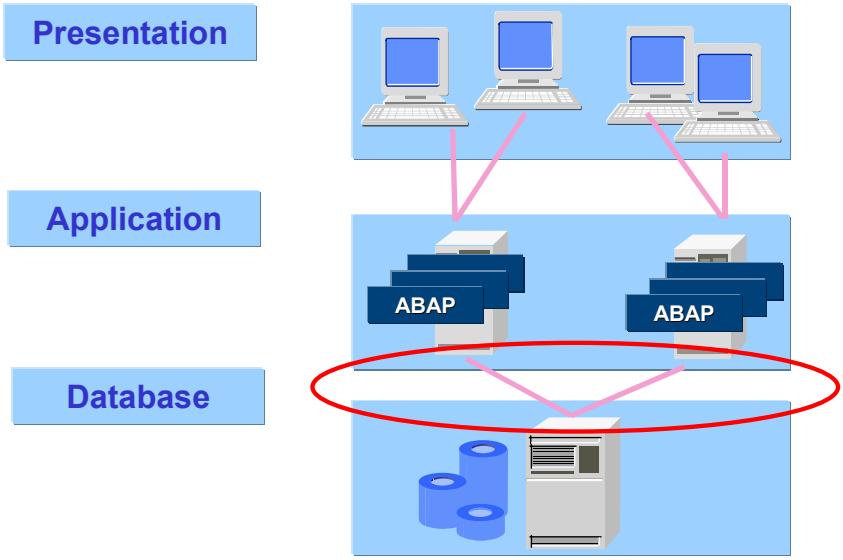
If the user selects a line of booking data by double-clicking, the system triggers the AT LINE-SELECTION event and takes the data stored in the HIDE area to fill them into the SPA/GPA parameters of the initial screen of the transaction. Then it calls the transaction. Since you do not suppress the initial screen using AND SKIP FIRST SCREEN, the initial screen may appear as follows:

Filling an Initial Screen using SPA/GPA Parameters



If you would use the AND SKIP FIRST SCREEN option with the CALL TRANSACTION statement, the second screen would appear immediately, since all obligatory fields of the first screen are filled.

ABAP Database Access



This section describes how ABAP programs communicate with the central database in the R/3 System.

Accessing the Database in the R/3 System

Berechtigungskonzept [Page 504]

In the R/3 System, long-life data is stored in relational database tables. In a relational database model, the real world is represented by tables. A table is a two-dimensional matrix, consisting of lines and columns (fields). The smallest possible combination of fields that can uniquely identify each line of the table is called the key. Each table must have at least one key, and each table has one key that is defined as its primary key. Relationships between tables are represented by foreign keys.

Standard SQL

SQL (Structured Query Language) is a largely standardized language for accessing relational databases. It can be divided into three areas:

- Data Manipulation Language (DML)

  Statements for reading and changing data in database tables.

- Data Definition Language (DDL)

  Statements for creating and administering database tables.

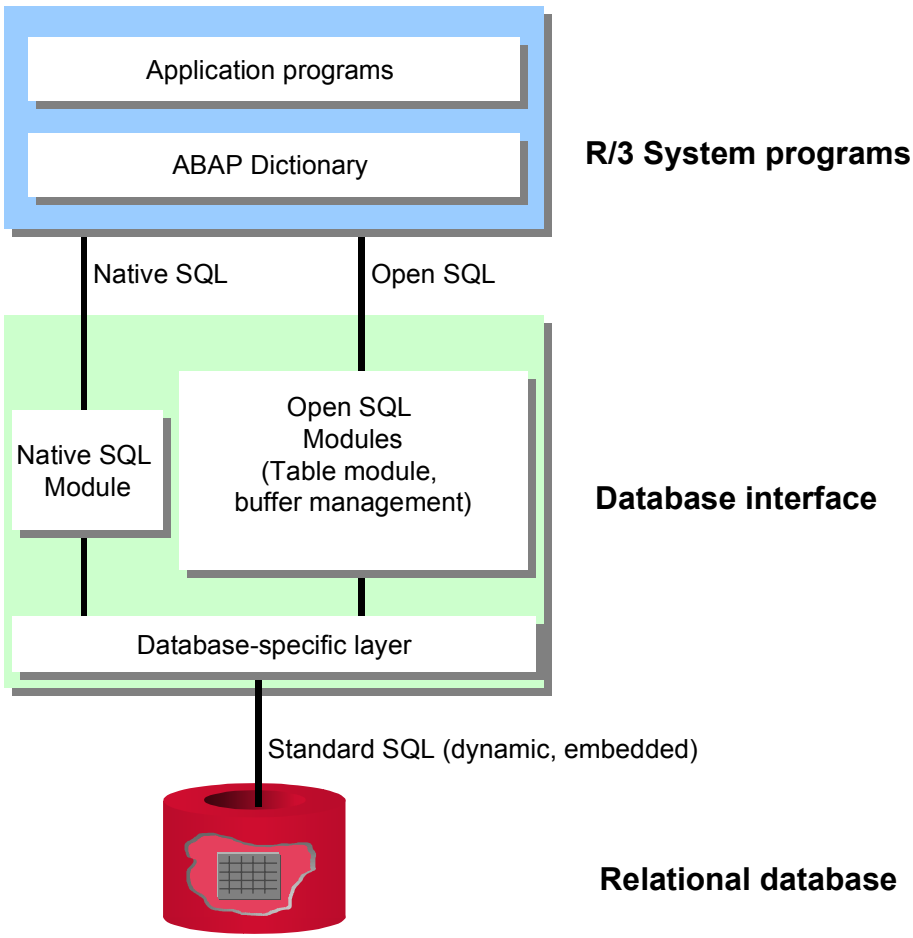
- Data Control Language (DCL)

  Statements for authorization and consistency checks.

Each database has a programming interface that allows you to access the database tables using SQL statements. The SQL statements in these programming interfaces are not fully standardized. To access a specific database system, you must refer to the documentation of that system for a list of the SQL statements available and their correct syntax.

The Database Interface

To make the R/3 System independent of the database system with which you use it despite the differences in the SQL syntax between various databases, each work process on an application server has a database interface. The R/3 System communicates with the database by means of this interface. The database interface converts all of the database requests from the R/3 System into the correct Standard SQL statements for the database system. To do this, it uses a database-specific component that shields the differences between database systems from the rest of the database interface. You choose the appropriate layer when you install the R/3 System.



There are two ways of accessing the database from a program - with Open SQL or Native SQL.

Open SQL

Open SQL statements are a subset of Standard SQL that is fully integrated in ABAP. They allow you to access data irrespective of the database system that the R/3 installation is using. Open SQL consists of the Data Manipulation Language (DML) part of Standard SQL; in other words, it allows you to read (SELECT) and change (INSERT, UPDATE, DELETE) data.

Open SQL also goes beyond Standard SQL to provide statements that, in conjunction with other ABAP constructions, can simplify or speed up database access. It also allows you to buffer certain tables on the application server, saving excessive database access. In this case, the database interface is responsible for comparing the buffer with the database. Buffers are partly stored in the working memory of the current work process, and partly in the shared memory for all work processes on an application server. Where an R/3 System is distributed across more than one application server, the data in the various buffers is synchronized at set intervals by the buffer management. When buffering the database, you must remember that data in the buffer is not always up to date. For this reason, you should only use the buffer for data which does not often change. You specify whether a table can be buffered in its definition in the ABAP Dictionary.

Native SQL

Native SQL is only loosely integrated into ABAP, and allows access to all of the functions contained in the programming interface of the respective database system. Unlike Open SQL statements, Native SQL statements are not checked and converted, but instead are sent directly to the database system. When you use Native SQL, the function of the database-dependent layer is minimal. Programs that use Native SQL are specific to the database system for which they were written. When writing R/3 applications, you should avoid using Native SQL wherever possible. It is used, however, in some parts of the R/3 Basis System - for example, for creating or changing table definitions in the ABAP Dictionary.

The ABAP Dictionary

The ABAP Dictionary, part of the ABAP Workbench, allows you to create and administer database tables. Open SQL contains no statements from the DDL part of Standard SQL. Normal application programs should not create or change their own database tables.

The ABAP Dictionary uses the DDL part of Open SQL to create and change database tables. It also administers the ABAP Dictionary in the database. The ABAP Dictionary contains metadescriptions of all database tables in the R/3 System. Only database tables that you create using the ABAP Dictionary appear in the Dictionary. Open SQL statements can only access tables that exist in the ABAP Dictionary.

Authorization and Consistency Checks

The DCL part of Standard SQL is not used in R/3 programs. The work processes within the R/3 System are logged onto the database system as users with full rights. The authorizations of programs or users to read or change database tables is administered within the R/3 System using the R/3 authorization concept. Equally, transactions must ensure their own data consistency using the R/3 locking concept. For more information, refer to Programming Database Updates [Page 1260]

Open SQL

Open SQL consists of a set of ABAP statements that perform operations on the central database in the R/3 System. The results of the operations and any error messages are independent of the database system in use. Open SQL thus provides a uniform syntax and semantics for all of the database systems supported by SAP. ABAP programs that only use Open SQL statements will work in any R/3 System, regardless of the database system in use. Open SQL statements can only work with database tables that have been created in the ABAP Dictionary.

In the ABAP Dictionary, you can combine columns of different database tables to a database view (or view for short). In Open SQL statements, views are handled in exactly the same way as database tables. Any references to database tables in the following sections can equally apply to views.

Overview

Open SQL contains the following keywords:

Keyword	Function
SELECT	Reads data from database tables
INSERT	Adds lines to database tables
UPDATE	Changes the contents of lines of database tables
MODIFY	Inserts lines into database tables or changes the contents of existing lines
DELETE	Deletes lines from database tables
OPEN CURSOR, FETCH, CLOSE CURSOR	Reads lines of database tables using the cursor

Return Codes

All Open SQL statements fill the following two system fields with return codes:

- SY-SUBRC

  After **every** Open SQL statement, the system field SY-SUBRC contains the value 0 if the operation was successful, a value other than 0 if not.

- SY-DBCNT

  After an open SQL statement, the system field SY-DBCNT contains the number of database lines processed.

Client Handling

A single R/3 System can manage the application data for several separate areas of a business (for example, branches). Each of these commercially separate areas in the R/3 System is called a client, and has a number. When a user logs onto an R/3 System, they specify a client. The first column in the structure of every database table containing application data is the client field (MANDT, from the German word for client). It is also the first field of the table key. Only universal system tables are client-independent, and do not contain a client name.

By default, Open SQL statements use automatic client handling. Statements that access **client-dependent** application tables only use the data from the current client. You cannot specify a condition for the client field in the WHERE clause of an Open SQL statement. If you do so, the system will either return an error during the syntax check or a runtime error will occur. You

Open SQL

cannot overwrite the MANDT field of a database using Open SQL statements. If you specify a different client in a work area, the ABAP runtime environment automatically overwrites it with the current one before processing the Open SQL statement further.

Should you need to specify the client specifically in an Open SQL statement, use the addition

... CLIENT SPECIFIED ....

directly after the name of the database table. This addition disables the automatic client handling and you can use the field MANDT both in the WHERE clause and in a table work area.

Reading data [Page 1043]

Changing data [Page 1090]

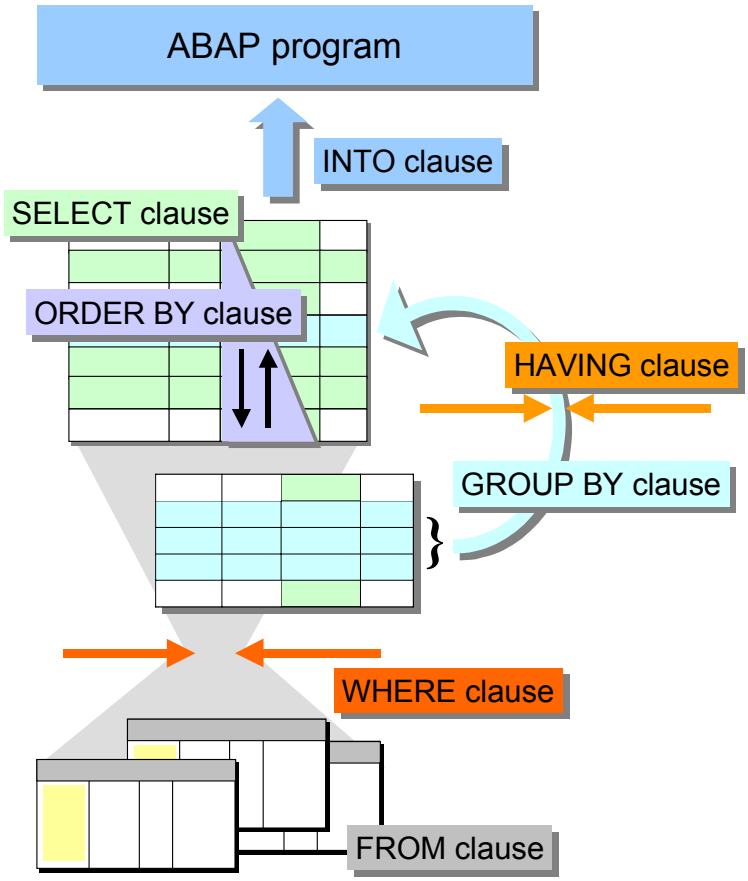
Reading Data

Sperrkonflikten [Page 1089]

The Open SQL statement for reading data from database tables is:

```
SELECT    <result>
INTO      <target>
FROM      <source>
[WHERE    <condition>]
[GROUP BY <fields>]
[HAVING   <cond>]
[ORDER BY <fields>].
```

The SELECT statement is divided into a series of simple clauses, each of which has a different part to play in selecting, placing, and arranging the data from the database.



Clause	Description
SELECT <result> [Page 1045]	The SELECT clause defines the structure of the data you want to read, that is, whether one line or several, which columns you want to read, and whether identical entries are acceptable or not.

Reading Data

INTO <target> [Page 1052]	The INTO clause determines the target area <target> into which the selected data is to be read.
FROM <source> [Page 1058]	The FROM clause specifies the database table or view <source> from which the data is to be selected. It can also be placed before the INTO clause.
WHERE <cond> [Page 1064]	The WHERE clause specifies which lines are to be read by specifying conditions for the selection.
GROUP BY <fields> [Page 1072]	The GROUP-BY clause produces a single line of results from groups of several lines. A group is a set of lines with identical values for each column listed in <fields>.
HAVING <cond> [Page 1075]	The HAVING clause sets logical conditions for the lines combined using GROUP BY.
ORDER BY <cond> [Page 1077]	The ORDER-BY clause defines a sequence <fields> for the lines resulting from the selection.

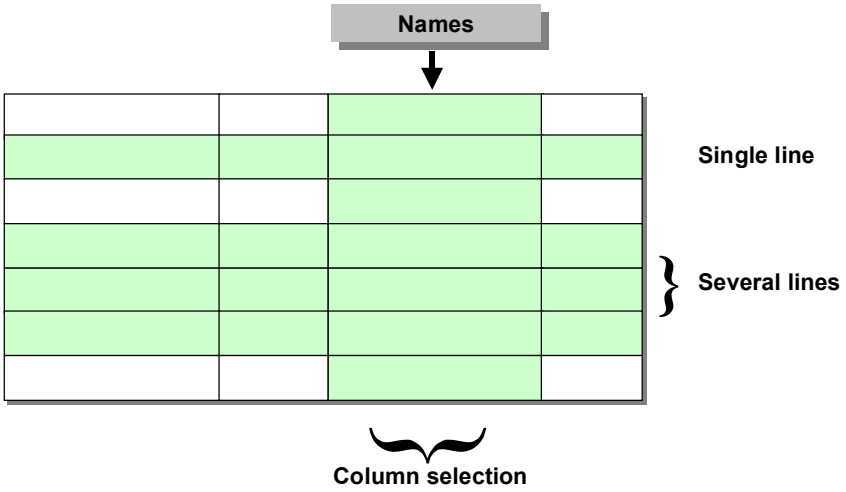
The individual clauses and the ways in which they combine are all very important factors in the SELECT statement. Although it is a single statement like any other, beginning with the SELECT keyword and ending with a period, its division into clauses, and the ways in which they combine, make it more powerful than other statements. A single SELECT statement can perform functions ranging from simply reading a single line to executing a very complicated database query.

You can use SELECT statements in the WHERE and HAVING clauses. These are called subqueries [Page 1080].

You can decouple the INTO clause from the SELECT statement by reading from the database using a cursor [Page 1084].

Defining Selections

The SELECT clause defines the structure of the result set (selection) that you want to read from the database.



The selection can be flat (one line) or tabular (several lines). You can specify whether to accept or exclude duplicate entries. The SELECT clause also specifies the names of the columns to be read. You can replace the names of the database fields with alternative names. Aggregate functions can be applied to individual columns.

The SELECT clause can be divided into two parts for lines and columns:

SELECT <lines> <cols> ...

<lines> specifies whether you want to read one or more lines. <cols> defines the column selection.

Reading a Single Line

To read a single entry from the database, use the following:

SELECT SINGLE <cols> ... WHERE ...

To ensure that the line can be uniquely identified, you must specify values for all of the fields of the primary key of the table in the WHERE clause. If the WHERE clause does not contain all of the key fields, the syntax check produces a warning, and the SELECT statement reads the first entry that it finds that matches the key fields that you have specified.

The result of the selection is either an elementary field or a flat structure, depending on the number of columns you specified in <cols>. The target area in the INTO clause must be appropriately convertible.

If the system finds a line with the corresponding key, SY-SUBRC is set to 0, otherwise to 4.

Reading Several Lines

To read a several entries from the database, use the following:

SELECT [DISTINCT] <cols> ... WHERE ...

Defining Selections

If you do not use DISTINCT (<lines> is then empty), the system reads all of the lines that satisfy the WHERE condition. If you use DISTINCT, the system excludes duplicate entries.

The result of the selection is a table. The target area of the INTO clause can be an internal table with a line type appropriate for <cols>. If the target area is not an internal table, but a flat structure, you must include an ENDSELECT statement after the SELECT statement:

```
SELECT [DISTINCT] <cols> ... WHERE ...
  ...
ENDSELECT.
```

The lines are read in a loop [Page 245] one by one into the target area specified in the INTO clause You can work with the target area within the loop.

If at least one line is read, SY-SUBRC is set to 0 after the statement (or loop) has been processed. If no lines are read, SY-SUBRC is set to 4. The number of lines read is placed in the system field SY-DBCNT. Within the loop, SY-DBCNT already contains the number of lines that have already been passed to the target area.

Technically, it is possible to nest SELECT loops. However, for performance reasons, you should avoid doing so. If you want to read interdependent data from more than one database table, you can use a join in the FROM clause or a subquery in the WHERE clause.

Reading the Whole Line

To read all of the columns in the database table, use the following:

```
SELECT <lines> * ...
```

This reads all columns for the specified lines. The data type of the selected lines is a structure with exactly the same data type as the database table in the ABAP Dictionary. The target area of the INTO clause should be compatible with, or at least convertible into this data type. In the other clauses, you can only address the columns under their names in the database table.

Reading individual columns can be considerably more efficient than reading all of the columns in a table. You should therefore only read the columns that you need in your program.

Reading Single Columns

To read single columns from the database table, use the following:

```
SELECT <lines> <s1> [AS <a1>] <s2> [AS <a2>] ...
```

where $<s_i>$ are single columns. There are different ways of addressing the columns, depending on the form of your FROM clause:

- $<s_i>$ specifies the component $<c_i>$ of the database table. This is only possible if the corresponding name is unique. This is always the case if you only specify a single database table in the FROM clause.
- $<s_i>$ specifies the full name of the required component in the form <dbtab>~$<c_i>$, where <dbtab> is the name of the database table. This is always necessary if the component name appears in more than one database table in the FROM clause, and is only possible if the database table <dbtab> only appears once in the FROM clause.
- $<s_i>$ specifies the full name of the required component in the form <tabalias>~$<c_i>$, where <tabalias> is an alias name for the database table. This is always necessary if the table from which you want to read occurs more than once in the FROM clause. You must define the alias name <tabalias> in the FROM clause.

In the SELECT clause, you can use the AS addition to specify an alias name <a_i> for each column <s_i>. The alias column name is used instead of the real name in the INTO and ORDER BY clauses. This allows you for example, to read the contents of a column <s_i> into a component <a_i> of a structure when you use the INTO CORRESPONDING FIELDS OF addition.

The data type of a column is the Dictionary type of the corresponding underlying domain in the ABAP Dictionary. You must choose the data types of the target fields in the INTO clause so that the Dictionary types can easily be converted. For a table of Dictionary types and their corresponding ABAP data types, refer to Data Types in the ABAP Dictionary [Page 105].

Reading Aggregate Data for Columns

To read aggregate data for a column in the database, use the following:

```
SELECT <lines> <agg>( [DISTINCT] <s1> ) [AS <a1>]
               <agg>( [DISTINCT] <s2> ) [AS <a2>] ...
```

where <s_i> are the same field labels as above. The expression <agg> represents one of the following aggregate functions:

- MAX: returns the maximum value of the column <s_i>
- MIN: returns the minimum value of the column <s_i>
- AVG: returns the average value of the column <s_i>
- SUM: returns the sum value of the column <s_i>
- COUNT: counts values or lines as follows:
  - COUNT(DISTINCT <s_i>) returns the number of different values in the column <s_i>.
  - COUNT(*) returns the total number of lines in the selection.

You can exclude duplicate values from the calculation using the DISTINCT option. The spaces between the parentheses and the arguments of the aggregate expressions must not be left out. The arithmetic operators AVG and SUM only work with numeric fields.

The data type of aggregate functions using MAX, MIN, or SUM is the Dictionary type of the corresponding column. Aggregate expressions with the function AVG have the Dictionary type FLTP, and those with COUNT have the Dictionary type INT4. The target field should have the corresponding type. When you calculate average values, it is a good idea to use the ABAP type F. However, remember that the database system may use different approximations to ABAP. When you calculate a sum, ensure that the target field is large enough.

Unlike ABAP, database systems recognize null values in database fields. A null value means that a field has no contents, and it is not included in calculations. The result of the calculation is only null if all of the lines in the selection contain the null value in the corresponding field. In ABAP, the null value is interpreted as zero (depending on the data type of the field).

In the AS addition, you can define an alternative column name <a_i> for each aggregate expression. The alias column name is used instead of the real name in the INTO and ORDER BY clauses. This is the only way of sorting by an aggregate expression in the ORDER BY clause.

If the list in the SELECT clause (excepting aggregate expressions) contains one or more field names, the field names must also be listed in the GROUP BY clause. The aggregate functions do not then apply to all of the selected lines, but to the individual groups of lines.

Specifying Columns Dynamically

You can also specify <cols> dynamically as follows:

SELECT <lines> (<itab>) ...

The parentheses must include the name of an internal table <itab> that is either empty or contains $<s_1>$ $<s_2>$ ... to specify the columns or aggregate expressions to be read. For this purpose, the line type of <itab> must be a type C field with a maximum length of 72. If the internal table is empty, the system reads all columns.

Examples

Reading certain columns of a single line:

```
DATA WA TYPE SPFLI.

SELECT SINGLE CARRID CONNID CITYFROM CITYTO
INTO   CORRESPONDING FIELDS OF WA
FROM   SPFLI
WHERE  CARRID EQ 'LH' AND CONNID EQ '0400'.

IF SY-SUBRC EQ 0.
  WRITE: / WA-CARRID, WA-CONNID, WA-CITYFROM, WA-CITYTO.
ENDIF.
```

The output is:

```
LH  0400 FRANKFURT            NEW YORK
```

SINGLE in the SELECT clause means that the statement reads a single entry from the database table SPFLI where the primary key fields CARRID and CONNID have the values specified in the WHERE clause. The columns specified in the SELECT clause are transferred to the identically-named components of the structure WA.

Reading particular columns of more than one line:

```
DATA: ITAB TYPE STANDARD TABLE OF SPFLI,
      WA LIKE LINE OF ITAB.

SELECT CARRID CONNID CITYFROM CITYTO
INTO   CORRESPONDING FIELDS OF TABLE ITAB
FROM   SPFLI
WHERE  CARRID EQ 'LH'.

IF SY-SUBRC EQ 0.
  LOOP AT ITAB INTO WA.
    WRITE: / WA-CARRID, WA-CONNID, WA-CITYFROM, WA-CITYTO.
  ENDLOOP.
ENDIF.
```

The output is:

```
LH   0400 FRANKFURT            NEW YORK
LH   2402 FRANKFURT            BERLIN
LH   0402 FRANKFURT            NEW YORK
```

Since there are no lines specified in the SELECT clause, the statement reads all of the lines from the database table SPFLI that satisfy the condition in the WHERE clause. The columns specified in the SELECT clause are transferred to the identically-named components of the internal table ITAB.

Reading all columns of more than one line:

```
DATA WA TYPE SPFLI.

SELECT *
INTO   CORRESPONDING FIELDS OF WA
FROM   SPFLI
WHERE  CARRID EQ 'LH'.

  WRITE: / SY-DBCNT,
           WA-CARRID, WA-CONNID, WA-CITYFROM, WA-CITYTO.

ENDSELECT.
```

The output is:

```
  1  LH   0400 FRANKFURT            NEW YORK
  2  LH   2402 FRANKFURT            BERLIN
  3  LH   0402 FRANKFURT            NEW YORK
```

Since there are no lines specified in the SELECT clause, the statement reads all of the lines from the database table SPFLI that satisfy the condition in the WHERE clause. All of the columns in the table are transferred to the identically-named components of the flat structure WA. This is why you must conclude the SELECT loop with the ENDSELECT statement.

Aggregate functions

Suppose a database table TEST, consisting of two columns and 10 lines:

COL_1	COL_2
1	3
2	1
3	5
4	7
5	2
6	3
7	1
8	9
9	4
10	3

The following coding demonstrates the aggregate functions:

```
DATA RESULT TYPE P DECIMALS 2.

SELECT <agg>( [DISTINCT] COL_2 )
INTO   RESULT
FROM   TEST.

WRITE RESULT.
```

The following table shows the results of this program extract according to different combinations of aggregate expressions <agg> and the DISTINCT option.

Aggregate expression	DISTINCT	Result
MAX	**no**	**9.00**
MAX	**yes**	**9.00**
MIN	**no**	**1.00**
MIN	**yes**	**1.00**
AVG	**no**	**3.80**
AVG	**yes**	**4.43**
SUM	**no**	**38.00**
SUM	**yes**	**31.00**
COUNT	**yes**	**7.00**
COUNT(*)	**---**	**10.00**

Specifying Columns Dynamically

```
DATA: ITAB TYPE STANDARD TABLE OF SPFLI,
      WA LIKE LINE OF ITAB.

DATA: LINE(72) TYPE C,
      LIST LIKE TABLE OF LINE(72).

LINE = ' CITYFROM CITYTO '.
APPEND LINE TO LIST.

SELECT DISTINCT (LIST)
       INTO CORRESPONDING FIELDS OF TABLE ITAB
       FROM SPFLI.

IF SY-SUBRC EQ 0.
  LOOP AT ITAB INTO WA.
    WRITE: / WA-CITYFROM, WA-CITYTO.
  ENDLOOP.
ENDIF.
```

The output is:

```
FRANKFURT             BERLIN
FRANKFURT             NEW YORK
FRANKFURT             SAN FRANCISCO
NEW YORK              SAN FRANCISCO
ROME                  FRANKFURT
SINGAPORE             FRANKFURT
SINGAPORE             HONGKONG
SINGAPORE             SAN FRANCISCO
TOKYO                 ROME
```

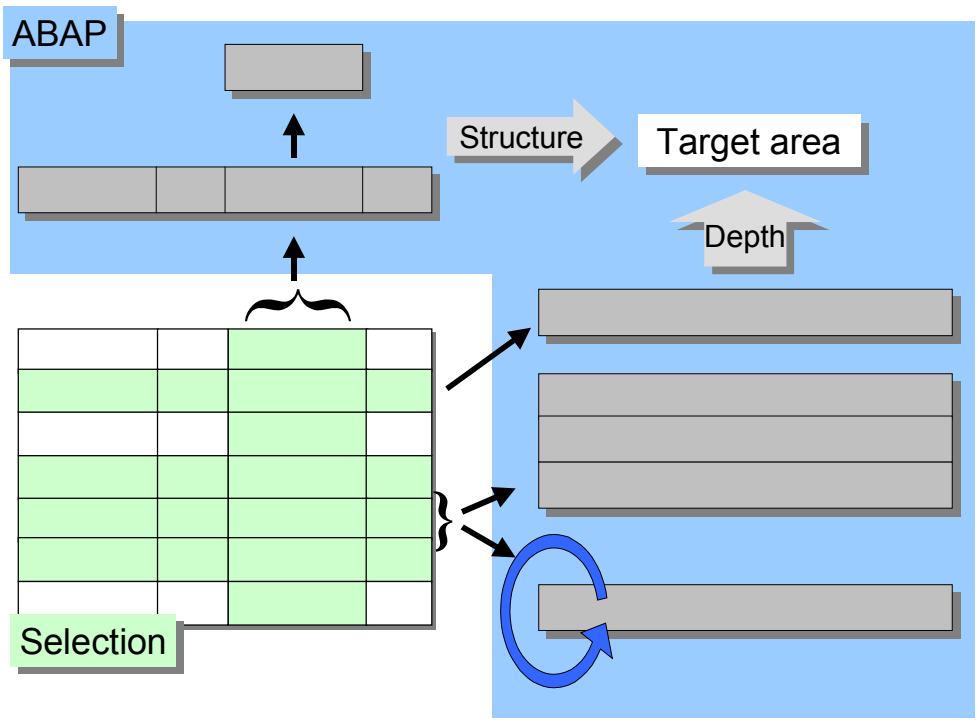
The internal table ITAB contains the columns of the database table SPFLI to be read. The DISTINCT addition in the SELECT clause means that the statement only reads those lines that have different contents in both of these columns. The result is a list of possible routes.

Specifying a Target Area

The INTO clause defines the target area into which the selection from the SELECT clause is written. Suitable target areas are variables [Page 123] whose data type is compatible with or convertible into that of the selection in the SELECT clause.

The SELECT clause determines the data type of the target area as follows:

- The <lines> specification in the SELECT clause determines the depth of the target area, that is, whether it is a flat or a tabular structure.
- The <cols> specification in the SELECT clause determines the structure (line type) of the target area.



If you select a single line, the target area must be flat. If you select more than one line, the target area may be either tabular or flat. If the target area is flat, you need to use a SELECT loop.

When you select all of the columns, the target area must either be a structure or convertible into one. When you select individual columns, the target area can be a component of a structure or a single field.

The elementary data types in the selection in the SELECT clause are Dictionary types. These Dictionary types must be able to be converted into the ABAP data types of the corresponding elementary components of the target area. For a table of data types, refer to Data Types in the ABAP Dictionary [Page 105].

Specifying a Flat Work Area

You can specify a flat work area regardless of whether you are reading from a single line or from several. To read data into one, use the following in the INTO clause:

SELECT ... INTO [CORRESPONDING FIELDS OF] <wa> ...

The target area must be at least as large as the line to be read into it. However, this is the only restriction on its data type. You should base your choice of data type on the specifications in the SELECT clause such that the columns that you have read can be addressed in <wa>.

If you specify an asterisk (*) in the SELECT clause (select all columns), the data is transferred into the work area from left to right according to the structure of the database table. The structure of <wa> is irrelevant in this case. In order to enable you to address the values in the individual columns after the SELECT statement, the work area should have the same structure as the database table.

If you specified individual columns or aggregate expressions in the SELECT clause, the columns are transferred into the work area from left to right according to the structure of the work area.

When you read the data into <wa>, its previous contents are overwritten. However, the components of <wa> that are not affected by the SELECT statement retain their previous values.

If the work area is structured, you can use the CORRESPONDING FIELDS addition. This transfers only the contents of fields whose names are identical in the database table and the work area. This includes any alias column names that you specified in the selection. Working with the CORRESPONDING FIELDS option of the INTO clause does not limit the amount of data read from the database, but only the amount of data that is read from the resulting set into the ABAP program. The only way of restricting the number of columns read is in the SELECT clause.

If you use the asterisk (*) option to read all of the columns of a single database table <dbtab> in the SELECT clause, the INTO clause may be empty. The SELECT statement then writes the data by default into the table work area with the same name <dbtab> as the database table itself. You **must** declare this table work area using the TABLES [Page 130] statement. Before Release 4.0, this was necessary before you could read data from a database table at all, and was frequently used as an implicit work area. Nowadays, table work areas are still useful as interface work areas, but should no longer be used as the work area in the SELECT statement. Like internal tables without header lines, having different names for the source and target areas makes programs clearer.

Specifying Internal Tables

When you read several lines of a database table, you can place them in an internal table. To do this, use the following in the INTO clause:

```
SELECT ... INTO|APPENDING [CORRESPONDING FIELDS OF] TABLE <itab>
                [PACKAGE SIZE <n>] ...
```

The same applies to the line type of <itab>, the way in which the data for a line of the database table are assigned to a table line, and the CORRESPONDING FIELDS addition as for flat work areas (see above).

The internal table is filled with all of the lines of the selection. When you use INTO, all existing lines in the table are deleted. When you use APPENDING; the new lines are added to the existing internal table <itab>. With APPENDING, the system adds the lines to the internal table appropriately for the table type. Fields in the internal table not affected by the selection are filled with initial values.

If you use the PACKAGE SIZE addition, the lines of the selection are not written into the internal table at once, but in packets. You can define packets of <n> lines that are written one after the other into the internal table. If you use INTO, each packet replaces the preceding one. If you use APPENDING, the packets are inserted one after the other. This is only possible in a loop that

ends with ENDSELECT. Outside the SELECT loop, the contents of the internal table are undetermined. You must process the selected lines within the loop.

Specifying Single Fields

If you specify single columns of the database table or aggregate expressions in the SELECT clause, you can read the data into single fields for a single entry or for multiple entries in a SELECT loop. To read data into single fields, use the following in the INTO clause:

SELECT ... INTO (<f_1>, <f_2>, ...). ...

You must specify as many individual fields <f_i> as are specified in the field list of the SELECT clause. The fields in the SELECT clause are assigned, from left to right, to the fields in the list in the INTO clause.

You do not have to use a field list in the INTO clause when you specify individual fields in the SELECT clause - you can also use the CORRESPONDING FIELDS addition to read the data into a flat work area or an internal table instead. In that case, the target area does not need to contain the same number of elements as the list in the SELECT clause.

Examples

Flat structure as target area

```
DATA WA TYPE SPFLI.

SELECT *
INTO   WA
FROM   SPFLI.

  WRITE: / WA-CARRID ...

ENDSELECT.
```

This example uses a flat structure with the same data type as the database table SPFLI as the target area in a SELECT loop. Within the loop, it is possible to address the contents of the individual columns.

```
DATA SPFLI TYPE SPFLI.

SELECT *
FROM   SPFLI.

  WRITE: / SPFLI-CARRID ...

ENDSELECT.
```

This example declares a structure SPFLI with the same name as the database table you want to read. This structure is used implicitly as the target area in the SELECT loop. Since the names are the same, it is possible to overlook the fact that you are working with an ABAP data object here and not the database table itself.

Internal table as target area

```
DATA: BEGIN OF WA,
         CARRID   TYPE SPFLI-CARRID,
```

```
          CONNID   TYPE SPFLI-CONNID,
          CITYFROM TYPE SPFLI-CITYFROM,
          CITYTO   TYPE SPFLI-CITYTO,
        END OF WA,
        ITAB LIKE SORTED TABLE OF WA
                  WITH NON-UNIQUE KEY CITYFROM CITYTO.

SELECT CARRID CONNID CITYFROM CITYTO
INTO   CORRESPONDING FIELDS OF TABLE ITAB
FROM   SPFLI.

IF SY-SUBRC EQ 0.
  WRITE: / SY-DBCNT, 'Connections'.
  SKIP.
  LOOP AT ITAB INTO WA.
    WRITE: / WA-CARRID, WA-CONNID, WA-CITYFROM, WA-CITYTO.
  ENDLOOP.
ENDIF.
```

The output is:

```
        10  Connections

LH   2402 FRANKFURT           BERLIN
LH   0400 FRANKFURT           NEW YORK
LH   0402 FRANKFURT           NEW YORK
UA   0941 FRANKFURT           SAN FRANCISCO
AA   0017 NEW YORK            SAN FRANCISCO
AZ   0555 ROME                FRANKFURT
QF   0005 SINGAPORE           FRANKFURT
SQ   0866 SINGAPORE           HONGKONG
SQ   0002 SINGAPORE           SAN FRANCISCO
AZ   0789 TOKYO               ROME
```

The target area is a sorted table ITAB containing four fields with the same names and data types as the database table SPFLI. The program uses the CORRESPONDING FIELDS addition to place the columns from the SELECT clause into the corresponding fields of the internal table. Because ITAB is a sorted table, the data is inserted into the table sorted by the table key of ITAB.

Reading packets into an internal table

```
DATA: WA   TYPE SPFLI,
      ITAB TYPE SORTED TABLE OF SPFLI
                WITH UNIQUE KEY CARRID CONNID.

SELECT CARRID CONNID
FROM   SPFLI
INTO   CORRESPONDING FIELDS OF TABLE ITAB
       PACKAGE SIZE 3.
```

```
  LOOP AT ITAB INTO WA.
    WRITE: / WA-CARRID, WA-CONNID.
  ENDLOOP.

  SKIP 1.

ENDSELECT.
```

The output is:

```
AA  0017
AZ  0555
AZ  0789

LH  0400
LH  0402
LH  2402

QF  0005
SQ  0002
SQ  0866

UA  0941
```

The example reads packets of three lines each into the sorted table ITAB. In each pass of the SELECT loop, the internal table has a different sorted content.

If you were to use APPENDING instead of INTO, the list would look like this:

```
AA  0017
AZ  0555
AZ  0789

AA  0017
AZ  0555
AZ  0789
LH  0400
LH  0402
LH  2402

AA  0017
AZ  0555
AZ  0789
LH  0400
LH  0402
LH  2402
QF  0005
SQ  0002
SQ  0866

AA  0017
AZ  0555
AZ  0789
LH  0400
LH  0402
LH  2402
QF  0005
SQ  0002
SQ  0866
UA  0941
```

In each loop pass, a new packet is sorted into the internal table.

Single fields as target area:

```
DATA:   AVERAGE TYPE P DECIMALS 2,
        SUM     TYPE P DECIMALS 2.

SELECT AVG( LUGGWEIGHT ) SUM( LUGGWEIGHT )
INTO   (AVERAGE, SUM)
FROM   SBOOK.

WRITE: / 'Average:', AVERAGE,
       / 'Sum    :', SUM.
```

The output is:

```
Average:             4,00
Sum    :        11.778,70
```

The SELECT clause contains two aggregate expressions for calculating the average and sum of the field LUGGWEIGHT from database table SBOOK. The target fields are called AVERAGE and SUM.

Using aliases:

```
DATA: BEGIN OF LUGGAGE,
        AVERAGE TYPE P DECIMALS 2,
        SUM     TYPE P DECIMALS 2,
      END OF LUGGAGE.

SELECT AVG( LUGGWEIGHT ) AS AVERAGE SUM( LUGGWEIGHT ) AS SUM
INTO   CORRESPONDING FIELDS OF LUGGAGE
FROM   SBOOK.

WRITE: / 'Average:', LUGGAGE-AVERAGE,
       / 'Sum    :', LUGGAGE-SUM.
```

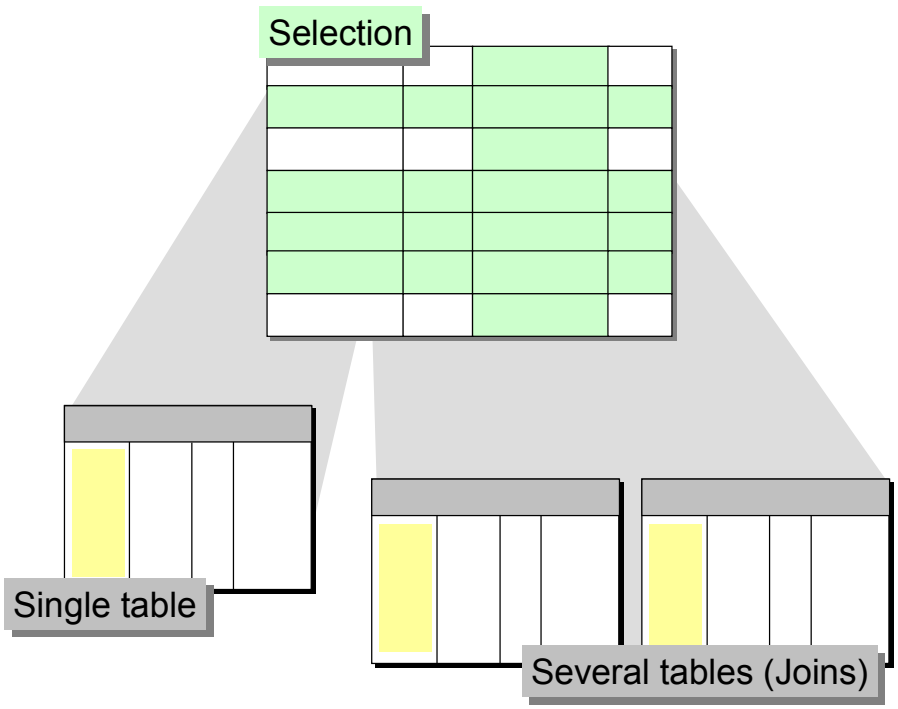
The output is:

```
Average:             4,00
Sum    :        11.778,70
```

This example has the same effect as the previous one. The only difference is that a structure is used as the target area instead of individual fields, and that the names of the structure components are used as aliases in the SELECT clause.

Specifying Database Tables

The FROM clause determines the database tables from which the data specified in the SELECT clause is read. You can specify either a single table or more than one table, linked using inner or outer joins. The names of database tables may be specified statically or dynamically, and you can use alias names. You can also use the FROM clause to bypass the SAP buffer and restrict the number of lines to be read from the database.



“Database table” can equally mean an ABAP Dictionary view. A view links two or more database tables in the ABAP Dictionary, providing a static join that is available systemwide. You can specify the name of a view wherever the name of a database table may occur in the FROM clause.

The FROM clause has two parts - one for specifying database tables, and one for other additions:

SELECT ... FROM <tables> <options> ...

In <tables>, you specify the names of database tables and define joins. <options> allows you to specify further additions that control the database access.

Specifying Database Tables Statically

To specify the name of a database table statically, use the following:

SELECT ... FROM <dbtab> [AS <alias>] <options> ...

The database table <dbtab> must exist in the ABAP Dictionary. The AS addition allows you to specify an alternative name <alias> that you can then use in the SELECT; FROM, WHERE, and GROUP BY clauses. This can eliminate ambiguity when you use more than one database table,

especially when you use a single database table more than once in a join. Once you have defined an alias, you may no longer use the real name of the database table

Specifying Database Tables Dynamically

To specify the name of a database table dynamically, use the following:

SELECT ... FROM (<name>) <options> ...

The field <name> must contain the name of a database table in the ABAP Dictionary. The table name must be written in uppercase. When you specify the name of a database table dynamically, you cannot use an empty INTO clause to read all of the columns into the work area <dbtab>. It is also not possible to use alternative table names.

Specifying Two or More Database Tables as an Inner Join

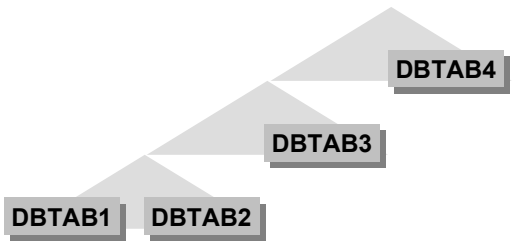
In a relational database, you normally need to read data simultaneously from more than one database table into an application program. You can read from more than one table in a single SELECT statement, such that the data in the tables all has to meet the same conditions, using the following join expression:

SELECT ...
...
FROM <tab> [INNER] JOIN <dbtab> [AS <alias>] ON <cond> <options>
...

where <dbtab> is a single database table and <tab> is either a table or another join expression. The database tables can be specified statically or dynamically as described above. You may also use aliases. You can enclose each join expression in parentheses. The INNER addition is optional.

A join expression links each line of <tab> with the lines in <dbtab> that meet the condition <cond>. This means that there is always one or more lines from the right-hand table that is linked to each line from the left-hand table by the join. If <dbtab> does not contain any lines that meet the condition <cond>, the line from <tab> is **not** included in the selection.

```
SELECT ...
...
FROM ( ( ( DBTAB1 INNER JOIN DBTAB2 ON ... )
                  INNER JOIN DBTAB3 ON ... )
                  INNER JOIN DBTAB4 ON ... )
...
```



The syntax of the <cond> condition is like that of the WHERE clause, although individual comparisons can only be linked using AND. Furthermore, each comparison must contain a

column from the right-hand table <dbtab>. It does not matter on which side of the comparison it occurs. For the column names in the comparison, you can use the same names that occur in the SELECT clause, to differentiate columns from different database tables that have the same names.

The comparisons in the condition <cond> can appear in the WHERE clause instead of the ON clause, since both clauses are applied equally to the temporary table containing all of the lines resulting from the join. However, each join must contain at least one comparison in the condition <cond>.

Specifying Two or More Database Tables as a Left Outer Join

In an inner join, a line from the left-hand database table or join is only included in the selection if there is one or more lines in the right-hand database table that meet the ON condition <cond>. The left outer join, on the other hand, reads lines from the left-hand database table or join even if there is no corresponding line in the right-hand table.

```
SELECT ...
...
FROM <tab> LEFT [OUTER] JOIN <dbtab> [AS <alias>] ON <cond> <options>
...
```

<tab> and <dbtab> are subject to the same rules and conditions as in an inner join. The OUTER addition is optional. The tables are linked in the same way as the inner join with the one exception that all lines selected from <tab> are included in the final selection. If <dbtab> does not contain any lines that meet the condition <cond>, the system includes a single line in the selection whose columns from <dbtab> are filled with null values.

In the left outer join, more restrictions apply to the condition <cond> than in the inner join. In addition to the above restrictions:

- EQ or = is the only permitted relational operator.
- There must be at least one comparison between columns from <tab> and <dbtab>.
- The WHERE clause may not contain any comparisons with columns from <dbtab>. All comparisons using columns from <dbtab> must appear in the condition <cond>.

Client Handling

As already mentioned, you can switch off the automatic client handling in Open SQL statements using a special addition. In the SELECT statement, the addition comes after the options in the FROM clause:

```
SELECT ... FROM <tables> CLIENT SPECIFIED ...
```

If you use this addition, you can then address the client fields in the individual clauses of the SELECT statement.

Disabling Data Buffering

If buffering is allowed for a table in the ABAP Dictionary, the SELECT statement always reads the data from the buffer in the database interface of the current application server. To read data directly from the database table instead of from the buffer, use the following:

```
SELECT ... FROM <tables> BYPASSING BUFFER ...
```

This addition guarantees that the data you read is the most up to date. However, as a rule, only data that does not change frequently should be buffered, and using the buffer where appropriate improves performance. You should therefore only use this option where really necessary.

Restricting the Number of Lines

To restrict the absolute number of lines included in the selection, use the following:

SELECT ... FROM <tables> UP TO <n> ROWS ...

If <n> is a positive integer, the system reads a maximum of <n> lines. If <n> is zero, the system reads all lines that meet the selection criteria. If you use the ORDER BY clause as well, the system reads all lines belonging to the selection, sorts them, and then places the first <n> lines in the selection set.

Examples

Specifying a database table statically:

```
DATA WA TYPE SCARR.

SELECT *
INTO   WA
FROM   SCARR UP TO 4 ROWS.

  WRITE: / WA-CARRID, WA-CARRNAME.

ENDSELECT.
```

The output is:

```
AA  American Airlines
AC  Air Canada
AF  Air France
AZ  Alitalia
```

The system reads four lines from the database table SCARR.

Specifying a database table dynamically:

```
DATA WA TYPE SCARR.

DATA NAME(10) VALUE 'SCARR'.

SELECT *
INTO   WA
FROM   (NAME) CLIENT SPECIFIED
WHERE  MANDT = '000'.

  WRITE: / WA-CARRID, WA-CARRNAME.

ENDSELECT.
```

A condition for the MANDT field is allowed, since the example uses the CLIENT SPECIFIED option. If NAME had contained the value ‘scarr’ instead of ‘SCARR’, a runtime error would have occurred.

Inner join:

```
DATA: BEGIN OF WA,
        CARRID TYPE SPFLI-CARRID,
        CONNID TYPE SPFLI-CONNID,
        FLDATE TYPE SFLIGHT-FLDATE,
        BOOKID TYPE SBOOK-BOOKID,
      END OF WA,
      ITAB LIKE SORTED TABLE OF WA
                WITH UNIQUE KEY CARRID CONNID FLDATE BOOKID.

SELECT P~CARRID P~CONNID F~FLDATE B~BOOKID
INTO   CORRESPONDING FIELDS OF TABLE ITAB
FROM   ( ( SPFLI AS P
           INNER JOIN SFLIGHT AS F ON P~CARRID = F~CARRID AND
                                      P~CONNID =
F~CONNID      )
           INNER JOIN SBOOK   AS B ON B~CARRID = F~CARRID AND
                                      B~CONNID = F~CONNID AND
                                      B~FLDATE =
F~FLDATE      )
WHERE P~CITYFROM = 'FRANKFURT' AND
      P~CITYTO   = 'NEW YORK'  AND
      F~SEATSMAX > F~SEATSOCC.

LOOP AT ITAB INTO WA.
  AT NEW FLDATE.
    WRITE: / WA-CARRID, WA-CONNID, WA-FLDATE.
  ENDAT.
  WRITE / WA-BOOKID.
ENDLOOP.
```

This example links the columns CARRID, CONNID, FLDATE, and BOOKID of the table SPFLI, SFLIGHT, and SBOOK, and creates a list of booking numbers for all flights from Frankfurt to New York that are not fully booked. An alias name is assigned to each table.

Left outer join:

```
DATA: BEGIN OF WA,
        CARRID   TYPE SCARR-CARRID,
        CARRNAME TYPE SCARR-CARRNAME,
        CONNID   TYPE SPFLI-CONNID,
      END OF WA,
      ITAB LIKE SORTED TABLE OF WA
                WITH NON-UNIQUE KEY CARRID.

SELECT S~CARRID S~CARRNAME P~CONNID
INTO   CORRESPONDING FIELDS OF TABLE ITAB
FROM   SCARR AS S
       LEFT OUTER JOIN SPFLI AS P ON S~CARRID   =  P~CARRID
```

```
AND
                                          P~CITYFROM = 'FRANKFURT'.

LOOP AT ITAB INTO WA.
  WRITE: / WA-CARRID, WA-CARRNAME, WA-CONNID.
ENDLOOP.
```

The output might look like this:

```
AA  American Airlines    0000
AC  Air Canada           0000
AF  Air France           0000
AZ  Alitalia             0000
BA  British Airways      0000
BL  Pacific Airlines     0000
CO  Continental Airlines 0000
DL  Delta Airlines       0000
FC  Berliner Spez. Flug  0000
LH  Lufthansa            0400
LH  Lufthansa            2402
LH  Lufthansa            0402
NG  Lauda Air            0000
NU  Japan Transocean Air 0000
NW  Northwest Airlines   0000
QF  Qantas Airways       0000
SA  South African Air.   0000
SQ  Singapore Airlines   0000
SR  Swissair             0000
UA  Unites Airlines      0941
```

The example links the columns CARRID, CARRNAME, and CONNID of the tables SCARR and SPFLI using the condition in the left outer join that the airline must fly from Frankfurt. All other airlines have a null value in the CONNID column in the selection.

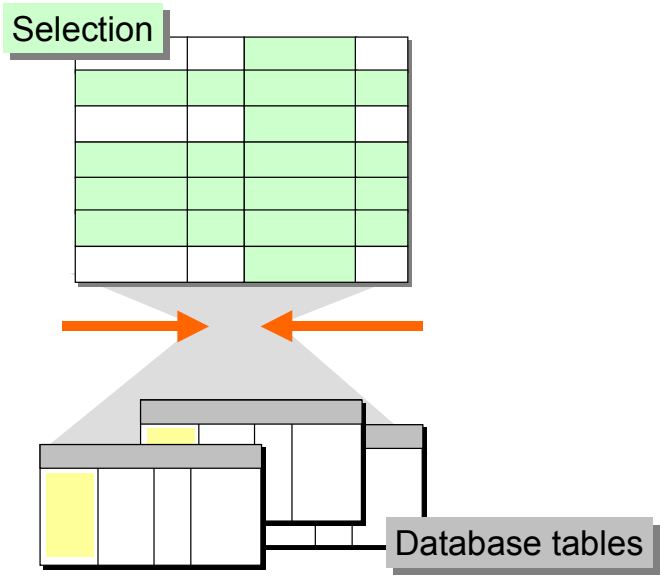
If the left outer join is replaced with an inner join, the list looks like this:

```
LH  Lufthansa            0400
LH  Lufthansa            2402
LH  Lufthansa            0402
UA  Unites Airlines      0941
```

Only lines that fulfill the ON condition are included in the selection.

Selecting Lines

The WHERE clause restricts the number of lines selected by specifying conditions that must be met.



As well as in the SELECT statement, the WHERE clause is also used in the OPEN CURSOR, UPDATE, and DELETE statements. The general form of the WHERE clause is:

SELECT ... WHERE <cond> ...

The <cond> conditions in the WHERE clause can be comparisons or a series of other special expressions. You can combine a series of conditions into a single condition. Conditions may also be programmed dynamically.

The conditions <cond> in the WHERE clause are often like logical expressions [Page 225], but not identical, since the syntax and semantics follow that of Standard SQL. In the conditions in the WHERE clause, you name columns using a field name as in the SELECT clause. In the following descriptions, <s> always represents a column of one of the database tables named in the FROM clause. The result of a condition may be true, false, or unknown. A line is only selected if the condition is true for it. A condition is unknown if one of the columns involved contains a null value.

Comparisons for All Types

To compare the value of a column of any data type with another value, use the following:

SELECT ... WHERE <s> <operator> <f> ...

<f> can be another column in a database table from the FROM clause, a data object, or a scalar subquery [Page 1080].

You can use the following expressions for the relational operator:

<operator>	Meaning
EQ	equal to

=	equal to
NE	not equal to
<>	not equal to
><	not equal to
LT	less than
<	less than
LE	less than or equal to
<=	less than or equal to
GT	greater than
>	greater than
GE	greater than or equal to
>=	greater than or equal to

The values of the operands are converted if necessary. The conversion may be dependent on the platform and codepage.

Values in Intervals

To find out whether the value of a column lies within a particular interval, use:

SELECT ... WHERE <s> [NOT] BETWEEN <f_1> AND <f_2> ...

The condition is true if the value of column <s> is [not] between the values of the data objects <f_1> and <f_2>. You cannot use BETWEEN in the ON condition of the FROM clause.

Comparing Strings

To find out whether the value of a column matches a pattern, use:

SELECT ... WHERE <s> [NOT] LIKE <f> [ESCAPE <h>] ...

The condition is true if the value of the column <s> matches [does not match] the pattern in the data object <f>. You can only use this test for text fields. The data type of the column must be alphanumeric. <f> must have data type C.

You can use the following wildcard characters in <f>:

- % for a sequence of any characters (including spaces).
- _ for a single character.

For example, ABC_EFG% matches the strings ABCxEFGxyz and ABCxEFG, but not ABCEFGxyz. If you want to use the two wildcard characters explicitly in the comparison, use the ESCAPE option. ESCAPE <h> specifies an escape symbol <h>. If preceded by <h>, the wildcards and the escape symbol itself lose their usual function within the pattern <f>. The use of _ and % corresponds to Standard SQL usage. Logical expressions elsewhere in ABAP use other wildcard characters (+ and *).

You cannot use LIKE in the ON condition of the FROM clause.

Checking Lists of Values

To find out whether the value of a column is contained in a list of values, use:

SELECT ... WHERE <s> [NOT] IN (<f_1>, ..., <f_n>) ...

The condition is true if the value of column <s> is [not] in the list <f_1> ... <f_n>.

Checking Subqueries

To find out whether the value of a column is contained in a scalar subquery [Page 1080], use:

SELECT ... WHERE <s> [NOT] IN <subquery> ...

The condition is true if the value of <s> is [not] contained in the results set of the scalar subquery <subquery>.

To find out whether the selection of a subquery [Page 1080] contains lines at all, use:

SELECT ... WHERE [NOT] EXISTS <subquery> ...

This condition is true if the result set of the subquery <subquery> contains at least one [no] line. The subquery does not have to be scalar.

You cannot check a subquery in the ON condition of the FROM clause.

Checking Selection Tables

To find out whether the value of a column satisfies the conditions in a selection table [Page 704], use:

SELECT ... WHERE <s> [NOT] IN <seltab> ...

The condition is true if the value of <s> [does not] satisfy the conditions stored in <seltab>. <seltab> can be either a real selection table or a RANGES table. You cannot check a selection table in the ON condition of the FROM clause.

Checking for Null Values

To find out whether the value of a column is null, use:

SELECT ... WHERE <s> IS [NOT] NULL ...

The condition is true if the value of <s> is [not] null.

Negating Conditions

To negate the result of a condition, use:

SELECT ... WHERE NOT <cond> ...

The condition is true if <cond> is false, and false if <cond> is true. The result of an unknown condition remains unknown when negated.

Linking Conditions

You can combine two conditions into one using the AND and OR operators:

SELECT ... WHERE <$cond_1$> AND <$cond_2$> ...

This condition is true if <$\mathbf{cond_1}$> and <$\mathbf{cond_2}$> are true.

SELECT ... WHERE <$cond_1$> OR <$cond_2$> ...

This condition is true if one or both of <$\mathbf{cond_1}$> and <$\mathbf{cond_2}$> are true.

NOT takes priority over AND, and AND takes priority over OR. However, you can also control the processing sequence using parentheses.

Dynamic Conditions

To specify a condition dynamically, use:

SELECT ... WHERE (<itab>) ...

where <itab> is an internal table with line type C and maximum length 72 characters. All of the conditions listed above except for selection tables, can be written into the lines of <itab>. However, you may only use literals, and not the names of data objects. The internal table can also be left empty.

If you only want to specify a part of the condition dynamically, use:

SELECT ... WHERE <cond> AND (<itab>) ...

You cannot link a static and a dynamic condition using OR.

You may only use dynamic conditions in the WHERE clause of the SELECT statement.

Tabular Conditions

The WHERE clause of the SELECT statement has a special variant that allows you to derive conditions from the lines and columns of an internal table:

SELECT ... FOR ALL ENTRIES IN <itab> WHERE <cond> ...

<cond> may be formulated as described above. If you specify a field of the internal table <itab> as an operand in a condition, you address all lines of the internal table. The comparison is then performed for each line of the internal table. For each line, the system selects the lines from the database table that satisfy the condition. The result set of the SELECT statement is the union of the individual selections for each line of the internal table. Duplicate lines are automatically eliminated from the result set. If <itab> is empty, the addition FOR ALL ENTRIES is disregarded, and all entries are read.

The internal table <itab> must have a structured line type, and each field that occurs in the condition <cond> must be compatible with the column of the database with which it is compared. Do not use the operators LIKE, BETWEEN, and IN in comparisons using internal table fields. You may not use the ORDER BY clause in the same SELECT statement.

You can use the option FOR ALL ENTRIES to replace nested select loops by operations on internal tables. This can significantly improve the performance for large sets of selected data.

Examples

Conditions in the WHERE clause:

... WHERE CARRID = 'UA'.

This condition is true if the column CARRID has the contents UA.

... WHERE NUM GE 15.

This condition is true if the column NUM contains numbers greater than or equal to 15.

... WHERE CITYFROM NE 'FRANKFURT'.

This condition is true if the column CITYFROM does not contain the string FRANKFURT.

... WHERE NUM BETWEEN 15 AND 45.

This condition is true if the column NUM contains numbers between 15 and 45.

... WHERE NUM NOT BETWEEN 1 AND 99.

This condition is true if the column NUM contains numbers not between 1 and 99.

... WHERE NAME NOT BETWEEN 'A' AND 'H'.

This condition is true if the column NAME is one character long and its contents are not between A and H.

... WHERE CITY LIKE '%town%'.

This condition is true if the column CITY contains a string containing the pattern ‘town’.

... WHERE NAME NOT LIKE '_n%'.

This condition is true if the column NAME contains a value whose second character is not ‘n’.

... WHERE FUNCNAME LIKE 'EDIT#_%' ESCAPE '#'.

This condition is true if the contents of the column FUNCNAME begin with EDIT_.

... WHERE CITY IN ('BERLIN', 'NEW YORK', 'LONDON').

This condition is true if the column CITY contains one of the values BERLIN, NEW YORK, or LONDON.

... WHERE CITY NOT IN ('FRANKFURT', 'ROME').

This condition is true if the column CITY does not contain the values FRANKFURT or ROME.

... WHERE (NUMBER = '0001' OR NUMBER = '0002') AND
NOT (COUNTRY = 'F' OR COUNTRY = 'USA').

This condition is true if the column NUMBER contains the value 0001 or 0002 and the column COUNTRY contains neither F nor USA.

Dynamic conditions

```
DATA: COND(72) TYPE C,
      ITAB LIKE TABLE OF COND.

PARAMETERS: CITY1(10) TYPE C, CITY2(10) TYPE C.

DATA WA TYPE SPFLI-CITYFROM.

CONCATENATE 'CITYFROM = ''' CITY1 '''' INTO COND.
APPEND COND TO ITAB.
CONCATENATE 'OR CITYFROM = ''' CITY2 '''' INTO COND.
APPEND COND TO ITAB.
CONCATENATE 'OR CITYFROM = ''' 'BERLIN' '''' INTO COND.
APPEND COND TO ITAB.

LOOP AT ITAB INTO COND.
  WRITE COND.
ENDLOOP.
```

```
SKIP.

SELECT CITYFROM
INTO   WA
FROM   SPFLI
WHERE  (ITAB).

  WRITE / WA.

ENDSELECT.
```

If the user enters FRANKFURT and BERLIN for the parameters CITY1 and CITY2 on the selection screen, the list display is as follows:

```
CITYFROM = 'FRANKFURT'
OR CITYFROM = 'BERLIN'
OR CITYFROM = 'BERLIN'

FRANKFURT
FRANKFURT
FRANKFURT
BERLIN
FRANKFURT
FRANKFURT
```

The first three lines show the contents of the internal table ITAB. Exactly the corresponding table lines are selected.

Tabular conditions

```
DATA: BEGIN OF LINE,
        CARRID   TYPE SPFLI-CARRID,
        CONNID   TYPE SPFLI-CONNID,
        CITYFROM TYPE SPFLI-CITYFROM,
        CITYTO   TYPE SPFLI-CITYTO,
      END OF LINE,
      ITAB LIKE TABLE OF LINE.

LINE-CITYFROM = 'FRANKFURT'.
LINE-CITYTO   = 'BERLIN'.
APPEND LINE TO ITAB.

LINE-CITYFROM = 'NEW YORK'.
LINE-CITYTO   = 'SAN FRANCISCO'.
APPEND LINE TO ITAB.

SELECT CARRID CONNID CITYFROM CITYTO
INTO   CORRESPONDING FIELDS OF LINE
FROM   SPFLI
FOR ALL ENTRIES IN ITAB
WHERE  CITYFROM = ITAB-CITYFROM AND CITYTO = ITAB-CITYTO.

  WRITE: / LINE-CARRID, LINE-CONNID, LINE-CITYFROM, LINE-
CITYTO.

ENDSELECT.
```

The output is as follows:

```
AA  0017 NEW YORK            SAN FRANCISCO
DL  1699 NEW YORK            SAN FRANCISCO
LH  2402 FRANKFURT           BERLIN
```

This example selects all lines in which the following conditions are fulfilled:

- The CITYFROM column contains FRANKFURT and the CITYTO column contains BERLIN.
- The CITYFROM column contains NEW YORK and the CITYTO column contains SAN FRANCISCO.

Tabular conditions

```
DATA: TAB_SPFLI   TYPE TABLE OF SPFLI,
      TAB_SFLIGHT TYPE SORTED TABLE OF SFLIGHT
                      WITH UNIQUE KEY TABLE LINE,
      WA LIKE LINE OF TAB_SFLIGHT.

SELECT CARRID CONNID
INTO   CORRESPONDING FIELDS OF TABLE TAB_SPFLI
FROM   SPFLI
WHERE  CITYFROM  = 'NEW YORK'.

SELECT CARRID CONNID FLDATE
INTO   CORRESPONDING FIELDS OF TABLE TAB_SFLIGHT
FROM   SFLIGHT
FOR ALL ENTRIES IN TAB_SPFLI
WHERE  CARRID = TAB_SPFLI-CARRID AND
       CONNID = TAB_SPFLI-CONNID.

LOOP AT TAB_SFLIGHT INTO WA.

  AT NEW CONNID.
    WRITE: / WA-CARRID, WA-CONNID.
  ENDAT.

  WRITE: / WA-FLDATE.

ENDLOOP.
```

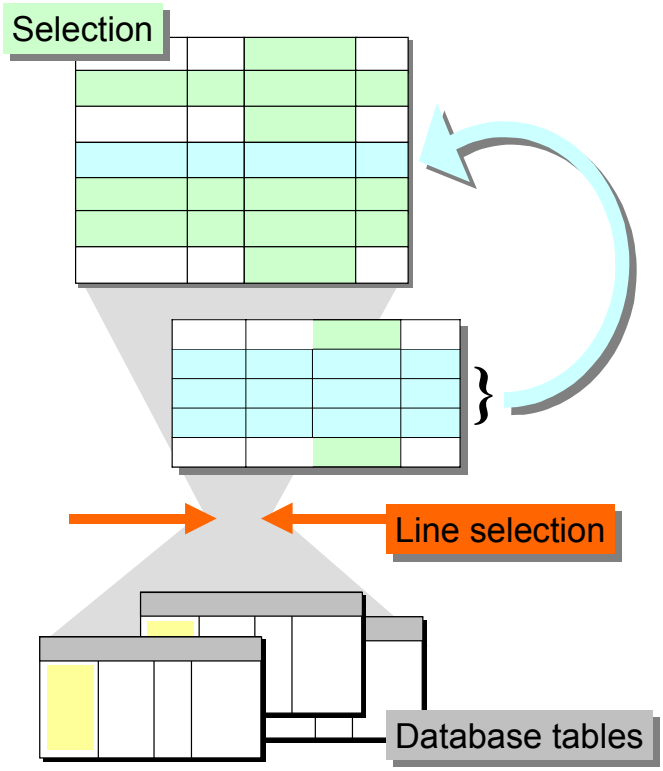
The output is as follows:

```
AA  0017
1998/08/28
1998/09/30
1998/11/19
1998/11/22
1998/11/29
1998/12/19
1998/12/21
DL  1699
1998/08/28
1998/09/30
1998/11/19
1998/11/22
1998/11/29
1998/12/19
1998/12/21
```

This example selects flight data from SFLIGHT for all connections for which the column CITYFROM in table SPFLI has the value NEW YORK. You could also use a join in the FROM clause to select the same data in a single SELECT statement.

Grouping Lines

The GROUP BY clause summarizes several lines from the database table into a single line of the selection.



The GROUP BY clause allows you to summarize lines that have the same content in particular columns. Aggregate functions are applied to the other columns. You can specify the columns in the GROUP BY clause either statically or dynamically.

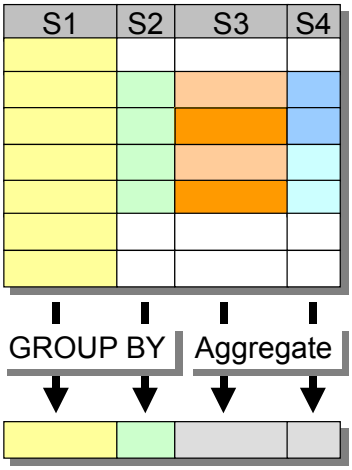
Specifying Columns Statically

To specify the columns in the GROUP BY clause statically, use:

SELECT <lines> <s_1> [AS <a_1>] <s_2> [AS <a_2>] ...
<agg> <s_m> [AS <a_m>] <agg> <s_n> [AS <a_n>] ...
...
GROUP BY <s_1> <s_2> ....

To use the GROUP BY clause, you must specify all of the relevant columns in the SELECT clause. In the GROUP BY clause, you list the field names of the columns whose contents must be the same. You can only use the field names as they appear in the database table. Alias names from the SELECT clause are not allowed.

All columns of the SELECT clause that are not listed in the GROUP BY clause must be included in aggregate functions. This defines how the contents of these columns is calculated when the lines are summarized.



Specifying Columns Dynamically

To specify the columns in the GROUP BY clause dynamically, use:

... GROUP BY (<itab>) ...

where <itab> is an internal table with line type C and maximum length 72 characters containing the column names $<s_1>$ $<s_2>$ .....

Example

```
DATA: CARRID TYPE SFLIGHT-CARRID,
      MINIMUM TYPE P DECIMALS 2,
      MAXIMUM TYPE P DECIMALS 2.

SELECT   CARRID MIN( PRICE ) MAX( PRICE )
INTO     (CARRID, MINIMUM, MAXIMUM)
FROM     SFLIGHT
GROUP BY CARRID.

  WRITE: / CARRID, MINIMUM, MAXIMUM.

ENDSELECT.
```

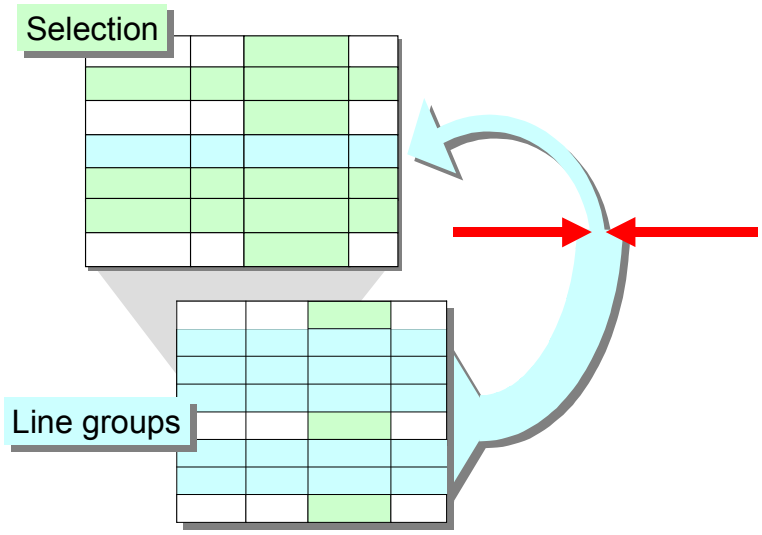
The output is as follows:

```
AA              513,69              513,69
AZ            3.602,02           26.674,45
DL              513,69              513,69
LH              485,00            1.332,00
QF            1.577,29            1.577,29
SQ              487,05            1.180,15
UA            1.068,00            1.068,00
```

The lines in the database table SFLIGHT that have the same value in the column CARRID are summarized. The smallest and largest values of PRICE are determined for each group and placed in the summarized line.

Selecting Groups of Lines

The HAVING clause uses conditions to restrict the number of groups selected.



You can only use the HAVING clause in conjunction with the GROUP BY clause.

To select line groups, use:

```
SELECT <lines> <s1> [AS <a1>] <s2> [AS <a2>] ...
       <agg> <sm> [AS <am>] <agg> <sn> [AS <an>] ...
  ...
  GROUP BY <s1> <s2> ....
  HAVING <cond>.
```

The conditions <cond> that you can use in the HAVING clause are the same as those in the SELECT clause [Page 1064], with the restrictions that you can only use columns from the SELECT clause, and not all of the columns from the database tables in the FROM clause. If you use an invalid column, a runtime error results.

On the other hand, you can enter aggregate expressions for **all** columns read from the database table that do not appear in the GROUP BY clause. This means that you can use aggregate expressions, even if they do not appear in the SELECT clause. You cannot use aggregate expressions in the conditions in the WHERE clause.

As in the WHERE clause, you can specify the conditions in the HAVING clause as the contents of an internal table with line type C and length 72.

Example

```
DATA WA TYPE SFLIGHT.

SELECT   CONNID
INTO     WA-CONNID
FROM     SFLIGHT
```

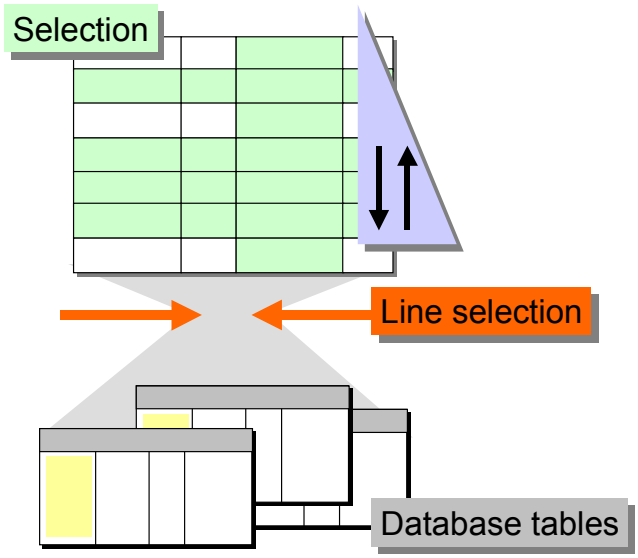
Selecting Groups of Lines

```
WHERE    CARRID = 'LH'
GROUP BY CONNID
HAVING   SUM( SEATSOCC ) > 300.

  WRITE: / WA-CARRID, WA-CONNID.

ENDSELECT.
```

This example selects groups of lines from database table SFLIGHT with the value ‘LH’ for CARRID and identical values of CONNID. The groups are then restricted further by the condition that the sum of the contents of the column SEATSOCC for a group must be greater than 300.

Specifying a Sort Order

The ORDER BY clause sorts the lines in the selection according to the contents of their columns.



If you do not use the ORDER BY clause, the sequence of the lines in the selection is indeterminate, and can vary each time the SELECT statement is executed. You can sort the selection by any column (not necessarily those of the primary key), and specify the columns either statically or dynamically.

Sorting by the Primary Key

To sort the selection set in ascending order by the primary key, use the following:

```
SELECT <lines> *
...
... ORDER BY PRIMARY KEY.
```

This sorting method is only possible if you use an asterisk (*) in the SELECT clause to select all columns. Furthermore, it only works if you specify a single database table in the FROM clause. You cannot use views or joins, since neither has a defined primary key.

Sorting by any Columns

To sort the lines in the selection set by any columns, use the following:

```
SELECT ...
...
ORDER BY <s1> [ASCENDING|DESCENDING]
         <s2> [ASCENDING|DESCENDING] ...
```

The lines are sorted by the columns <s_1>, <s_2>, ... You determine the direction of the sort using one of the additions ASCENDING or DESCENDING. The default is ascending order. The sort order depends on the sequence in which you list the columns.

Specifying a Sort Order

You can use either the names of the columns as specified in the SELECT clause or their alias names. You may only use columns that occur in the SELECT clause. By using alias names for aggregate expressions, you can use them as sort fields.

Specifying the Columns Dynamically

To specify the columns in the ORDER BY clause dynamically, use:

SELECT ...
...
ORDER BY (<itab>).

where <itab> is an internal table with line type C and maximum length 72 characters containing the column names $<s_1>$ $<s_2>$ .....

Example

```
DATA: BEGIN OF WA,
        CARRID TYPE SFLIGHT-CARRID,
        CONNID TYPE SFLIGHT-CONNID,
        MIN    TYPE I,
      END OF WA.

SELECT   CARRID CONNID MIN( SEATSOCC ) AS MIN
INTO     CORRESPONDING FIELDS OF WA
FROM     SFLIGHT
GROUP BY CARRID CONNID
ORDER BY CARRID MIN DESCENDING.

  WRITE: / WA-CARRID, WA-CONNID, WA-MIN.

ENDSELECT.
```

The output is as follows:

```
AA  0064        186
AA  0017         95
AZ  0790        233
AZ  0555        205
AZ  0788         96
AZ  0789         79
DL  1984         97
DL  1699         73
LH  0400        233
LH  0402        141
LH  2407        119
LH  2402         97
QF  0005        205
QF  0006         97
SQ  0158        177
SQ  0866         94
SQ  0988         75
SQ  0002         73
UA  0941        236
UA  3504        142
```

The lines of the database table SFLIGHT are grouped according to the columns CARRID and CONNID, and the system finds the minimum value of the column SEATSOCC for each group. The selection is sorted by CARRID in ascending order and by the minimum value of SEATSOCC in descending order, using the alias name MIN for the aggregate expression.

Subqueries

A subquery is a special SELECT statement containing a subquery within particular conditions of the WHERE or HAVING clauses. You cannot use them in the ON condition of the FROM clause. Their syntax is:

```
( SELECT    <result>
  FROM      <source>
  [WHERE    <condition>]
  [GROUP BY <fields>]
  [HAVING   <cond>]           )
```

As you can see, this syntax is restricted in comparison with the full SELECT statement, since it contains no INTO or ORDER BY clause.

You can nest subqueries, that is, the WHERE and HAVING clauses of subqueries can themselves contain a subquery. When a nested subquery in the WHERE clause uses fields from the previous query, it is known as a correlated query. The subquery is then processed for each line of the database table that satisfies the previous condition.

Scalar Subqueries

In a scalar subquery, the selection in the SELECT clause is restricted to one column or aggregate expression. The expression <result> of the SELECT clause is:

... <line> [<agg>] <s>

You can only enter a single field in the SELECT clause.

Subqueries in Conditions

A non-scalar subquery can only have a WHERE or HAVING clause in the [NOT] EXISTS <subquery> condition.

This condition is true if the result set of the subquery contains at least one [no] line.

Scalar Subqueries in Conditions

As well as in the above condition, you can also use scalar subqueries in further conditions.

Checking a Value of the Subquery

The following is a possible condition with scalar subqueries:

... <s> [NOT] IN <subquery>

The condition is true if the value of <s> is [not] contained in the results set of the scalar subquery <subquery>.

Scalar Subqueries in Comparisons

The other conditions can all be comparisons whose operators are contained in the table for comparisons with all types in the WHERE clause [Page 1064]. There is a difference, depending on whether the subquery selection contains one or more lines.

Single-line Subquery

If the selection in the subquery only contains one line, use the following for the comparison:

... <s> <operator> <subquery> ...

The value of <s> is compared with the value in the selection from the subquery. The condition is either true of false.

The subquery may only contain one line, otherwise a runtime error occurs. You can make sure that the subquery only returns one line by using the SINGLE expression in the SELECT clause.

Multiple-line Subquery

If the selection from the subquery returns more than one line, you must write the comparison as follows:

... <s> <operator> ALL|ANY|SOME <subquery> ...

If you use the ALL prefix, the condition is only true if the comparison is true for all lines in the subquery. If you use the ANY or SOME prefix, the condition is only true if the comparison is true for at least one line of the subquery. The equality operator (= or EQ) in conjunction with ANY or SOME has the same effect as the IN operator for checking a value.

If the selection from the subquery contains several lines and you do not use the ALL, ANY, or SOME expression, a runtime error occurs.

Examples

Correlated, non-scalar subquery:

```
DATA: NAME_TAB TYPE TABLE OF SCARR-CARRNAME,
      NAME  LIKE LINE OF NAME_TAB.

SELECT CARRNAME
INTO   TABLE NAME_TAB
FROM   SCARR
WHERE  EXISTS ( SELECT *
                FROM   SPFLI
                WHERE  CARRID   =  SCARR~CARRID AND
                       CITYFROM = 'NEW YORK'         ).

LOOP AT NAME_TAB INTO NAME.
  WRITE: / NAME.
ENDLOOP.
```

This example selects all lines from database table SCARR for airlines that fly from New York.

Scalar subquery:

```
DATA: CARR_ID TYPE SPFLI-CARRID VALUE 'LH',
      CONN_ID TYPE SPFLI-CONNID VALUE '0400'.

DATA: CITY  TYPE SGEOCITY-CITY,
      LATI  TYPE P DECIMALS 2,
      LONGI TYPE P DECIMALS 2.

SELECT SINGLE CITY LATITUDE LONGITUDE
INTO   (CITY, LATI, LONGI)
```

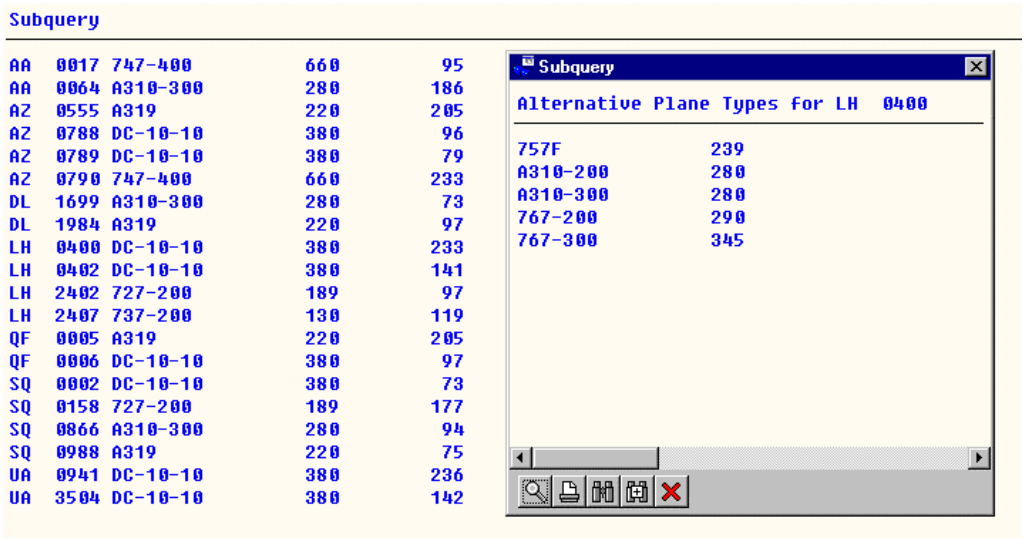
```
FROM   SGEOCITY
WHERE  CITY IN ( SELECT CITYFROM
                 FROM   SPFLI
                 WHERE  CARRID = CARR_ID AND
                        CONNID = CONN_ID      ).

WRITE: CITY, LATI, LONGI.
```

This example reads the latitude and longitude of the departure city of flight LH 402 from database table SGEOCITY.

Scalar subquery:

```
DATA: WA TYPE SFLIGHT,
      PLANE LIKE WA-PLANETYPE,
      SEATS LIKE WA-SEATSMAX.

SELECT   CARRID CONNID PLANETYPE SEATSMAX MAX( SEATSOCC )
INTO     (WA-CARRID, WA-CONNID, WA-PLANETYPE,
          WA-SEATSMAX, WA-SEATSOCC)
FROM     SFLIGHT
GROUP BY CARRID CONNID PLANETYPE SEATSMAX
ORDER BY CARRID CONNID.

  WRITE: /  WA-CARRID,
            WA-CONNID,
            WA-PLANETYPE,
            WA-SEATSMAX,
            WA-SEATSOCC.

  HIDE: WA-CARRID, WA-CONNID, WA-SEATSMAX.

ENDSELECT.

AT LINE-SELECTION.

  WINDOW STARTING AT 45 3 ENDING AT 85 13.

  WRITE: 'Alternative Plane Types',
         'for', WA-CARRID, WA-CONNID.

  ULINE.

  SELECT PLANETYPE SEATSMAX
  INTO   (PLANE, SEATS)
  FROM   SAPLANE AS PLANE
  WHERE  SEATSMAX < WA-SEATSMAX AND
         SEATSMAX >= ALL ( SELECT SEATSOCC
                           FROM   SFLIGHT
                           WHERE  CARRID = WA-CARRID AND
                                  CONNID = WA-CONNID     )
  ORDER BY SEATSMAX.

    WRITE: / PLANE, SEATS.

  ENDSELECT.
```

The list output, after double-clicking a line, looks like this:



The detail list displays all aircraft types that have fewer seats than the currently-allocated aircraft type, but enough to carry all of the passengers currently booked on the flight.

Using a Cursor to Read Data

In the normal SELECT statement, the data from the selection is always read directly into the target area specified in the INTO clause during the SELECT statement. When you use a cursor to read data, you decouple the process from the SELECT statement. To do this, you must open a cursor for a SELECT statement. Afterwards, you can place the lines from the selection into a flat target area.

Opening and Closing Cursors

To open a cursor for a SELECT statement, use the following:

```
OPEN CURSOR [WITH HOLD] <c> FOR SELECT    <result>
                        FROM      <source>
                        [WHERE    <condition>]
                        [GROUP BY <fields>]
                        [HAVING   <cond>]
                        [ORDER BY <fields>].
```

You must first have declared the cursor <c> using the DATA statement and the special data type CURSOR. You can use all clauses of the SELECT statement apart from the INTO clause. Furthermore, you can only formulate the SELECT clause [Page 1045] so that the selection consists of more than one line. This means that you may not use the SINGLE addition, and that the column selection may not contain only aggregate expressions.

An open cursor points to an internal handler, similarly to a reference variable pointing to an object. You can reassign cursors so that more than one points to the same handler. In a MOVE statement, the target cursor adopts all of the attributes of the source cursor, namely its position, and all of the clauses in the OPEN CURSOR statement.

You can also open more than one cursor in parallel for a single database table. If a cursor is already open, you cannot reopen it. To close a cursor explicitly, use the following statement:

```
CLOSE CURSOR <c>.
```

You should use this statement to close all cursors that you no longer require, since only a limited number of cursors may be open simultaneously. With one exception, a database LUW [Page 1262] is concluded when you close a cursor either explicitly or implicitly. The WITH HOLD addition in the OPEN CURSOR statement allows you to prevent a cursor from being closed when a database commit occurs in Native SQL [Page 1114].

Reading Data

An open cursor is linked to a multiple-line selection in the database table. To read the data into a target area in the ABAP program, use the following:

```
FETCH NEXT CURSOR <c> INTO <target>.
```

This writes one line of the selection into the target area <target>, and the cursor moves one line further in the selection set. The fetch statement decouples the INTO clause [Page 1052] from the other clauses in the SELECT statement. The target area <target> must be a flat-structured variable, since only single lines can be passed. Otherwise, the rules for the INTO clause are the same as in the SELECT statement.

If the cursor is on a list line, the system sets SY-SUBRC to 0, otherwise to 4. After a FETCH statement, the system field SY-DBCNT contains the number of lines already read by the current cursor.

Examples

```
DATA: C1 TYPE CURSOR,
      C2 TYPE CURSOR.

DATA: WA1 TYPE SPFLI,
      WA2 TYPE SPFLI.

DATA: FLAG1,
      FLAG2.

OPEN CURSOR: C1 FOR SELECT CARRID CONNID
                    FROM   SPFLI
                    WHERE  CARRID = 'LH',

             C2 FOR SELECT CARRID CONNID CITYFROM CITYTO
                    FROM   SPFLI
                    WHERE  CARRID = 'AZ'.

DO.
  IF FLAG1 NE 'X'.
    FETCH NEXT CURSOR C1 INTO CORRESPONDING FIELDS OF WA1.
    IF SY-SUBRC <> 0.
      CLOSE CURSOR C1.
      FLAG1 = 'X'.
    ELSE.
      WRITE: / WA1-CARRID, WA1-CONNID.
    ENDIF.
  ENDIF.
  IF FLAG2 NE 'X'.
    FETCH NEXT CURSOR C2 INTO CORRESPONDING FIELDS OF WA2.
    IF SY-SUBRC <> 0.
      CLOSE CURSOR C2.
      FLAG2 = 'X'.
    ELSE.
      WRITE: / WA2-CARRID, WA2-CONNID,
               WA2-CITYFROM, WA2-CITYTO.
    ENDIF.
  ENDIF.
  IF FLAG1 = 'X' AND FLAG2 = 'X'.
    EXIT.
  ENDIF.
ENDDO.
```

The output is as follows:

```
LH  0400
AZ  0555 ROME                FRANKFURT
LH  0402
AZ  0788 ROME                TOKYO
LH  2402
AZ  0789 TOKYO               ROME
LH  2407
AZ  0790 ROME                OSAKA
```

The database table SPFLI is read using two cursors, each with different conditions. The selected lines are read alternately in a DO loop.

```
DATA C TYPE CURSOR.

DATA WA TYPE SBOOK.

OPEN CURSOR C FOR SELECT CARRID CONNID FLDATE BOOKID SMOKER
                  FROM SBOOK
                  ORDER BY CARRID CONNID FLDATE SMOKER BOOKID.

FETCH NEXT CURSOR C INTO CORRESPONDING FIELDS OF WA.

WHILE SY-SUBRC = 0.
  IF WA-SMOKER = ' '.
    PERFORM NONSMOKER USING C.
  ELSEIF WA-SMOKER = 'X'.
    PERFORM SMOKER USING C.
    SKIP.
  ELSE.
    EXIT.
  ENDIF.
ENDWHILE.

FORM NONSMOKER USING N_CUR TYPE CURSOR.
  WHILE WA-SMOKER = ' ' AND SY-SUBRC = 0.
    FORMAT COLOR = 5.
    WRITE: / WA-CARRID, WA-CONNID, WA-FLDATE, WA-BOOKID.
    FETCH NEXT CURSOR N_CUR INTO CORRESPONDING FIELDS OF WA.
  ENDWHILE.
ENDFORM.

FORM SMOKER USING S_CUR TYPE CURSOR.
  WHILE WA-SMOKER = 'X' AND SY-SUBRC = 0.
    FORMAT COLOR = 6.
    WRITE: / WA-CARRID, WA-CONNID, WA-FLDATE, WA-BOOKID.
    FETCH NEXT CURSOR S_CUR INTO CORRESPONDING FIELDS OF WA.
  ENDWHILE.
ENDFORM.
```

The following is an extract from the list display:

```
AA   0017 1998/11/29 00000078
AA   0017 1998/11/29 00000079
AA   0017 1998/11/29 00000080
AA   0017 1998/11/29 00000081
AA   0017 1998/11/29 00000070
AA   0017 1998/11/29 00000071

AA   0017 1998/12/19 00000011
AA   0017 1998/12/19 00000012
AA   0017 1998/12/19 00000013
AA   0017 1998/12/19 00000015
AA   0017 1998/12/19 00000017
AA   0017 1998/12/19 00000018
AA   0017 1998/12/19 00000019
AA   0017 1998/12/19 00000020
AA   0017 1998/12/19 00000021
AA   0017 1998/12/19 00000023
AA   0017 1998/12/19 00000024
AA   0017 1998/12/19 00000025
AA   0017 1998/12/19 00000027
AA   0017 1998/12/19 00000014
AA   0017 1998/12/19 00000016
AA   0017 1998/12/19 00000022
AA   0017 1998/12/19 00000026

AA   0017 1998/12/21 00000001
AA   0017 1998/12/21 00000002
AA   0017 1998/12/21 00000003
AA   0017 1998/12/21 00000004
AA   0017 1998/12/21 00000005
AA   0017 1998/12/21 00000006
AA   0017 1998/12/21 00000008
AA   0017 1998/12/21 00000009
AA   0017 1998/12/21 00000010
AA   0017 1998/12/21 00000007
```

The program opens a cursor for the database table SBOOK. After the first FETCH statement, a subroutine is called, which is dependent on the contents of the SMOKER column. The cursor is passed to an interface parameter in the subroutine. The subroutines read further lines until the contents of the SMOKER column change. The subroutines perform different tasks using the lines read by the cursor.

```
DATA: WA_SPFLI   TYPE SPFLI,
      WA_SFLIGHT TYPE SFLIGHT.

DATA: C1 TYPE CURSOR,
      C2 TYPE CURSOR.

OPEN CURSOR C1 FOR   SELECT *
               FROM  SPFLI
               ORDER BY PRIMARY KEY.
```

Using a Cursor to Read Data

```
OPEN CURSOR C2 FOR   SELECT *
               FROM  SFLIGHT
               ORDER BY PRIMARY KEY.

DO.
  FETCH NEXT CURSOR C1 INTO WA_SPFLI.
  IF SY-SUBRC NE 0.
    EXIT.
  ENDIF.
  WRITE: / WA_SPFLI-CARRID, WA_SPFLI-CONNID.
  DO.
    FETCH NEXT CURSOR C2 INTO WA_SFLIGHT.
    IF  SY-SUBRC <> 0 OR WA_SFLIGHT-CARRID <> WA_SPFLI-CARRID
                      OR WA_SFLIGHT-CONNID <> WA_SPFLI-CONNID.
      EXIT.
    ELSE.
    WRITE: / WA_SFLIGHT-CARRID, WA_SFLIGHT-CONNID,
             WA_SFLIGHT-FLDATE.
    ENDIF.
  ENDDO.
ENDDO.
```

The output is as follows:

```
AA  0017
AA  0017 19.11.1998
AA  0017 22.11.1998
AA  0017 29.11.1998
AA  0017 19.12.1998
AA  0017 21.12.1998
AZ  0555
AZ  0555 22.11.1998
AZ  0555 29.11.1998
AZ  0555 19.12.1998
AZ  0555 21.12.1998
AZ  0789
AZ  0789 02.12.1998
AZ  0789 09.12.1998
AZ  0789 29.12.1998
AZ  0789 31.12.1998
LH  0400
LH  0400 02.12.1998
LH  0400 09.12.1998
LH  0400 29.12.1998
LH  0400 31.12.1998
```

The program opens a cursor for each of the table SPFLI and SFLIGHT. Since both tables are linked by a foreign key relationship, it is possible to program a nested loop by sorting the selection by its primary key, so that the data read in the inner loop depends on the data in the outer loop. This programming method is quicker than using nested SELECT statements, since the cursor for the inner loop does not continually have to be reopened.

Locking Conflicts

In R/3 on DB2 for OS/390, regular commits at a recommended frequency of at least one per minute also need to be included in read-only transactions and reports to release concurrently acquired locks on the database. This is because locks occur when reading buffer tables, cluster and pool tables. These tables are read using the Read Stability isolation level, and this sets shared locks to prevent simultaneous changes to tables.
Apart from timeouts and deadlocks, locking conflicts can cause failures of DDL (Data Definition Language) statements, such as CREATE, ALTER, and DROP, and slow down online reorganizations.

If a particular cursor needs to be open across commits, you can use the WITH HOLD option to preserve the cursor position.

You should execute the commits using Native SQL as follows:

```
IF SY-DBSYS = 'DB2'.
  EXEC SQL.
    COMMIT WORK
  ENDEXEC.
ENDIF.
```

Changing Data

Open SQL contains a set of statements that allow you to change data in the database. You can insert, change and delete entries in database tables. However, you must remember that Open SQL statements do not check authorization [Page 502] or the consistency of data in the database. The following statements are purely technical means of programming database updates. They are to be used with care, and, outside the SAP transaction concept [Page 1260], only to be used in exceptional cases. The SAP transaction concept addresses the question of database update programming in the R/3 System. It discusses the difference between a database LUW and an SAP LUW, and explains about SAP transactions and the R/3 locking concept.

Inserting Lines into Tables

The Open SQL statement for inserting data into a database table is:

INSERT INTO <target> <lines>.

It allows you to insert one or more lines into the database table <target>. You can only insert lines into an ABAP Dictionary view if it only contains fields from one table, and its maintenance status is defined as *Read and change*. You may specify the database table <target> either statically or dynamically.

Specifying a Database Table

To specify the database table statically, enter the following for <target>:

INSERT INTO <dbtab> [CLIENT SPECIFIED] <lines>.

where <dbtab> is the name of a database table defined in the ABAP Dictionary.

To specify the database table dynamically, enter the following for <target>:

INSERT INTO (<name>) [CLIENT SPECIFIED] <lines>.

where the field <name> contains the name of a database table defined in the ABAP Dictionary.

You can use the CLIENT SPECIFIED addition to disable automatic client handling.

Inserting a Single Line

To insert a single line into a database table, use the following:

INSERT INTO <target> VALUES <wa>.

The contents of the work area <wa> are written to the database table <dbtab>. The work area <wa> must be a data object with at least the same length and alignment [Page 195] as the line structure of the database table. The data is placed in the database table according to the line structure of the table, and regardless of the structure of the work area. It is a good idea to define the work area with reference to the structure of the database table.

If the database table does not already contain a line with the same primary key as specified in the work area, the operation is completed successfully and SY-SUBRC is set to 0. Otherwise, the line is not inserted, and SY-SUBRC is set to 4.

You can also insert single lines using the following shortened form of the INSERT statement:

INSERT <target> FROM <wa>.

Using FROM instead of VALUE allows you to omit the INTO clause. Shorter still is:

INSERT <dbtab>.

In this case, the contents of the table work area <dbtab> are inserted into the database table with the same name. You **must** declare this table work area using the TABLES [Page 130] statement. In this case, it is not possible to specify the name of the database table dynamically. Table work areas with the same name as the database table (necessary before Release 4.0) should no longer be used for the sake of clarity.

Inserting Several Lines

To insert several lines into a database table, use the following:

Inserting Lines into Tables

INSERT <target> FROM TABLE <itab> [ACCEPTING DUPLICATE KEYS].

This writes **all** lines of the internal table <itab> to the database table in one single operation. The same rules apply to the line type of <itab> as to the work area <wa> described above.

If the system is able to insert all of the lines from the internal table, SY-SUBRC is set to 0. If one or more lines cannot be inserted because the database already contains a line with the same primary key, a runtime error occurs. You can prevent the runtime error occurring by using the addition ACCEPTING DUPLICATE KEYS. In this case, the lines that would otherwise cause runtime errors are discarded, and SY-SUBRC is set to 4.

The system field SY-DBCNT contains the number of lines inserted into the database table, regardless of the value in SY-SUBRC.

Whenever you want to insert more than one line into a database table, it is more efficient to work with an internal table than to insert the lines one by one.

Examples

Adding single lines

```
TABLES SPFLI.

DATA WA TYPE SPFLI.

WA-CARRID = 'LH'.
WA-CITYFROM = 'WASHINGTON'.
...
INSERT INTO SPFLI VALUES WA.

WA-CARRID = 'UA'.
WA-CITYFROM = 'LONDON'.
...
INSERT SPFLI FROM WA.

SPFLI-CARRID = 'LH'.
SPFLI-CITYFROM = 'BERLIN'.
...
INSERT SPFLI.
```

This program inserts a single line into the database table SPFLI using each of the three possible variants of the INSERT statement.

Instead of

INSERT SPFLI

in the last line, you could also use the longer forms

INSERT SPFLI FROM SPFLI

or

INSERT INTO SPFLI VALUES SPFLI

The name SPFLI is therefore not unique.

```
DATA: ITAB TYPE HASHED TABLE OF SPFLI
                  WITH UNIQUE KEY CARRID CONNID,
      WA LIKE LINE OF ITAB.

WA-CARRID = 'UA'. WA-CONNID = '0011'. WA-CITYFROM = ...
INSERT WA INTO TABLE ITAB.

WA-CARRID = 'LH'. WA-CONNID = '1245'. WA-CITYFROM = ...
INSERT WA INTO TABLE ITAB.

WA-CARRID = 'AA'. WA-CONNID = '4574'. WA-CITYFROM = ...
INSERT WA INTO TABLE ITAB.

...

INSERT SPFLI FROM TABLE ITAB ACCEPTING DUPLICATE KEYS.

IF SY-SUBRC = 0.
  ...
ELSEIF SY-SUBRC = 4.
  ...
ENDIF.
```

This example fills a hashed table ITAB and inserts its contents into the database table SPFLI. The program examines the contents of SY-SUBRC to see if the operation was successful.

Changing Lines

The Open SQL statement for changing data in a database table is:

UPDATE <target> <lines>.

It allows you to change one or more lines in the database table <target>. You can only change lines in an ABAP Dictionary view if it only contains fields from one table, and its maintenance status is defined as *Read and change*. You may specify the database table <target> either statically or dynamically.

Specifying a Database Table

To specify the database table statically, enter the following for <target>:

UPDATE <dbtab> [CLIENT SPECIFIED] <lines>.

where <dbtab> is the name of a database table defined in the ABAP Dictionary.

To specify the database table dynamically, enter the following for <target>:

UPDATE (<name>) [CLIENT SPECIFIED] <lines>.

where the field <name> contains the name of a database table defined in the ABAP Dictionary.

You can use the CLIENT SPECIFIED addition to disable automatic client handling.

Changing Lines Column by Column

To change certain columns in the database table, use the following:

```
UPDATE <target> SET <set1> <set2> ... [WHERE <cond>].
```

The WHERE clause [Page 1064] determines the lines that are changed. If you do not specify a WHERE clause, **all** lines are changed. The expressions $<set_1>$ are three different SET statements that determine the columns to be changed, and how they are to be changed:

- $<s_i>$ = <f>

  The value in column $<s_i>$ is set to the value <f> for all lines selected.

- $<s_i>$ = $<s_i>$ + <f>

  The value in column $<s_i>$ is increased by the value of <f> for all lines selected.

- $<s_i>$ = $<s_i>$ - <f>

  The value in column $<s_i>$ is decreased by the value of <f> for all lines selected.

<f> can be a data object or a column of the database table. You address the columns using their direct names.

If at least one line is changed, the system sets SY-SUBRC to 0, otherwise to 4. SY-DBCNT contains the number of lines changed.

If you use SET statements, you cannot specify the database table dynamically.

Overwriting Individual Lines From a Work Area

To overwrite a single line in a database table with the contents of a work area, use the following:

UPDATE <target> FROM <wa>.

The contents of the work area <wa> overwrite the line in the database table <dbtab> that has the same primary key. The work area <wa> must be a data object with at least the same length and alignment [Page 195] as the line structure of the database table. The data is placed in the database table according to the line structure of the table, and regardless of the structure of the work area. It is a good idea to define the work area with reference to the structure of the database table.

If the database table contains a line with the same primary key as specified in the work area, the operation is completed successfully and SY-SUBRC is set to 0. Otherwise, the line is not inserted, and SY-SUBRC is set to 4.

A shortened form of the above statement is:

UPDATE <dbtab>.

In this case, the contents of the table work area <dbtab> are used to update the database table with the same name. You **must** declare this table work area using the TABLES [Page 130] statement. In this case, it is not possible to specify the name of the database table dynamically. Table work areas with the same name as the database table (necessary before Release 4.0) should no longer be used for the sake of clarity.

Overwriting Several Lines Using an Internal Table

To overwrite several lines in a database table with the contents of an internal table, use the following:

UPDATE <target> FROM TABLE <itab>.

The contents of the internal table <itab> overwrite the lines in the database table <dbtab> that have the same primary keys. The same rules apply to the line type of <itab> as to the work area <wa> described above.

If the system cannot change a line because no line with the specified key exists, it does not terminate the entire operation, but continues processing the next line of the internal table.

If all lines from the internal table have been processed, SY-SUBRC is set to 0. Otherwise, it is set to 4. If not all lines are used, you can calculate the number of unused lines by subtracting the number of processed lines in SY-DBCNT from the total number of lines in the internal table. If the internal table is empty, SY-SUBRC and SY-DBCNT are set to 0.

Whenever you want to overwrite more than one line in a database table, it is more efficient to work with an internal table than to change the lines one by one.

Examples

```
UPDATE SFLIGHT SET PLANETYPE = 'A310'
               PRICE = PRICE - '100.00'
               WHERE CARRID = 'LH' AND CONNID = '0402'.
```

This example overwrites the contents of the PLANETYPE column with A310 and decreases the value of the PRICE column by 100 for each entry in SFLIGHT where CARRID contains ‘LH’ and CONNID contains ‘402’.

```
TABLES SPFLI.
```

```
DATA WA TYPE SPFLI.

MOVE 'AA'         TO WA-CARRID.
MOVE '0064'       TO WA-CONNID.
MOVE 'WASHINGTON' TO WA-CITYFROM.
...
UPDATE SPFLI FROM WA.

MOVE 'LH'     TO SPFLI-CARRID.
MOVE '0017'   TO SPFLI-CONNID.
MOVE 'BERLIN' TO SPFLI-CITYFROM.
...
UPDATE SPFLI.
```

CARRID and CONNID are the primary key fields of table SPFLI. All fields of those lines where the primary key fields are "AA" and "0064", or "LH" and "0017", are replaced by the values in the corresponding fields of the work area WA or the table work area SPFLI.

```
DATA: ITAB TYPE HASHED TABLE OF SPFLI
           WITH UNIQUE KEY CARRID CONNID,
      WA LIKE LINE OF ITAB.

WA-CARRID = 'UA'. WA-CONNID = '0011'. WA-CITYFROM = ...
INSERT WA INTO TABLE ITAB.

WA-CARRID = 'LH'. WA-CONNID = '1245'. WA-CITYFROM = ...
INSERT WA INTO TABLE ITAB.

WA-CARRID = 'AA'. WA-CONNID = '4574'. WA-CITYFROM = ...
INSERT WA INTO TABLE ITAB.

...

UPDATE SPFLI FROM TABLE ITAB.
```

This example fills a hashed table ITAB and then overwrites the lines in SPFLI that have the same primary key (CARRID and CONNID) as a line in the internal table.

Deleting Lines

The Open SQL statement for deleting lines from a database table is:

DELETE [FROM] <target> <lines>.

It allows you to delete one or more lines from the database table <target>. You can only delete lines from an ABAP Dictionary view if it only contains fields from one table, and its maintenance status is defined as *Read and change*. You may specify the database table <target> either statically or dynamically.

Specifying a Database Table

To specify the database table statically, enter the following for <target>:

DELETE [FROM] <dbtab> [CLIENT SPECIFIED] <lines>.

where <dbtab> is the name of a database table defined in the ABAP Dictionary.

To specify the database table dynamically, enter the following for <target>:

DELETE [FROM] (<name>) [CLIENT SPECIFIED] <lines>.

where the field <name> contains the name of a database table defined in the ABAP Dictionary.

You can use the CLIENT SPECIFIED addition to disable automatic client handling.

Selecting Lines Using Conditions

To select the lines that you want to delete using a condition, use the following:

DELETE FROM <target> WHERE <cond>.

All of the lines in the database table that satisfy the conditions in the WHERE clause [Page 1064] are deleted. The FROM expression must occur between the keyword and the database table.

You should take particular care when programming the WHERE clause to ensure that you do not delete the wrong lines. For example, if you specify an empty internal table in a dynamic WHERE clause, all of the lines in the table are deleted.

If at least one line is deleted, the system sets SY-SUBRC to 0, otherwise to 4. SY-DBCNT contains the number of lines deleted.

Selecting Single Lines Using Work Areas

Instead of using a WHERE clause, you can select lines for deletion using the contents of a work area. In this case, you would write:

DELETE <target> FROM <wa>.

This deletes the line with the same primary key as the work area <wa>. The FROM expression must not occur between the keyword and the database table. The work area <wa> must be a data object with at least the same length and alignment [Page 195] as the line structure of the database table. The key is read according to the structure of the table line, and not that of the work area. It is a good idea to define the work area with reference to the structure of the database table.

Deleting Lines

If the database table contains a line with the same primary key as specified in the work area, the operation is completed successfully and SY-SUBRC is set to 0. Otherwise, the line is not deleted, and SY-SUBRC is set to 4.

A shortened form of the above statement is:

DELETE <dbtab>.

In this case, the contents of the table work area <dbtab> are used to delete from the database table with the same name. You **must** declare this table work area using the TABLES [Page 130] statement. In this case, it is not possible to specify the name of the database table dynamically. Table work areas with the same name as the database table (necessary before Release 4.0) should no longer be used for the sake of clarity.

Selecting Several Lines Using an Internal Table

You can also use an internal table to delete several lines:

DELETE <target> FROM TABLE itab <wa>.

This deletes all lines from the database that have the same primary key as a line in the internal table <itab>. The same rules apply to the line type of <itab> as to the work area <wa> described above.

If the system cannot delete a line because no line with the specified key exists, it does not terminate the entire operation, but continues processing the next line of the internal table.

If all lines from the internal table have been processed, SY-SUBRC is set to 0. Otherwise, it is set to 4. If not all lines are used, you can calculate the number of unused lines by subtracting the number of deleted lines in SY-DBCNT from the total number of lines in the internal table. If the internal table is empty, SY-SUBRC and SY-DBCNT are set to 0.

Whenever you want to delete more than one line from a database table, it is more efficient to work with an internal table than to insert the lines one by one.

Examples

```
DELETE FROM SFLIGHT WHERE PLANETYPE = 'A310' AND
                          CARRID = 'LH'.
```

This deletes all lines from SFLIGHT where PLANETYPE has the value A310 and CARRID contains the value LH.

```
TABLES SPFLI.

DATA: BEGIN OF WA,
        CARRID TYPE SPFLI-CARRID,
        CONNID TYPE SPFLI-CONNID,
      END OF WA.

MOVE 'AA'          TO WA-CARRID.
MOVE '0064'        TO WA-CONNID.
DELETE SPFLI FROM WA.
```

```
MOVE 'LH'     TO SPFLI-CARRID.
MOVE '0017'   TO SPFLI-CONNID.

DELETE SPFLI.
```

CARRID and CONNID are the primary key fields of table SPFLI. The lines with the primary keys AA 0064 and LH 0017 are deleted.

```
DATA: BEGIN OF WA,
        CARRID TYPE SPFLI-CARRID,
        CONNID TYPE SPFLI-CONNID,
      END OF WA,
      ITAB LIKE HASHED TABLE OF WA
           WITH UNIQUE KEY CARRID CONNID.

WA-CARRID = 'UA'. WA-CONNID = '0011'.
INSERT WA INTO TABLE ITAB.

WA-CARRID = 'LH'. WA-CONNID = '1245'.
INSERT WA INTO TABLE ITAB.

WA-CARRID = 'AA'. WA-CONNID = '4574'.
INSERT WA INTO TABLE ITAB.

...

DELETE SPFLI FROM TABLE ITAB.
```

This example defines a hashed table ITAB with the structure of the primary key of the database table SPFLI. After ITAB has been filled, those lines of the database table are deleted that have the same contents in the primary key fields (CARRID and CONNID) as a line in the internal table.

Inserting or Changing Lines

To insert lines into a database table regardless of whether there is already a line in the table with the same primary key, use the following:

MODIFY <target> <lines>.

If the database table contains no line with the same primary key as the line to be inserted, MODIFY works like INSERT, that is, the line is added.

If the database already contains a line with the same primary key as the line to be inserted, MODIFY works like UPDATE, that is, the line is changed.

For performance reasons, you should use MODIFY only if you cannot distinguish between these two options in your ABAP program.

You can add or change one or more lines <lines> in a database table <target>. You can only insert or change lines in an ABAP Dictionary view if it only contains fields from one table, and its maintenance status is defined as *Read and change*. You may specify the database table <target> either statically or dynamically.

Specifying a Database Table

To specify the database table statically, enter the following for <target>:

MODIFY <dbtab> [CLIENT SPECIFIED] <lines>.

where <dbtab> is the name of a database table defined in the ABAP Dictionary.

To specify the database table dynamically, enter the following for <target>:

MODIFY (<name>) [CLIENT SPECIFIED] <lines>.

where the field <name> contains the name of a database table defined in the ABAP Dictionary.

You can use the CLIENT SPECIFIED addition to disable automatic client handling.

Inserting or Changing Single Lines

To insert or change a single line in a database table, use the following:

MODIFY <target> FROM <wa>.

The contents of the work area <wa> are written to the database table <dbtab>. The work area <wa> must be a data object with at least the same length and alignment [Page 195] as the line structure of the database table. The data is placed in the database table according to the line structure of the table, and regardless of the structure of the work area. It is a good idea to define the work area with reference to the structure of the database table.

If the database table does not already contain a line with the same primary key as specified in the work area, a new line is inserted. If the database table does already contain a line with the same primary key as specified in the work area, the existing line is overwritten. SY-SUBRC is always set to 0.

A shortened form of the above statement is:

MODIFY <dbtab>.

In this case, the contents of the table work area <dbtab> are inserted into the database table with the same name. You **must** declare this table work area using the TABLES [Page 130] statement. In this case, it is not possible to specify the name of the database table dynamically.

Table work areas with the same name as the database table (necessary before Release 4.0) should no longer be used for the sake of clarity.

Inserting or Changing Several Lines

To insert or change several lines in a database table, use the following:

MODIFY <target> FROM TABLE <itab>.

Those lines of the internal table <itab> for which there is not already a line in the database table with the same primary key are inserted into the table. Those lines of the internal table <itab> for which there is already a line in the database table with the same primary key overwrite the existing line in the database table. The same rules apply to the line type of <itab> as to the work area <wa> described above.

SY-SUBRC is always set to 0. SY-DBCNT is set to the number of lines in the internal table.

Committing Database Changes

Open SQL contains the statements

COMMIT WORK.

and

ROLLBACK WORK.

for confirming or undoing database updates. COMMIT WORK always concludes a database LUW [Page 1262] and starts a new one. ROLLBACK WORK always undoes all changes back to the start of the database LUW.

These statements are part of the SAP transaction concept [Page 1260], and should only be used in this context.

As well as the COMMIT WORK and ROLLBACK WORK statements, there are other situations where data is implicitly written to the database or rolled back, and where database LUWs therefore begin and end.

Furthermore, the above statements also control SAP LUWs [Page 1265], which extend over several database LUWs.

The ABAP statements COMMIT WORK and ROLLBACK WORK only work like the corresponding Standard SQL statements if the entire application program is represented by a single implicit database LUW, that is, for example, in a single dialog step.

Performance Notes

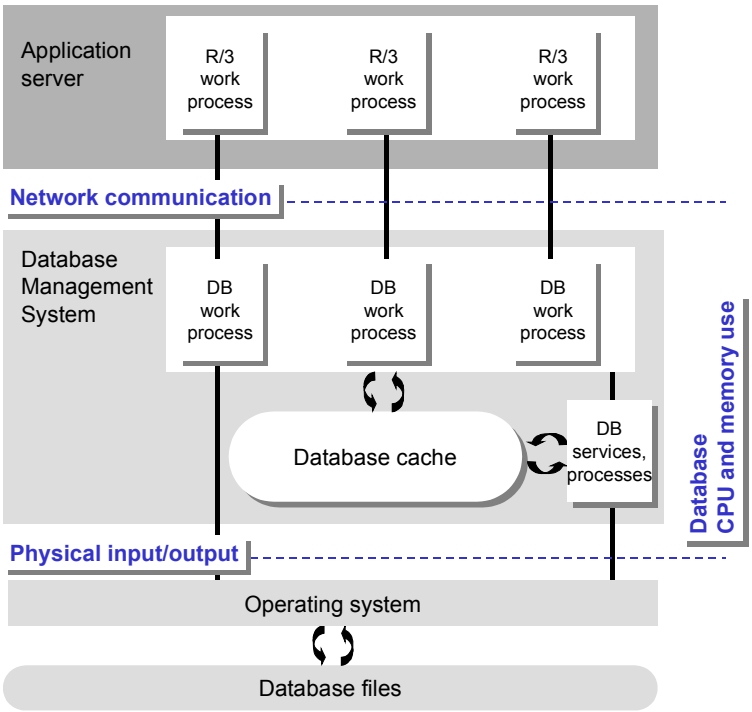
The performance of an ABAP program is largely determined by the efficiency of its database accesses. It is therefore worth analyzing your SQL statements closely. To help you to do this, you should use the *Performance Trace* and *Runtime Analysis* tools (*Test* menu in the ABAP Workbench). In particular, the *SQL Trace* in the *Performance Trace* shows you which parts of Open SQL statements are processed where, and how long they take.

To understand how SQL statements affect the runtime of ABAP programs, you need to understand the underlying system architecture. The work processes [Page 32] of an application server [Page 27] are logged onto the database system (server) as users (clients) for as long as the R/3 System is running. The database management system (DBMS) forms the connection between users and data.

DBMS architecture

The general structure of a DBMS is as follows:

- A database work process provides a service that database clients can call.
- There are different database services for different tasks, for example, for establishing connections, changing database tables, locking database entries, archiving, and so on.
- There is a large shared memory area, containing the DBMS cache and other resources such as the statement cache, and redo information.
- The database files are stored on a hard disk, and are managed by the file system.



Within this architecture, there are four points that are important in respect of performance:

- Physical input/output (I/O)

  Reading and writing database files is the greatest bottleneck. The mark of a well-configured system is the speed of its I/O.

- The memory used by the database cache.
- The CPU usage on the host on which the database is installed.

  On a symmetrical multi-processor system, this is irrelevant.

- Network communication

  Although unimportant for small data volumes, it becomes a bottleneck when large quantities of data are involved.

The Optimizer

Each database system uses an optimizer whose task is to create the execution plan for SQL statements (for example, to determine whether to use an index or table scan). There are two kinds of optimizers:

Rule based

Rule based optimizers analyze the structure of an SQL statement (mainly the SELECT and WHERE clauses without their values) and the table index or indexes. They then use an algorithm to work out which method to use to execute the statement.

Cost based

Cost based optimizers use the above procedure, but also analyze some of the values in the WHERE clause and the table statistics. The statistics contain low and high values of the fields, or a histogram containing the distribution of data in the table. Since the cost based optimizer uses more information about the table, it usually leads to faster database access. Its disadvantage is that the statistics have to be periodically updated.

Use

ORACLE databases up to and including release 7.1 use a rule-based optimizer. From Release 7.2 (R/3 Release 4.0A), they use a cost-based optimizer. All other database systems use a cost-based optimizer.

Rules for Efficient Open SQL Programming

Based on the above architecture, there are five rules that you should follow to make database accesses in ABAP programs more efficient. They apply in particular to the following database systems:

- ORACLE
- INFORMIX
- ADABAS
- DB2/400 (uses EBCDIC codepage).
- Microsoft SQL Server

For the following database systems, they apply either in part or not at all:

- DB2/6000
- DB2/MVS
- ORACLE Parallel Server (OPS)

The five rules are explained in the following sections:

Keep the Result Set Small [Page 1106]
Minimize the Amount of Data Transferred [Page 1107]
Minimize the Number of Data Transfers [Page 1108]
Minimize the Search Overhead [Page 1110]
Reduce the Database Load [Page 1112]

Keep the Result Set Small

You should aim to keep the result set small. This reduces both the amount of memory used in the database system and the network load when transferring data to the application server. To reduce the size of your result sets, use the WHERE and HAVING clauses.

Using the WHERE Clause

Whenever you access a database table, you should use a WHERE clause in the corresponding Open SQL statement. Even if a program containing a SELECT statement with no WHERE clause performs well in tests, it may slow down rapidly in your production system, where the data volume increases daily. You should only dispense with the WHERE clause in exceptional cases where you really need the entire contents of the database table every time the statement is executed.

When you use the WHERE clause, the database system optimizes the access and only transfers the required data. You should never transfer unwanted data to the application server and then filter it using ABAP statements.

Using the HAVING Clause

After selecting the required lines in the WHERE clause, the system then processes the GROUP BY clause, if one exists, and summarizes the database lines selected. The HAVING clause allows you to restrict the grouped lines, and in particular, the aggregate expressions, by applying further conditions.

Effect

If you use the WHERE and HAVING clauses correctly:

- There are no more physical I/Os in the database than necessary
- No unwanted data is stored in the database cache (it could otherwise displace data that is actually required)
- The CPU usage of the database host is minimize
- The network load is reduced, since only the data that is required by the application is transferred to the application server.

Minimize the Amount of Data Transferred

Data is transferred between the database system and the application server in blocks. Each block is up to 32 KB in size (the precise size depends on your network communication hardware). Administration information is transported in the blocks as well as the data.

To minimize the network load, you should transfer as few blocks as possible. Open SQL allows you to do this as follows:

Restrict the Number of Lines

If you only want to read a certain number of lines in a SELECT statement, use the UP TO <n> ROWS addition in the FROM clause. This tells the database system only to transfer <n> lines back to the application server. This is more efficient than transferring more lines than necessary back to the application server and then discarding them in your ABAP program.

If you expect your WHERE clause to return a large number of duplicate entries, you can use the DISTINCT addition in the SELECT clause.

Restrict the Number of Columns

You should only read the columns from a database table that you actually need in the program. To do this, list the columns in the SELECT clause. Note here that the INTO CORRESPONDING FIELDS addition in the INTO clause is only efficient with large volumes of data, otherwise the runtime required to compare the names is too great. For small amounts of data, use a list of variables in the INTO clause.

Do not use * to select all columns unless you really need them. However, if you list individual columns, you may have to adjust the program if the structure of the database table is changed in the ABAP Dictionary. If you specify the database table dynamically, you must always read all of its columns.

Use Aggregate Functions

If you only want to use data for calculations, it is often more efficient to use the aggregate functions of the SELECT clause than to read the individual entries from the database and perform the calculations in the ABAP program.

Aggregate functions allow you to find out the number of values and find the sum, average, minimum, and maximum values. Following an aggregate expression, only its result is transferred from the database.

Data Transfer when Changing Table Lines

When you use the UPDATE statement to change lines in the table, you should use the WHERE clause to specify the relevant lines, and then SET statements to change only the required columns.

When you use a work area to overwrite table lines, too much data is often transferred. Furthermore, this method requires an extra SELECT statement to fill the work area.

Minimize the Number of Data Transfers

In every Open SQL statement, data is transferred between the application server and the database system. Furthermore, the database system has to construct or reopen the appropriate administration data for each database access. You can therefore minimize the load on the network and the database system by minimizing the number of times you access the database.

Multiple Operations Instead of Single Operations

When you change data using INSERT, UPDATE, and DELETE, use internal tables instead of single entries. If you read data using SELECT, it is worth using multiple operations if you want to process the data more than once, other wise, a simple select loop is more efficient.

Avoid Repeated Access

As a rule you should read a given set of data once only in your program, and using a single access. Avoid accessing the same data more than once (for example, SELECT before an UPDATE).

Avoid Nested SELECT Loops

A simple SELECT loop is a single database access whose result is passed to the ABAP program line by line. Nested SELECT loops mean that the number of accesses in the inner loop is multiplied by the number of accesses in the outer loop. You should therefore only use nested SELECT loops if the selection in the outer loop contains very few lines.

However, using combinations of data from different database tables is more the rule than the exception in the relational data model. You can use the following techniques to avoid nested SELECT statements:

ABAP Dictionary Views

You can define joins between database tables statically and systemwide as views in the ABAP Dictionary. ABAP Dictionary views can be used by all ABAP programs. One of their advantages is that fields that are common to both tables (join fields) are only transferred once from the database to the application server.

Views in the ABAP Dictionary are implemented as inner joins. If the inner table contains no lines that correspond to lines in the outer table, no data is transferred. This is not always the desired result. For example, when you read data from a text table, you want to include lines in the selection even if the corresponding text does not exist in the required language. If you want to include all of the data from the outer table, you can program a left outer join in ABAP.

The links between the tables in the view are created and optimized by the database system. Like database tables, you can buffer views on the application server. The same buffering rules apply to views as to tables. In other words, it is most appropriate for views that you use mostly to read data. This reduces the network load and the amount of physical I/O in the database.

Joins in the FROM Clause

You can read data from more than one database table in a single SELECT statement by using inner or left outer joins in the FROM clause.

The disadvantage of using joins is that redundant data is read from the hierarchically-superior table if there is a 1:N relationship between the outer and inner tables. This can considerably

increase the amount of data transferred from the database to the application server. Therefore, when you program a join, you should ensure that the SELECT clause contains a list of only the columns that you really need. Furthermore, joins bypass the table buffer and read directly from the database. For this reason, you should use an ABAP Dictionary view instead of a join if you only want to read the data.

The runtime of a join statement is heavily dependent on the database optimizer, especially when it contains more than two database tables. However, joins are nearly always quicker than using nested SELECT statements.

Subqueries in the WHERE and HAVING Clauses

Another way of accessing more than one database table in the same Open SQL statement is to use subqueries in the WHERE or HAVING clause. The data from a subquery is not transferred to the application server. Instead, it is used to evaluate conditions in the database system. This is a simple and effective way of programming complex database operations.

Using Internal Tables

It is also possible to avoid nested SELECT loops by placing the selection from the outer loop in an internal table and then running the inner selection once only using the FOR ALL ENTRIES addition. This technique stems from the time before joins were allowed in the FROM clause. On the other hand, it does prevent redundant data from being transferred from the database.

Using a Cursor to Read Data

A further method is to decouple the INTO clause from the SELECT statement by opening a cursor using OPEN CURSOR and reading data line by line using FETCH NEXT CURSOR. You must open a new cursor for each nested loop. In this case, you must ensure yourself that the correct lines are read from the database tables in the correct order. This usually requires a foreign key relationship between the database tables, and that they are sorted by the foreign key.

Minimize the Search Overhead

You minimize the size of the result set by using the WHERE and HAVING clauses. To increase the efficiency of these clauses, you should formulate them to fit with the database table indexes.

Database Indexes

Indexes speed up data selection from the database. They consist of selected fields of a table, of which a copy is then made in **sorted** order. If you specify the index fields correctly in a condition in the WHERE or HAVING clause, the system only searches part of the index (index range scan).

The primary index is always created automatically in the R/3 System. It consists of the primary key fields of the database table. This means that for each combination of fields in the index, there is a maximum of one line in the table. This kind of index is also known as UNIQUE.

If you cannot use the primary index to determine the result set because, for example, none of the primary index fields occur in the WHERE or HAVING clause, the system searches through the entire table (full table scan). For this case, you can create secondary indexes, which can restrict the number of table entries searched to form the result set.

You specify the fields of secondary indexes using the ABAP Dictionary [Ext.]. You can also determine whether the index is unique or not. However, you should not create secondary indexes to cover all possible combinations of fields.

Only create one if you select data by fields that are not contained in another index, and the performance is very poor. Furthermore, you should only create secondary indexes for database tables from which you mainly read, since indexes have to be updated each time the database table is changed. As a rule, secondary indexes should not contain more than four fields, and you should not have more than five indexes for a single database table.

If a table has more than five indexes, you run the risk of the optimizer choosing the wrong one for a particular operation. For this reason, you should avoid indexes with overlapping contents.

Secondary indexes should contain columns that you use frequently in a selection, and that are as highly selective as possible. The fewer table entries that can be selected by a certain column, the higher that column's selectivity. Place the most selective fields at the beginning of the index. Your secondary index should be so selective that each index entry corresponds to at most five percent of the table entries. If this is not the case, it is not worth creating the index. You should also avoid creating indexes for fields that are not always filled, where their value is initial for most entries in the table.

If all of the columns in the SELECT clause are contained in the index, the system does not have to search the actual table data after reading from the index. If you have a SELECT clause with very few columns, you can improve performance dramatically by including these columns in a secondary index.

Formulating Conditions for Indexes

You should bear in mind the following when formulating conditions for the WHERE and HAVING clauses so that the system can use a database index and does not have to use a full table scan.

Check for Equality and Link Using AND

The database index search is particularly efficient if you check all index fields for equality (= or EQ) and link the expressions using AND.

Use Positive Conditions

The database system only supports queries that describe the result in positive terms, for example, EQ or LIKE. It does not support negative expressions like NE or NOT LIKE.

If possible, avoid using the NOT operator in the WHERE clause, because it is not supported by database indexes; invert the logical expression instead.

Using OR

The optimizer usually stops working when an OR expression occurs in the condition. This means that the columns checked using OR are not included in the index search. An exception to this are OR expressions at the outside of conditions. You should try to reformulate conditions that apply OR expressions to columns relevant to the index, for example, into an IN condition.

Using Part of the Index

If you construct an index from several columns, the system can still use it even if you only specify a few of the columns in a condition. However, in this case, the sequence of the columns in the index is important. A column can only be used in the index search if all of the columns before it in the index definition have also been specified in the condition.

Checking for Null Values

The IS NULL condition can cause problems with indexes. Some database systems do not store null values in the index structure. Consequently, this field cannot be used in the index.

Avoid Complex Conditions

Avoid complex conditions, since the statements have to be broken down into their individual components by the database system.

Reduce the Database Load

Unlike application servers and presentation servers, there is only one database server in your system. You should therefore aim to reduce the database load as much as possible. You can use the following methods:

Buffer Tables on the Application Server

You can considerably reduce the time required to access data by buffering it in the application server table buffer. Reading a single entry from table T001 can take between 8 and 600 milliseconds, while reading it from the table buffer takes 0.2 - 1 milliseconds.

Whether a table can be buffered or not depends its technical attributes in the ABAP Dictionary. There are three buffering types:

- **Resident buffering** (100%) The first time the table is accessed, its entire contents are loaded in the table buffer.
- **Generic buffering** In this case, you need to specify a generic key (some of the key fields) in the technical settings of the table in the ABAP Dictionary. The table contents are then divided into generic areas. When you access data with one of the generic keys, the whole generic area is loaded into the table buffer. Client-specific tables are often buffered generically by client.
- **Partial buffering** (single entry) Only single entries are read from the database and stored in the table buffer.

When you read from buffered tables, the following happens:

1. An ABAP program requests data from a buffered table.
2. The ABAP processor interprets the Open SQL statement. If the table is defined as a buffered table in the ABAP Dictionary, the ABAP processor checks in the local buffer on the application server to see if the table (or part of it) has already been buffered.
3. If the table has not yet been buffered, the request is passed on to the database. If the data exists in the buffer, it is sent to the program.
4. The database server passes the data to the application server, which places it in the table buffer.
5. The data is passed to the program.

When you change a buffered table, the following happens:

1. The database table is changed and the buffer on the application server is updated. The database interface logs the update statement in the table DDLOG. If the system has more than one application server, the buffer on the other servers is not updated at once.
2. All application servers periodically read the contents of table DDLOG, and delete the corresponding contents from their buffers where necessary. The granularity depends on the buffering type. The table buffers in a distributed system are generally synchronized every 60 seconds (parameter: rsdisp/bufreftime).
3. Within this period, users on non-synchronized application servers will read old data. The data is not recognized as obsolete until the next buffer synchronization. The next time it is accessed, it is re-read from the database.

You should buffer the following types of tables:

- Tables that are read very frequently
- Tables that are changed very infrequently
- Relatively small tables (few lines, few columns, or short columns)
- Tables where delayed update is acceptable.

Once you have buffered a table, take care not to use any Open SQL statements that bypass the buffer.

The SELECT statement bypasses the buffer when you use any of the following:

- The BYPASSING BUFFER addition in the FROM clause
- The DISTINCT addition in the SELECT clause
- Aggregate expressions in the SELECT clause
- Joins in the FROM clause
- The IS NULL condition in the WHERE clause
- Subqueries in the WHERE clause
- The ORDER BY clause
- The GROUP BY clause
- The FOR UPDATE addition

Furthermore, all Native SQL statements bypass the buffer.

Avoid Reading Data Repeatedly

If you avoid reading the same data repeatedly, you both reduce the number of database accesses and reduce the load on the database. Furthermore, a “dirty read” may occur with database tables other than Oracle. This means that the second time you read data from a database table, it may be different from the data read the first time. To ensure that the data in your program is consistent, you should read it once only and then store it in an internal table.

Sort Data in Your ABAP Programs

The ORDER BY clause in the SELECT statement is not necessarily optimized by the database system or executed with the correct index. This can result in increased runtime costs. You should only use ORDER BY if the database sort uses the same index with which the table is read. To find out which index the system uses, use SQL Trace in the ABAP Workbench Performance Trace. If the indexes are not the same, it is more efficient to read the data into an internal table or extract and sort it in the ABAP program using the SORT statement.

Use Logical Databases

SAP supplies logical databases [Page 1163] for all applications. A logical database is an ABAP program that decouples Open SQL statements from application programs. They are optimized for the best possible database performance. However, it is important that you use the right logical database. The hierarchy of the data you want to read must reflect the structure of the logical database, otherwise, they can have a negative effect on performance. For example, if you want to read data from a table right at the bottom of the hierarchy of the logical database, it has to read at least the key fields of all tables above it in the hierarchy. In this case, it is more efficient to use a SELECT statement.

Native SQL

Open SQL allows you to access database tables declared in the ABAP Dictionary regardless of the database platform that you R/3 System is using. Native SQL allows you to use database-specific SQL statements in an ABAP program. This means that you can use database tables that are not administered by the ABAP Dictionary, and therefore integrate data that is not part of the R/3 System.

As a rule, an ABAP program containing database-specific SQL statements will **not** run under different database systems. If your program will be used on more than one database platform, only use Open SQL statements.

Native SQL Statements in ABAP Programs

To use a Native SQL statement, you must precede it with the EXEC SQL statement, and follow it with the ENDEXEC statement as follows:

```
EXEC SQL [PERFORMING <form>].
  <Native SQL statement>
ENDEXEC.
```

There is no period after Native SQL statements. Furthermore, using inverted commas (“) or an asterisk (*) at the beginning of a line in a native SQL statement does not introduce a comment as it would in normal ABAP syntax. You need to know whether table and field names are case-sensitive in your chosen database.

In Native SQL statements, the data is transported between the database table and the ABAP program using host variables. These are declared in the ABAP program, and preceded in the Native SQL statement by a colon (:). You can use elementary structures as host variables. Exceptionally, structures in an INTO clause are treated as though all of their fields were listed individually.

If the selection in a Native SQL SELECT statement is a table, you can pass it to ABAP line by line using the PERFORMING addition. The program calls a subroutine <form> for each line read. You can process the data further within the subroutine.

As in Open SQL, after the ENDEXEC statement, SY-DBCNT contains the number of lines processed. In nearly all cases, SY-SUBRC contains the value 0 after the ENDEXEC statement. Cursor operations form an exception: After FETCH, SY-SUBRC is 4 if no more recors could be read. This also applies when you read a result set using EXEC SQL PERFORMING.

```
REPORT demo_native_sql.

DATA: BEGIN OF wa,
        connid   TYPE spfli-connid,
        cityfrom TYPE spfli-cityfrom,
        cityto   TYPE spfli-cityto,
      END OF wa.

DATA c1 TYPE spfli-carrid VALUE 'LH'.

EXEC SQL PERFORMING loop_output.
  SELECT connid, cityfrom, cityto
  INTO   :wa
  FROM   spfli
```

```
  WHERE  carrid = :c1
ENDEXEC.

FORM loop_output.
  WRITE: / wa-connid, wa-cityfrom, wa-cityto.
ENDFORM.
```

The output is as follows:

```
0400 FRANKFURT            NEW YORK
2402 FRANKFURT            BERLIN
0402 FRANKFURT            NEW YORK
```

The program uses the work area WA and the field C1 in the Native SQL SELECT statement. WA is the target area into which the selected data is written. The structure WA in the INTO clause is treated as though its components were all listed individually. INTO :WA-CONNID, :WA-CITYFROM, :WA-CITYTO. C1 is used in the WHERE clause. The subroutine LOOP_OUTPUT writes the data from WA to the screen.

Scope of Native SQL

Native SQL allows you to execute (nearly) all statements available through the SQL programming interface (usually known as SQL Call Interface or similar) for executing SQL program code directly (using EXEC IMMEDIATE or a similar command). The following sections list the statements that are not supported.

Native SQL and the Database Interface

Native SQL statements bypass the R/3 database interface. There is no table logging, and no synchronizatino with the database buffer on the application server. For this reason, you should, wherever possible, use Open SQL to change database tables declared in the ABAP Dictionary. In particular, tables declared in the ABAP Dictionary that contain long columns with the types LCHR or LRAW should only be addressed using Open SQL, since the columns contain extra, database-specific length information for the column. Native SQL does not take this information into account, and may therefore produce incorrect results. Furthermore, Native SQL does not support automatic client handling. Instead, you must treat client fields like any other.

Native SQL and Transactions

To ensure that transactions in the R/3 System are consistent, you should not use any transaction control statements (COMMIT, ROLLBACK WORK), or any statements that set transaction parameters (isolation level...) using Native SQL.

Stored Procedures

To standardize the specific syntax of different database products, ABAP has a uniform syntax:

```
EXECUTE PROCEDURE <name> ( <parameter list> )
```

The parameters are separated by commas. You must also specify whether the parameter is for input (IN), output (OUT) or input and output (INOUT). For further information, refer to SAPnet note 44977.

```
EXEC SQL
   EXECUTE PROCEDURE proc1 ( IN :x, OUT :y )
ENDEXEC.
```

Cursor Processing

Cursor processing in Native SQL is similar to that in Open SQL:

- `OPEN <cursor name> FOR <statement>`
- `FETCH NEXT <cursor name> INTO <target(s)>.`
- `CLOSE <cursor name>`

```
EXEC SQL
  OPEN c1 FOR
    SELECT client, arg1, arg2 FROM table_001
     WHERE client = '000' AND arg2 = :arg2
ENDEXEC.
DO.
  EXEC SQL.
    FETCH NEXT c1 INTO :wa-client, :wa-arg1, :wa-arg2
  ENDEXEC.
  IF sy-subrc <> 0.
    EXIT.
  ELSE.
    <verarbeite Daten>
  ENDIF.
ENDDO.
EXEC SQL.
  CLOSE c1
ENDEXEC.
```

This example opens a cursor, reads data line by line, and closes the cursor again. As in Open SQL, SY-SUBRC indicates whether a line could be read.

Data Types and Conversions

Using Native SQL, you can

- Transfer values from ABAP fields to the database
- Read data from the database and process it in ABAP programs.

Native SQL works without the administrative data about database tables stored in the ABAP Dictionary. Consequently, it cannot perform all of the consistency checks used in Open SQL. This places a larger degree of responsibility on application developers to work with ABAP fields of the correct type. You should always ensure that the ABAP data type and the type of the database column are identical.

If the database table is not defined in the ABAP Dictionary, you cannot refer directly to its data type. In this case, you should create a uniform type description in the ABAP Dictionary, which can then be used by all application programs.

If the table is defined in the ABAP Dictionary, you should remember that the sequence of fields in the ABAP Dictionary definition may not be the same as the actual sequence of fields in the

database. Using the asterisk (*) in the SELECT clause to read all columns into a corresponding work area would lead to meaningless results. In the worst case, it would cause an error.

The Native SQL module of the database interface passes a description of the type, size, and memory location of the ABAP fields used to the database system. The relevant database system operations are usually used to access and convert the data. You can find details of these operations in the manuals for the programming interface of the relevant database system. In some cases, Native SQL also performs other compatibility checks.

The documentation from the various database manufacturers provides detailed lists of combinations of ABAP data types and database column types, both for storing ABAP field values in database tables (INSERT, UPDATE) and for reading database contents into ABAP fields (SELECT). You can also apply these descriptions for the input and output parameters of database procedures. Any combinations not listed there are undefined, and should not be used.

The following sections provide details of the data types and conversions for individual databases. Although they are database-specific, there are also some common features.

Recommended type combinations are underlined. Only for these combinations is behavior guaranteed from release to release. For any other combinations, you should assume that the description only applies to the specified release.

The results of conversions are listed in a results column:

- “OK”: The conversion can be performed without loss of data.
- Operations that fail are indicated by their SQL error code. Errors of this kind always lead to program termination and an ABAP short dump.
- In some cases, data is transferred without an SQL error occurring. However, the data is truncated, rounded, or otherwise unusable:
  - [rtrunc] Right truncation.
    “Left” or “right” applies to the normal way of writing a value. So, for example, if a number is truncated, its decimal places are affected.
  - [ltrunc]: Left truncation
  - [round]: Number is rounded up or down during conversion
  - [0]: A number that was “too small” is rounded to 0 (underflow)
  - [undef]: The conversion result is undefined.
    There are several possible results. The concrete result is either not known at all, or can only be described using a set of rules that is too complicated for practical use.
  - [null]: The conversion returns the SQL value NULL.
  - [ok]: The conversion is performed without fields and unchecked.
    The original data is converted, but without its format being checked. The result may therefore be a value invalid for the result type, which cannot be processed further. An example of this is a date field containing the value “99999999” or “abcdefgh” after conversion.

Combinations of ABAP data type and database column type can be divided into finer subcategories. Here, for example, using the transfer direction ABAP → database (INSERT, UPDATE):

- If the width of the ABAP field is greater than that of the database column, the ABAP field may contain values for which there is not enough space in the database column. This can

produce other cases: The concrete data value in ABAP finds space in the database column, or not.

- If the ABAP field is at most as long as the database column, there is always space for the ABAP value in the database column.
- Some types, such as numeric columns, expect values in a particular format. This is particularly important in connection with character types, for example, when you want to write an ABAP character field (type C) into an integer column.

Native SQL for Oracle [Page 1119]

Native SQL for Informix [Page 1137]

Native SQL for DB2 Common Server [Page 1152]

Native SQL for Oracle

The Oracle-specific section of the Native SQL documentation deals with combinations of ABAP data types and column types of an Oracle table. It describes in detail the type conversions used for each type combination. The abbreviations introduced in the general description, such as [*rtrunc*] are used.

The list is divided roughly into the two data transfer directions:

- Storing values from ABAP fields in database columns INSERT).
- Reading values from database columns into ABAP fields (SELECT).

Each description assumes that the starting value is a permitted value for the corresponding type. It does not take account of cases where invalid bit patterns are created in an ABAP field due, for example, to field symbol operations.

INSERT

(A) The following applies to the database column type number(x,y), including the case y < 0:

- Significant figures are those with values >= 10^{-y}.
- Non-significant figures: Those with value < 10^{-y}.

ABAP Data Type: Character C(n)

Data value width: String length without trailing spaces

Database column type	**Case (*notes*)**	**Result**
varchar2 (z)	ABAP data value width > 4000	**1461**
	ABAP field width (n) > database column width (z) ABAP data value width > database column width ABAP data value width <= database column width	 **1401** OK
	ABAP field width (n) <= database column width (z)	OK
char (z)	ABAP data value width > 4000	**1461**
	ABAP field width (n) > database column width (z) ABAP data value width > database column width ABAP data value width >= database column width	 **1401** OK
	ABAP field width (n) <= database column width (z)	OK
long		OK
number (z)	ABAP data value is non-numeric	**1722**
	Loss of figures before decimal point in conversion	**1438**
	ABAP data value contains decimal places	[*round*]

	ABAP data value width <= database column width (z) [*data value width*. *Number of figures before decimal point*]	ok / [*round*]
number (z,y)	ABAP data value is non-numeric	**1722**
	Loss of significant figures (see (A)).	**1438**
	Loss of non-significant figures (see (A)).	[*round*]
	ABAP data value width <= database column width (z) [*data value width*. *Number of significant figures* (see (A)).	ok / [*round*]
float	ABAP data value is non-numeric	**1722**
	ABAP data value not in interval] -10^{126}, 10^{126}[	**1426**
	ABAP data value in interval] -10^{-130}, 10^{-130} [[Database data value rounded to 0]	[*0*]
	ABAP data value in interval] -10^{126}, 10^{126} [[*38th place is rounded, filled with zeros from the right*]	ok / [*0*] / [*round*]
date (9)	ABAP data value has an invalid database date format	**1861**
	ABAP date value has a valid database date format [*10-Jan-97*]	OK
raw (z)	ABAP data value width > 4000	**1461**
	ABAP data value has an invalid hexadecimal format [*Valid values are: 0-9, A-F, a-f. Upper-/lowercase irrelevant in the conversion]*	**1465**
	ABAP field width (n) > database column width (n>2z) ABAP data value width > database column width ABAP data value width <= database column width [*If the data value width has an odd-numbered length, a zero is added at the left*]	 **1401** OK
	ABAP data value width (n) <= database column width (n<=2z) [*If the data value width has an odd-numbered length, a zero is added at the left*]	OK
long raw	ABAP data value width > 4000	**1461**
	ABAP data value has an invalid hexadecimal format [*Valid values are:* 0-9, A-F, a-f. Upper-/lowercase irrelevant in the conversion]	**1465**
	ABAP data value width <= 4000	OK

ABAP Data Type: Numeric Text N(n)

Data value width: Number of digits without leading zeros

Database column type	**Case (*notes*)**	**Result**

varchar2 (z)	Data value width > 4000	**1461**
	ABAP field width (n) > database column width (z)	**1401**
	ABAP field width (n) <= database column width (z)	OK
char (z)	Data value width > 4000	**1461**
	ABAP field width (n) > database column width (z)	**1401**
	ABAP field width (n) <= database column width (z)	OK
long		OK
number (z)	ABAP field width (n) > database column width (z) ABAP data value width > database column width ABAP data value width <= database column width [*leading zeros truncated*]	 **1438** ok / [*ltrunc*]
	ABAP field width (n) = database column width (z) ABAP data value width <= database column width [*leading zeros truncated*]	 ok / [*ltrunc*]
	ABAP field width (n) < database column width (z)	OK
number (x,y) (z=x-y)	Loss of significant figures (see (A)).	**1438**
	ABAP data value width <= database column width (z) [*leading zeros truncated*]	ok / [*ltrunc*]
float	ABAP data value >= 10^{126}	**1426**
	ABAP data value < 10^{126} [[*38th place is rounded, filled with zeros from the right*]	ok / [*round*]
date		**1861**
raw (z)	ABAP field width (n) > database column width (n>2z)	**1401**
	ABAP data value width (n) <= database column width (n<=2z) [*If the data value width has an odd-numbered length, a zero is added at the left*]	OK
long raw	Data value width > 4000	**1461**
	Data value width <= 4000	OK

ABAP Data Type: Packed Number P(m,d)

m= bytes, d = decimal places

Data value width (n): Number of digits without leading zeros, n<=2m-1; n is increased by one for each of decimal point and minus sign.

Database column type	**Case (*notes*)**	**Result**

varchar2 (z)	ABAP data value width (n) < database column width (z)	**1401**
	ABAP data value width (n) <= database column width (z)	OK
char (z)	ABAP data value width (n) < database column width (z)	**1401**
	ABAP data value width (n) <= database column width (z)	OK
long		OK
number (z)	Loss of figures before decimal point in conversion	**1438**
	ABAP data value contains decimal places	[*round*]
	ABAP data value width (n) <= database column width (z) [*data value width*. *Number of figures before decimal point*]	ok / [*round*]
number (x,y) (z=x-y)	Loss of significant figures (see (A)).	**1438**
	Loss of non-significant figures (see (A)).	[*round*]
	ABAP data value width <= database column width (z) [*data value width*. *Number of significant figures* (see (A)).	ok / [*round*]
float		OK
date		**932**
raw		**932**
long raw		**932**

ABAP Data Type: Integer I

Data value width: Number of digits without leading zeros; n is increased by one if there is a minus sign.

Database column type	**Case (*notes*)**	**Result**
varchar2 (z)	ABAP data value width > database column width (z)	**1401**
	ABAP data value width <= database column width (z)	OK
char (z)	ABAP data value width > database column width (z)	**1401**
	ABAP data value width <= database column width (z)	OK
long		OK
number (z)	ABAP data value width > database column width (z) [*data value width*. *Number of digits*]	**1438**
	ABAP data value width <= database column width (z) [*data value width*. *Number of digits*]	OK
number (x,y) (z=x-y)	Loss of significant figures (see (A)).	**1438**

	Loss of non-significant figures (see (A)).	[*round*]
	ABAP data value width <= database column width (z) [*data value width*. *Number of significant figures* (see (A)).	ok / [*round*]
float		OK
date		**932**
raw		**932**
long raw		**932**

ABAP Data Type: Float F

Inaccuracies can occur with any conversions involving floating point numbers.

Data value width: Number of digits without leading zeros; n is increased by one for each of decimal point and minus sign.

Database column type	**Case (*notes*)**	**Result**
varchar2 (z)	ABAP data value width > database column width (z)	**1401**
	ABAP data value width <= database column width (z)	OK
char (z)	ABAP data value width > database column width (z)	**1401**
	ABAP data value width <= database column width (z)	OK
long		OK
number (z)	ABAP data value width > database column width (z) [*data value width*. *Number of figures before decimal point*]	**1438**
	ABAP data value contains decimal places	[*round*]
	ABAP data value width <= database column width (z) [*data value width*. *Number of figures before decimal point*]	ok / [*round*]
number (x,y) (z=x-y)	Loss of significant figures (see (A)).	**1438**
	Loss of non-significant figures (see (A)).	[*round*]
	ABAP data value width <= database column width (z) [*data value width*. *Number of significant figures* (see (A)).	ok / [*round*]
float	ABAP data value not in interval] -10^{126}, 10^{126}	**1426**
	ABAP data value in interval] -10^{-130}, 10^{-130} [[Database data value rounded to 0]	[*0*]
	ABAP data value not in interval] -10^{126}, 10^{126} [	ok / [*0*]

date		**932**
raw		**932**
long raw		**932**

ABAP Data Type: Date D(8)

Data value width = 8 (fixed)

Database column type	Case (*notes*)	Result
varchar2 (z)	ABAP data value width (8) > database column width (z)	**1401**
	ABAP data value width (8) <= database column width (z)	OK
char (z)	ABAP data value width (8) > database column width (z)	**1401**
	ABAP data value width (8) <= database column width (z)	OK
long		OK
number (z)	ABAP data value is non-numeric	**1722**
	ABAP data value width (8) > database column width (z)	**1438**
	ABAP data value width (8) <= database column width (z)	OK
number (x,y) (z=x-y)	ABAP data value is non-numeric	**1722**
	ABAP data value width (8) > database column width (z)	**1438**
	ABAP data value width (8) <= database column width (z)	OK
float	ABAP data value is non-numeric	**1722**
	ABAP data value is numeric	ok
date		**1861**
raw (z)	ABAP data value has an invalid hexadecimal format [*Valid values are: 0-9, A-F, a-f. Upper-/lowercase irrelevant in the conversion*]	**1465**
	ABAP data value width (8) > database column width (4>z)	**1401**
	ABAP data value width (8) <= database column width (4<=z)	OK
long raw	ABAP data value has an invalid hexadecimal format [*Valid values are: 0-9, A-F, a-f. Upper-/lowercase irrelevant in the conversion*]	**1465**
	ABAP data value has a valid hexadecimal format	OK

ABAP Data Type: Time T(6)

Data value width = 6 (fixed)

Database column type	Case (*notes*)	Result
varchar2 (z)	ABAP data value width (6) > database column width (z)	**1401**
	ABAP data value width (6) <= database column width (z)	OK
char (z)	ABAP data value width (6) > database column width (z)	**1401**
	ABAP data value width (6) <= database column width (z)	OK
long		OK
number (z)	ABAP data value is non-numeric	**1722**
	ABAP data value width (6) > database column width (z)	**1438**
	ABAP data value width (6) <= database column width (z)	OK
number (x,y) (z=x-y)	ABAP data value is non-numeric	**1722**
	ABAP data value width (6) > database column width (z)	**1438**
	ABAP data value width (6) <= database column width (z)	OK
float	ABAP data value is non-numeric	**1722**
	ABAP data value is numeric	OK
date		**1861**
raw (z)	ABAP data value has an invalid hexadecimal format [*Valid values are: 0-9, A-F, a-f. Upper-/lowercase irrelevant in the conversion*]	**1465**
	ABAP data value width (6) > database column width (3>z)	**1401**
	ABAP data value width (6) <= database column width (3<=z)	OK
long raw	ABAP data value has an invalid hexadecimal format [*Valid values are: 0-9, A-F, a-f. Upper-/lowercase irrelevant in the conversion*]	**1465**
	ABAP data value has a valid hexadecimal format	OK

ABAP Data Type: Hexadecimal X(n)

Data value width: Number of hexadecimal digits without trailing zeros

Database column type	Case (*notes*)	Result
varchar2 (z)	n > 4000	**1461**
	ABAP field width (2n) > database column width (z), (2n > z)	**1401**
	ABAP field width 2n = 8000, database column width = 4000	[*rtrunc ½*]
	ABAP field width (2n) <= database column width (z), (2n <= z)	OK

char (z)	ABAP field width (2n) > database column width (z), (2n > z)	**1401**
	ABAP field width (2n) <= database column width (z), (2n <= z)	OK
long	n > 4000	**1461**
	n <= 4000	[*rtrunc ½*]
number (z)		**932**
number (z,y)		**932**
float		**932**
date		**932**
raw (z)	n > 4000	**1461**
	ABAP field width (n) > database column width (z), (n>z)	**1401**
	ABAP field width (n)<= database column width (z), (n<=z)	OK
long raw	n > 4000	[*rtrunc*]
	n < 4000	OK

SELECT

(B) ABAP data types Date, Time

Converting a database value into an ABAP date or time field is like converting it into an ABAP character field with length 8 or 6 respectively. It is technically possible to convert an invalid format, but this returns an ABAP field value with an invalid format, which you cannot usefully use. Conversions of this kind are indicated in the table with [*ok*].

Conversion of database data values to

- ABAP date format require the numeric form YYYYMMDD.
- ABAP time format require the numeric form HHMMSS.

(C) ABAP data type Numeric text

Conversions into the ABAP type numeric text N(n) behave like conversions into an ABAP character field C(n), that is, the field contents are left-justified and filled with trailing spaces. This is contrary to the numeric text definition, according to which a numeric text field may only contain digits, and is filled with leading zeros. The numeric text field may contain a string with an invalid numeric text format following the conversion. Conversions of this kind are indicated in the table with [*ok*].

(D) ABAP data type hexadecimal

If the data value width is less than the field width, a result string is transferred (if the data value contains an odd number of characters, a leading zero is added). The rest of the field length is not filled with trailing zeros, but with undefined values.

Database Field Type varchar2 (z)

Data value width: String length (trailing spaces are not saved)

ABAP Data type	Case (*notes*)	Result
Character C(n)	Database column width (z) > ABAP field width (n)	
	Database data value width > ABAP field width	[*rtrunc*]
	Database data value width <= ABAP field width	OK
	Database column width (z) <= ABAP field width (n)	OK
Numeric text (N) (see (C))	[*Conversion as for character fields left-justified, filled with trailing spaces*]	[*ok*]
Packed Number P(m,d) (n=2m-1)	Database data value is non-numeric	**1722**
	Loss of figures before decimal point in conversion	[*null*]
	Loss of decimal places in conversion	[*rtrunc*]
	Database data value width <= ABAP field width (n-d) [*Data value or field width. Number of figures before decimal point*]	ok / [*rtrunc*]
Integer I	Database data value is non-numeric	**1722**
	ABAP data value not in interval] -10^{126}, 10^{126} [	**1455**
	Database data value contains decimal places	[*rtrunc*]
	ABAP data value in interval] -2^{31}, 2^{31} [	ok / [*rtrunc*]
Float F	Database data value is non-numeric	**1722**
	Database data value >= 10^{126} [Result is max. database float value (1E+126) or max. ABAP float value (1.979E+308)]	[*undef*]
	Database data value <= -10^{126} [Result is zero or min. ABAP float value (-1.797E+308)]	[*undef*]
	Database data value in interval] -10^{-130}, 10^{-130} [[*Result rounded to 0*]	[*0*]
	Truncation in conversion [*Accuracy in floating point limited to approx. 15 decimal places*]	[*round*]
	Database data value in interval] -10^{126}, 10^{126} [[Possible inaccuracies in conversion]	ok / [*0*] / [*round*]

Date D(8) (see (B))	Database column width (z) > ABAP field width (8) Database data value width > ABAP field width Database data value width <= ABAP field width	 [*rtrunc*] [*ok*]
	Database column width (z) <= ABAP field width (8)	[*ok*]
Time T(6) (see (B))	Database column width (z) > ABAP field width (6) Database data value width > ABAP field width Database data value width <= ABAP field width	 [*rtrunc*] [*ok*]
	Database column width (z) <= ABAP field width (6)	[*ok*]
Hexadecimal X(n) (see (D))	Database data value has an invalid hexadecimal format [*Valid values are: 0-9, A-F, a-f. Upper-/lowercase irrelevant in the conversion*]	**1465**
	Database data value width (x) > ABAP field width, (x <= z, x > 2n) Database data value width (x) = ABAP field width, (x <= z, x = 2n) Database data value width (x) < ABAP field width, (x <= z, x < 2n) [*Filled with trailing undefined values*]	[*rtrunc*] OK [*undef*]

Database Field Type char (z)

Data value width: String length (trailing spaces are not saved)

ABAP Data type	**Case (*notes*)**	**Result**
Character C(n)	Database column width (z) > ABAP field width (n) Database data value width > ABAP field width Database data value width <= ABAP field width	 [*rtrunc*] OK
	Database column width (z) <= ABAP field width (n)	OK
Numeric text (N) (see (C))	[*Conversion as for character fields* *left-justified, filled with trailing spaces*]	[*ok*]
Packed Number P(m,d) (n=2m-1)	Database data value is non-numeric	**1722**
	Loss of figures before decimal point in conversion	*[null]*
	Loss of decimal places in conversion	[*rtrunc*]

	Database data value width <= ABAP field width (n-d) [*Data value or field width. Number of figures before decimal point*]	ok / [*rtrunc*]
Integer I	Database data value is non-numeric	**1722**
	ABAP data value not in interval] -10^{126}, 10^{126} [	**1455**
	Database data value contains decimal places	[*rtrunc*]
	ABAP data value in interval] -2^{31}, 2^{31} [	ok / [*rtrunc*]
Float F	Database data value is non-numeric	**1722**
	Database data value >= 10^{126} [Result is max. database float value (1E+126) or max. ABAP float value (1.979E+308)]	[*undef*]
	Database data value <= -10^{126} [Result is zero or min. ABAP float value (-1.797E+308)]	[*undef*]
	Database data value in interval] -10^{-130}, 10^{-130} [[*Result rounded to 0*]	[*0*]
	Truncation in conversion [*Accuracy in floating point limited to approx. 15 decimal places*]	[*round*]
	Database data value in interval] -10^{126}, 10^{126} [[Possible inaccuracies in conversion]	ok / [*0*] / [*round*]
Date D(8) (see (B))	Database column width (z) > ABAP field width (8) Database data value width > ABAP field width Database data value width <= ABAP field width	 [*rtrunc*] [*ok*]
	Database column width (z) <= ABAP field width (8)	[*ok*]
Time T(6) (see (B))	Database column width (z) > ABAP field width (6) Database data value width > ABAP field width Database data value width <= ABAP field width	 [*rtrunc*] [*ok*]
	Database column width (z) <= ABAP field width (6)	[*ok*]
Hexadecimal X(n) (see (D))	Database data value has an invalid hexadecimal format [*Valid values are: 0-9, A-F, a-f. Upper-/lowercase irrelevant in the conversion*]	**1465**
	Database column width (z) > ABAP field width (n), (z>2n) Database data value width (x) > ABAP field width, (x <= z, x > 2n) Database data value width (x) <= ABAP field width, (x <= z, x <= 2n)	 [*rtrunc*] **1465**

	Database column width (z) = ABAP field width(n), (z=2n) Database data value width (x) = ABAP field width, (x <= z, x = 2n) Database data value width (x) < ABAP field width, (x <= z, x < 2n)	ok **1465**
	Database column width (z) < ABAP field width (n), (z <= 2n) [*Filled with trailing undefined values*]	[*undef*]

Database Field Type long

Data value width: String length (trailing spaces are not saved)

ABAP Data type	**Case (*notes*)**	**Result**
Character C(n)	Database data value width > ABAP field width	[*rtrunc*]
	Database data value width <= ABAP field width	OK
Numeric text (N) (see (C))	[*Conversion as for character fields left-justified, filled with trailing spaces*]	[*ok*]
Packed P(m,d) (n=2m-1)	Data value is non-numeric	**1722**
	Loss of digits before the decimal point	[null]
	Loss of digits after the decimal point	[*rtrunc*]
	Database data value width <= ABAP field width (n-d) [*Data value width: Number of figures before decimal point*]	ok / [*rtrunc*]
Integer I		[*undef*]
Float F		[*undef*]
Date D(8) (see (B))	Database data value width > ABAP field width (8)	[*rtrunc*]
	Database data value width <= ABAP field width (8)	[*ok*]
Time T(6) (see (B))	Database data value width > ABAP field width (6)	[*rtrunc*]
	Database data value width <= ABAP field width (6)	[*ok*]
Hexadecimal X(n) (see (D))		[*undef*]

Database Field Type number (z)

Data value width: Number of digits without leading zeros; n is increased by one if there is a minus sign.

ABAP Data type	**Case (*notes*)**	**Result**
Character C(n)	Database data value width > ABAP field width (n)	[*rtrunc*]
	Database data value width <= ABAP field width (n)	OK
Numeric text (N) (see (C))	[*Conversion as for character fields left-justified, filled with trailing spaces*]	[*ok*]
Packed P(m,d) (n=2m-1)	Loss of figures before decimal point in conversion	[null]
	Database data value width <= ABAP field width (n-d) [*Data value width: Number of figures before decimal point*]	OK
Integer I	ABAP data value not in interval] -2^{31}, 2^{31} [	**1455**
	ABAP data value in interval] -2^{31}, 2^{31} [	OK
Float F	Truncation in conversion [*Accuracy in floating point limited to approx. 15 decimal places*]	[*round*]
	Database data value $\in$] -10^{126}, 10^{126} [[Possible inaccuracies in conversion]	ok / [*round*]
Date D(8) (see (B))	Database column width (z) > ABAP field width (8)	
	Database data value width > ABAP field width	[*rtrunc*]
	Database data value width <= ABAP field width	[*ok*]
	Database column width (z) <= ABAP field width (8)	[*ok*]
Time T(6) (see (B))	Database column width (z) > ABAP field width (6)	
	Database data value width > ABAP field width	[*rtrunc*]
	Database data value width <= ABAP field width	[*ok*]
	Database column width (z) <= ABAP field width (6)	[*ok*]
Hexadecimal X(n) (see (D))		**932**

Database Field Type number (z,y)

Data value width: Number of digits (z) without leading zeros; n is increased by one for each of decimal point and minus sign.

ABAP Data type	Case (*notes*)	Result
Character C(n)	Database data value width > ABAP field width (n)	[*rtrunc*]
	Database data value width <= ABAP field width (n)	OK
Numeric text (N) (see (C))	[*Conversion as for character fields left-justified, filled with trailing spaces*]	[*ok*]
Packed P(m,d) (n=2m-1)	Loss of significant figures (see (A)).	[null]
	Loss of non-significant figures (see (A)).	[*rtrunc*]
	Database data value width <= ABAP field width (n-d) [*Data value width: Number of figures before decimal point*]	ok / [*rtrunc*]
Integer I	ABAP data value not in interval] -2^{31}, 2^{31} [	**1455**
	Database data value contains decimal places	[*rtrunc*]
	ABAP data value in interval] -2^{31}, 2^{31} [	ok / [*rtrunc*]
Float F	Truncation in conversion [*Accuracy in floating point limited to approx. 15 decimal places*]	[*round*]
	Database data value in interval] -10^{126}, 10^{126} [[Possible inaccuracies in conversion]	ok / [*round*]
Date D(8) (see (B))	Database column width (z) > ABAP field width (8), (z>=8)	
	Database data value width > ABAP field width	[*rtrunc*]
	Database data value width <= ABAP field width	[*ok*]
	Database column width (z) <= ABAP field width (8), (z<8)	[*ok*]
Time T(6) (see (B))	Database column width (z) > ABAP field width (6), (z>=6)	
	Database data value width > ABAP field width	[*rtrunc*]
	Database data value width <= ABAP field width	[*ok*]
	Database column width (z) <= ABAP field width (6), (z<6)	[*ok*]
Hexadecimal X(n) (see (D))		**932**

Database Field Type float

Data value width: Number of digits without leading zeros; n is increased by one for each of decimal point and minus sign.

ABAP Data type	Case (*notes*)	Result

Character C(n)	Database data value width > ABAP field width (n)	[*rtrunc*]
	Database data value width <= ABAP field width (n)	OK
Numeric text (N) (see (C))	[*Conversion as for character fields left-justified, filled with trailing spaces*]	[*ok*]
Packed P(m,d) (n=2m-1)	Loss of figures before decimal point in conversion	[null]
	Loss of decimal places in conversion	[*rtrunc*]
	Database data value width <= ABAP field width (n-d) [*Data value width: Number of figures before decimal point*]	ok / [*rtrunc*]
Integer I	ABAP data value not in interval] -2^{31}, 2^{31} [	**1455**
	Database data value contains decimal places	[*rtrunc*]
	Database data value in interval] -2^{31}, 2^{31} [	ok / [*rtrunc*]
Float F	Truncation in conversion [*Accuracy in floating point limited to approx. 15 decimal places*]	[*round*]
	Database data value in interval] -10^{126}, 10^{126} [[Possible inaccuracies in conversion]	ok / [*round*]
Date D(8) (see (B))	Database data value width > ABAP field width (8)	[*rtrunc*]
	Database data value width <= ABAP field width (8)	[*ok*]
Time T(6) (see (B))	Database data value width > ABAP field width (6)	[*rtrunc*]
	Database data value width <= ABAP field width (6)	[*ok*]
Hexadecimal X(n) (see (D))		**932**

Database Field Type date(9)

Data value width = 9 (fixed)

ABAP Data type	**Case (*notes*)**	**Result**
Character C(n)	Database data value width (9) > ABAP field width (n), (n<9)	[*rtrunc*]
	Database data value width (9) <= ABAP field width (n), (n>=9)	OK
Numeric text (N) (see (C))	[*Conversion as for character fields left-justified, filled with trailing spaces*]	[*ok*]

Packed P		**932**
Integer I		**932**
Float F		**932**
Date D(8) (see (B))		ok / [*rtrunc*]
Time T(6) (see (B))		ok / [*rtrunc*]
Hexadecimal X(n) (see (D))		**932**

Database Field Type raw(x)

Column width: z = 2x

Data value width: Number of hexadecimal digits without trailing zeros

ABAP **Data type**	**Case (*notes*)**	**Result**
Character C(n)	Database column width (z) > ABAP field width (n), (z>n)	
	Database data value width > ABAP field width	[*rtrunc*]
	Database data value width <= ABAP field width	OK
	Database column width (z) <= ABAP field width (n), (z<=n)	OK
Numeric text (N) (see (C))	[*Conversion as for character fields* *left-justified, filled with trailing spaces*]	[*ok*]
Packed P		**932**
Integer I		**932**
Float F		**932**
Date D(8) (see (B))	Database column width (z) > ABAP field width (8), (z>8)	
	Database data value width > ABAP field width	[*rtrunc*]
	Database data value width <= ABAP field width	[*ok*]
	Database column width (z) <= ABAP field width (8), (z<=8)	[*ok*]
Time T(6) (see (B))	Database column width (z) > ABAP field width (6), (z<6)	
	Database data value width > ABAP field width	[*rtrunc*]
	Database data value width <= ABAP field width	[*ok*]
	Database column width (z) <= ABAP field width (6), (z<=6)	[*ok*]

Hexadecimal X(n) (see (D))	Database data value width > ABAP field width Database data value width = ABAP field width Database column width < ABAP field width [*Filled with trailing undefined values*]	[*rtrunc*] OK [*undef*]

Database Field Type long raw

Data value width: Number of hexadecimal digits without trailing zeros

ABAP Data type	**Case (*notes*)**	**Result**
Character C(n)	Database data value width <= ABAP field width	OK
	Database data value width > ABAP field width	[*rtrunc*]
Numeric text (N) (see (C))	[*Conversion as for character fields left-justified, filled with trailing spaces*]	[*ok*]
Packed P		**932**
Integer I		**932**
Float F		**932**
Date D(8) (see (B))	Database data value width > ABAP field width (8)	[*rtrunc*]
	Database data value width <= ABAP field width (8)	[*ok*]
Time T(6) (see (B))	Database data value width > ABAP field width (6)	[*rtrunc*]
	Database data value width <= ABAP field width (6)	[*ok*]
Hexadecimal X(n) (see (D))	Database data value width > ABAP field width (2n)	[*rtrunc*]
	Database data value width = ABAP field width (2n)	OK
	Database column width < ABAP field width (2n) [*Filled with trailing undefined values*]	[*undef*]

Database Field Type rowid(18)

Data value width = 18 (fixed)

ABAP Data type	**Case (*notes*)**	**Result**
Character C(n)	Database data value width (18) > ABAP field width (n), (18 > 9)	[*rtrunc*]
	Database data value width (18) <= ABAP field width (n), (18 <= 9)	OK

Numeric text (N) (see (C))	[*Conversion as for character fields left-justified, filled with trailing spaces*]	[*ok*]
Packed P		**932**
Integer I		**932**
Float F		**932**
Date D(8) (see (B))		[*undef*]
Time T(6) (see (B))		[*undef*]
Hexadecimal X(n) (see (D))		**932**

Native SQL for Informix

The principal new features in the Native SQL interface from Release 4.0 are:

- Ability to connect to several databases in parallel (including non-SAP databases)
- Use of the cursor for stored procedures
- Access to non-SAP tables
- Ability to use (almost) all Native SQL statements for Informix The Native SQL statements that are not supported are listed below.

Isolation Levels

All SET ISOLATION LEVEL statements last not only within EXEC SQL - ENDEXEC, but also in subsequent statements, for example, in Open SQL for the same database connection. This can adversely affect the lock mechanism. To prevent this, you should reset the isolation level to DIRTY READ.

Furthermore, do not use the above statements in stored procedures, since they cannot be compared with the DBSL.

Data types

When you use Native SQL and host variables to access SAP tables in Informix, you should use work areas or the LIKE statement.

When accessing non-SAP tables, you must ensure that the ABAP variable type and the database field type are compatible, since the Native SQL module works without information from the ABAP Dictionary. The Native SQL module opens the type and memory area reserved in ABAP to the DBMS. This allows you to read and write directly to and from it. This means that the module behaves, with a few exceptions, exactly as though you were using ESQL/C. In other words, some conversions are not allowed and trigger error messages, and rounding errors and truncation are also possible. The principal data type descriptions and conversions for Informix databases are described in the **Programmers Manual**.

For example, if you use Native SQL to attempt to convert a DATE value into an ABAP variable with type I, the system will be unable to perform the conversion.

Below is a description of the various type compatibilities between ABAP variables and Informix database field types.

It lists all of the type conversions that are permitted and supported by the Native SQL module. Preferred type combinations are shown in bold type.

For illegal type combinations, the error procedure of the ABAP Workbench is given, along with the SQL error code of any resulting ABAP short dump. Remember, however, that this code can change from release to release.

There is no formal description of how other type combinations not listed here behave.

The first section deals with using Native SQL (INSERT and UPDATE) to save ABAP variables in non-SAP database tables. The second part describes the opposite direction - how to read from external database tables into ABAP variables using Native SQL (SELECT).

INSERT and UPDATE

Saving values from ABAP variables using Native SQL.

Each of the following tables represents a single ABAP data type. This is the type of the ABAP variable whose value you want to save.

In the left-hand column is the **SQL data type** of the database field of the external database table.

On the right is the reaction of the ABAP Workbench.

The description of the SQL data type DECIMAL applies also to the types MONEY and NUMERIC.

ABAP Data Type: C

The ABAP data type C can be reproduced in most databases. However, where there are non-CHAR fields, you must take care with length, format, and permitted value ranges, since rounding, truncation, or format errors can easily occur. Truncation and rounding errors are not returned as SQL errors.

DB **Data type**	**Test case [*notes*]**	**Result**
char	ABAP data value width > DB column width	*[rtrunc]*
	ABAP data value width <= DB column width	OK
varchar	ABAP data value width > DB column width	*[rtrunc]*
	ABAP data value width <= DB column width	OK
nchar	ABAP data value width > DB column width	*[rtrunc]*
	ABAP data value width <= DB column width	OK
text	Occurs in a DELETE statement, or text field occurs in the WHERE clause	**-615**
	otherwise	OK
byte		**-608**
date	ABAP data value does not have a valid DB date format	*[undef]*
	ABAP date value has a valid DB date format, but wrong value range	**-608**
	ABAP data value has a valid DB date format	OK
datetime	ABAP data value does not have a valid DB date time format	**-1262**
	ABAP date value has a valid DB date time format, but wrong value range	**-1218**
	ABAP data value has have a valid DB date time format	OK
interval	ABAP data value does not have a valid DB interval format	**-1264**
	ABAP data value does not have a valid DB interval format	**-1263**

	ABAP data value has a valid DB interval format	OK
decimal	ABAP data value is non-numeric	**-1213**
	Loss of significant figures in conversion	**-1226**
	Loss of non-significant figures	**-1226**
	No loss in conversion	OK
integer	ABAP data value is non-numeric	**-1213**
	Loss of significant figures in conversion	**-1215**
	No loss in conversion	OK
smallint	ABAP data value is non-numeric	**-1213**
	Loss of significant figures in conversion	**-1215**
	No loss in conversion	OK
float	ABAP data value is non-numeric	**-1213**
	Loss of significant figures in conversion	*[round]*
	Loss of non-significant figures	*[round]*
	No loss in conversion	OK
smallfloat	ABAP data value is non-numeric	**-1213**
	Loss of significant figures in conversion	*[round]*
	Loss of non-significant figures	*[round]*
	No loss in conversion	OK
double precision	ABAP data value is non-numeric	**-1213**
	Loss of significant figures in conversion	*[round]*
	Loss of non-significant figures	*[round]*
	No loss in conversion	OK

ABAP Data Type: N

DB Data type	**Test case [*notes*]**	**Result**
char	ABAP field width > DB column width	*[rtrunc]*
	ABAP field width <= DB column width	OK
varchar	ABAP field width > DB column width	*[rtrunc]*
	ABAP field width <= DB column width	OK
nchar	ABAP field width > DB column width	*[rtrunc]*
	ABAP field width <= DB column width	OK

text		**-608**
byte		**-608**
date		**-1218**
datetime, interval	DB column type consists of a single element (for example, datetime hour to hour) and ABAP value range is valid	OK
	otherwise	**-1218,-1261**
decimal	Loss of figures in conversion	**-1226**
	No loss in conversion	OK
integer	Loss of figures in conversion	**-1215**
	No loss in conversion	OK
smallint	Loss of figures in conversion	**-1215**
	No loss in conversion	OK
float	Loss of figures in conversion	*[round]*
	No loss in conversion	OK
smallfloat	Loss of significant figures in conversion	*[round]*
	No loss in conversion	OK
double precision	Loss of significant figures in conversion	*[round]*
	No loss in conversion	OK

ABAP Data Type: P

DB Data type	**Test case [*notes*]**	**Result**
char	ABAP data value width > DB column width	*[rtrunc]*
	ABAP data value width <= DB column width	OK
varchar	ABAP data value width > DB column width	*[rtrunc]*
	ABAP data value width <= DB column width	OK
nchar	ABAP data value width > DB column width	*[rtrunc]*
	ABAP data value width <= DB column width	OK
text		**-608**
byte		**-608**
date		**-1218**

datetime, interval	DB column type consists of a single element (for example, datetime hour to hour) and ABAP value range is valid	OK
	otherwise	**-1218,-1261**
decimal	Loss of figures in conversion	**-1226**
	No loss in conversion	OK
integer	Loss of figures in conversion	**-1215**
	No loss in conversion	OK
smallint	Loss of figures in conversion	**-1215**
	No loss in conversion	OK
float	Loss of figures in conversion	*[round]*
	No loss in conversion	OK
smallfloat	Loss of figures in conversion	*[round]*
	No loss in conversion	OK
double precision	Loss of figures in conversion	*[round]*
	No loss in conversion	OK

ABAP Data Type: I

DB Data type	Test case [*notes*]	Result
char	Loss of figures in conversion	**-1207**
	No loss in conversion	OK
varchar	Loss of figures in conversion	**-1207**
	No loss in conversion	OK
nchar	Loss of figures in conversion	**-1207**
	No loss in conversion	OK
text		**-608**
byte		**-608**
date		**-1218**
datetime, interval		**-1260**
decimal	Loss of figures in conversion	**-1226**
	No loss in conversion	OK
integer	No loss in conversion	OK

smallint	Loss of figures in conversion	**-1214**
	No loss in conversion	OK
float	Loss of figures in conversion	*[round]*
	No loss in conversion	OK
smallfloat	Loss of significant figures in conversion	*[round]*
	No loss in conversion	OK
double precision	Loss of significant figures in conversion	*[round]*
	No loss in conversion	OK

ABAP Data Type: F

DB Data type	**Test case [*notes*]**	**Result**
char	Loss of figures in conversion	**-1207**
	No loss in conversion	OK
varchar	Loss of figures in conversion	**-1207**
	No loss in conversion	OK
nchar	Loss of figures in conversion	**-1207**
	No loss in conversion	OK
text		**-608**
byte		**-608**
date		**-1218**
datetime, interval		**-1260**
decimal	Loss of figures in conversion	**-1226**
	No loss in conversion	OK
integer	Loss of figures in conversion	**-1215**
	No loss in conversion	OK
smallint	Loss of figures in conversion	**-1214**
	No loss in conversion	OK
float	No loss in conversion	OK
smallfloat	Loss of significant figures in conversion	*[round]*
	No loss in conversion	OK
double precision	No loss in conversion	OK

ABAP Data Type: D

DB Data type	Test case [*notes*]	Result
char	ABAP data value width (8) > DB column width	*[rtrunc]*
	ABAP data value width (8) <= DB column width	OK
varchar	ABAP data value width (8) > DB column width	*[rtrunc]*
	ABAP data value width (8) <= DB column width	OK
nchar	ABAP data value width (8) > DB column width	*[rtrunc]*
	ABAP data value width (8) <= DB column width	OK
date		**-1205**
other types	See ABAP data type C	see above

You cannot map the ABAP data type D to the Informix data type date because they have different formats. Format conversion is not possible because of the different display variants.

ABAP Data Type: T

DB Data type	Test case [*notes*]	Result
char	ABAP data value width (6) > DB column width	*[rtrunc]*
	ABAP data value width (6) <= DB column width	OK
varchar	ABAP data value width (6) > DB column width	*[rtrunc]*
	ABAP data value width (6) <= DB column width	OK
nchar	ABAP data value width (6) > DB column width	*[rtrunc]*
	ABAP data value width (6) <= DB column width	OK
datetime hour to second		**-1261**
other types	See ABAP data type C	see above

You cannot map the ABAP data type T to the Informix data type datetime because they have different formats. There is no reformatting.

ABAP Data Type: X

DB Data type	Test case [*notes*]	Result

char	ABAP data value width >= 256	**-609**
	ABAP data value width >= 256	OK
varchar	ABAP data value width >= 256	**-609**
	ABAP data value width >= 256	OK
nchar	ABAP data value width >= 256	**-609**
	ABAP data value width >= 256	OK
text	ABAP data value width >= 256	OK
byte	ABAP data value width >= 256	OK
date		*[undef]*
datetime		*[undef]*
interval		*[undef]*
decimal		*[undef]*
integer		*[undef]*
smallint		*[undef]*
float		*[undef]*
smallfloat		*[undef]*
double precision		*[undef]*

Since the DBSL for R/3 tables in Informix allows you to store hexadecimal fields shorter than 256 characters in CHAR format, the native SQL module allows you to access these fields. If the ABAP variable is shorter than 256 characters, the database field is interpreted as CHAR. The first two characters in the database field contain the length information.

This also means that the ABAP variable must be larger than 256 characters when you access a byte or text field.

SELECT

Reading database fields using Native SQL.

Each of the following tables refers to a **database field type**, which is also the type of the value you want to read from the database table. In some cases, several database data types behave in the same way. Where this occurs, the data types have been included in a single table to save space.

In the left-hand column is the **ABAP data type** of the target variable in the ABAP program.

On the right is the reaction of the ABAP Workbench.

Database Field Type char, varchar or nchar

ABAP Data type	Test case [*notes*]	Result
Character C	Database data value width > ABAP field width	*[rtrunc]*
	Database data value width = ABAP field width	OK
	Database data value width < ABAP field width [*left-justified, filled out with trailing blanks*]	OK
Numeric N	Database data value width > ABAP field width	*[rtrunc]*
	Database data value width = ABAP field width	OK
	Database data value width < ABAP field width [*right-justified, filled out with leading zeros*]	OK
Packed P	Database data value width > ABAP field width	*[rtrunc]*
	Database data value width <= ABAP field width	OK
Integer I	Database data value is non-numeric	**-1213**
	Loss of significant figures in conversion	**-1215**
	No loss in conversion	OK
Float F	Database data value is non-numeric	**-1213**
	Loss of significant figures in conversion	*[round]*
	Loss of non-significant figures in conversion	*[round]*
	No loss in conversion	OK
Date D	Database data value width > ABAP field width (8)	*[rtrunc]*
	Database data value width = ABAP field width (8)	OK
	Database data value width < ABAP field width (8) [*left-justified, filled out with trailing blanks*]	OK
Time T	Database data value width > ABAP field width (6)	*[rtrunc]*
	Database data value width = ABAP field width (6)	OK
	Database data value width < ABAP field width (6) [*left-justified, filled out with trailing blanks*]	OK
Hexadecimal X	Database data value width > 256 > ABAP field width	*[rtrunc]*
	ABAP data value width >= 256	**-1269**
	ABAP field width < 256 and database column width < 256 [*left-justified, 1st and 2nd characters lost*]	OK

Database Field Type text

ABAP Data type	Test case [*notes*]	Result
Character C	Database data value width > ABAP field width	*[rtrunc]*
	Database data value width < ABAP field width [*left-justified, filled out with trailing blanks*]	OK
	Database data value width = ABAP field width	OK
Numeric N		**-1269**
Packed P		**-1269**
Integer I		**-1269**
Float F		**-1269**
Date D		**-1269**
Time T		**-1269**
Hexadecimal X	ABAP data value width < 256	**-1269**
	ABAP data value width >= 256	OK

Database Field Type byte

ABAP Data type	Test case [*notes*]	Result
Character C	ABAP data value width < 256	**-1269**
	ABAP data value width >= 256 Database data value width > ABAP field width Database data value width < ABAP field width [*left-justified, filled out with trailing blanks*] Database data value width = ABAP field width	 [rtrunc] ok OK
Numeric N		**-1269**
Packed P		**-1269**
Integer I		**-1269**
Float F		**-1269**
Date D		**-1269**
Time T		**-1269**
Hexadecimal X	ABAP data value width < 256	**-1269**
	ABAP data value width >= 256	OK

Database Field Type date

ABAP Data type	Test case [*notes*]	Result
Character C	ABAP data value width < 10	*[rtrunc]*
	ABAP field width > 10 [*left-justified, filled out with trailing blanks*]	OK
	ABAP field width = 10	OK
Numeric N		*[undef]*
Packed P		*[undef]*
Integer I		*[undef]*
Float F		*[undef]*
Date D		*[undef]*
Time T		*[undef]*
Hexadecimal X		**-1269**

Database Field Type datetime or interval

ABAP Data type	Test case [*notes*]	Result
Character C	Database data value width > ABAP field width	*[rtrunc]*
	Database data value width < ABAP field width [*may not be left-justified, filled out with trailing blanks*]	OK
Numeric N		*[undef]*
Packed P		*[undef]*
Integer I		**-1260**
Float F		**-1260**
Date D		*[undef]*
Time T		*[undef]*
Hexadecimal X		**-1269**

Database Field Type decimal, numeric, or money

ABAP Data type	Test case [*notes*]	Result

Character C	Loss of figures in conversion	[rtrunc]
	No loss in conversion	OK
Numeric N	Loss of figures in conversion	[rtrunc]
	No loss in conversion	OK
Packed P	Loss of figures in conversion	[rtrunc]
	No loss in conversion	OK
Integer I	Loss of figures in conversion	**-1215**
	No loss in conversion	OK
Float F	Loss of figures in conversion	[round]
	No loss in conversion	OK
Date D		*[undef]*
Time T		*[undef]*
Hexadecimal X		*[undef]*

When you read the data type DECIMAL into an ABAP field with type C, you must allow space for thousand separators as well as for the decimal point.

Database Field Type integer or smallint

ABAP Data type	**Test case [*notes*]**	**Result**
Character C	Loss of significant figures in conversion	*[rtrunc]*
	No loss in conversion	OK
Numeric N	Loss of significant figures in conversion	[rtrunc]
	No loss in conversion	OK
Packed P	Loss of significant figures in conversion	[rtrunc]
	No loss in conversion	OK
Integer I	No loss in conversion	OK
Float F		OK
Date D		*[undef]*
Time T		*[undef]*
Hexadecimal X		*[undef]*

Database Field Type smallfloat, float, or double precision

ABAP Data type	Test case [*notes*]	Result
Character C	Loss of significant figures in conversion	[rtrunc]
	No loss in conversion	OK
Numeric N	Loss of significant figures in conversion	[rtrunc]
	No loss in conversion	OK
Packed P	Loss of significant figures in conversion	[rtrunc]
	No loss in conversion	OK
Integer I	Loss of significant figures in conversion	**-1215**
	No loss in conversion	OK
Float F		OK
Date D		*[undef]*
Time T		*[undef]*
Hexadecimal X		*[undef]*

Using Informix Native SQL in R/3 Release 4

You cannot use the following Native SQL statements in ABAP. They come under the following categories:

- NA
  not applicable. These statements are non-executable and generate an error.
- SP
  Special syntax. These statements require a special Native SQL syntax; see the keyword documentation for EXEC SQL
- NR
  Not recommended. These statements should not be used with an R/3 database that is managed using the ABAP Dictionary.

This list is not intended to be exhaustive. It is merely intended as a programming guideline. Any Informix SQL statements that are not listed here should be executable without any problem.

Informix SQL statement	Class
ALLOCATE DESCRIPTOR	NA
ALTER FRAGMENT	NR
ALTER INDEX	NR
ALTER TABLE … MODIFY NEXT SIZE	NR

ALTER TABLE … LOCK MODE	NR
ALTER TABLE … ADD ROWIDS	NR
ALTER TABLE … DROP ROWIDS	NR
BEGIN WORK	NA
CHECK TABLE	NR
CLOSE DATABASE	NR
CONNECT	SP
CREATE AUDIT	NR
CREATE DATABASE	NR
DATABASE	NR
DEALLOCATE DESCRIPTOR	NR
DECLARE CURSOR	SP
DELETE ... WHERE CURRENT OF <cursid>	NA
DESCRIBE	NA
DISCONNECT	SP
DROP AUDIT	NR
DROP DATABASE	NR
EXECUTE	NA
EXECUTE IMMEDIATE	NA
EXECUTE PROCEDURE	SP
FETCH	SP
FLUSH	NA
FREE	NA
GET DESCRIPTOR	NA
GET DIAGNOSTICS	NA
GRANT FRAGMENT	NR
INFO	NA
LOAD	NA
OPEN CURSOR	SP
OUTPUT	NA
PREPARE	NA
PUT	NA
RECOVER TABLE	NR

RENAME DATABASE	NR
REVOKE FRAGMENT	NR
ROLLFORWARD DATABASE	NR
SELECT ... INTO TEMP <tab>	NA
SELECT ... FOR UPDATE OF <column>	NA
SET CONNECTION	SP
SET DATASKIP	NR
SET DEBUG FILE	NR
SET DESCRIPTOR	NR
SET TRANSACTION	NR
START DATABASE	NR
UNLOAD	NA
UPDATE STATISTICS	NR
UPDATE ... WHERE CURRENT OF <cursid>	NA
WHENEVER	NA

Native SQL for DB2 Common Server

Below are the conversion tables for ABAP variables and data types in the **DB2 Common Server** database.

- You should, where possible, use the combinations of ABAP variable types and database data types in **bold type**.
- For illegal type combinations, the ABAP error procedure is given, along with the SQL error code of any resulting ABAP short dump.
- The behavior of any type combinations not listed here is undefined.

Note: The following documentation refers to the current database release DB2 UDB Version 5.0. SAP cannot accept any responsibility for consistency in other releases. Consistency will only be ensured if the database manufacturer (IBM) itself guarantees that the type conversions will be backwards-compatible. These tables are not to be regarded as a set of guaranteed conversion rules.

The following abbreviations are used:

- **ok**: The interface can perform the required conversion
- ***[info]*** : The conversion is performed. There are database warnings, but no SQL error. However, the data is truncated at the right [*rtrunc*], left [*trunc*], rounded [*round*], or undefined [*undef*].
- **<error code>:** The failure of the conversion is marked by the corresponding SQL error code from the database. This is included in the ABAP short dump.

Insert/Update

ABAP Data Type: Character C

DB Data type	Test case [*notes*]	Result
char	ABAP data value width > DB column width	**22001**
	ABAP data value width <= DB column width	OK
varchar	ABAP data value width > DB column width	**22001**
	ABAP data value width <= DB column width	OK
decimal	ABAP data value is non-numeric	**22005**
	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures	**22003**
	No loss in conversion	OK
numeric	ABAP data value is non-numeric	**22005**
	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures	**22003**
	No loss in conversion	OK
smallint	ABAP data value is non-numeric	**22005**
	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures	[*trunc*]
	No loss in conversion	OK
integer	ABAP data value is non-numeric	**22005**
	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures	[*trunc*]
	No loss in conversion	OK
float	ABAP data value is non-numeric	**22005**
	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures	[*trunc*]
	No loss in conversion	OK
double	ABAP data value is non-numeric	**22005**
	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures	[*trunc*]

	No loss in conversion	OK
date	ABAP data value does not have a valid DB date format	**22007**
	ABAP data value has a valid DB date format	OK
time	ABAP data value does not have a valid DB time format	**22007**
	ABAP data value has a valid DB time format	ok
timestamp	ABAP data value does not have a valid DB timestamp format	**22007**
	ABAP data value has a valid DB timestamp format	OK

ABAP Data Type: Numeric N

DB Data type	**Test case [*notes*]**	**Result**
char	ABAP data value width > DB column width	**22001**
	ABAP data value width <= DB column width	OK
varchar	ABAP data value width > DB column width	**22001**
	ABAP data value width <= DB column width	OK
decimal	Loss of figures in conversion	**22003**
	No loss in conversion	OK
numeric	Loss of figures in conversion	**22003**
	No loss in conversion	OK
smallint	Loss of figures in conversion	**22003**
	No loss in conversion	OK
integer	Loss of figures in conversion	**22003**
	No loss in conversion	OK
float	Loss of figures in conversion	**22003**
	No loss in conversion	OK
double	Loss of figures in conversion	**22003**
	No loss in conversion	OK
date		**22007**
time		**22007**
timestamp		**22007**

ABAP Data Type: Packed Number P

DB Data type	Test case [*notes*]	Result
decimal	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures	[*trunc*]
	No loss in conversion	OK
numeric	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures	[*trunc*]
	No loss in conversion	OK
smallint	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures	[*trunc*]
	No loss in conversion	OK
integer	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures	[*trunc*]
	No loss in conversion	OK
float	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures	[*trunc*]
	No loss in conversion	OK
double	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures	[*trunc*]
	No loss in conversion	OK

ABAP Data Type: Integer I

DB Data type	Test case [*notes*]	Result
char	Loss of figures in conversion	**22003**
	No loss in conversion	OK
varchar	Loss of figures in conversion	**22003**
	No loss in conversion	OK
decimal	Loss of figures in conversion	**22003**
	No loss in conversion	OK
numeric	Loss of figures in conversion	**22003**
	No loss in conversion	OK

smallint	Loss of figures in conversion	**22003**
	No loss in conversion	OK
integer	Loss of figures in conversion	**22003**
	No loss in conversion	OK
float	Loss of figures in conversion	**22003**
	No loss in conversion	OK
double	Loss of figures in conversion	**22003**
	No loss in conversion	OK
date		**22007**
time		**22007**
timestamp		**22007**

ABAP Data Type: Float F

DB Data type	Test case [*notes*]	Result
char	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures	**22003**
	No loss in conversion	OK
varchar	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures	**22003**
	No loss in conversion	OK
decimal	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures	[*trunc*]
	No loss in conversion	OK
numeric	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures	[*trunc*]
	No loss in conversion	OK
smallint	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures	[*trunc*]
	No loss in conversion	OK
integer	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures	[*trunc*]
	No loss in conversion	OK

float	Loss of significant figures in conversion	**22003**
	No loss in conversion	OK
double	Loss of significant figures in conversion	**22003**
	No loss in conversion	OK
date		**22007**
time		**22007**
timestamp		**22007**

ABAP Data Type: Date D

DB Data type	Test case [*notes*]	Result
char(8)	ABAP data value width (8) > DB column width	**22001**
	ABAP data value width (8) <= DB column width	OK
varchar	ABAP data value width (8) > DB column width	**22001**
	ABAP data value width (8) <= DB column width	OK
decimal	Loss of figures in conversion	**22003**
	No loss in conversion	OK
numeric	Loss of figures in conversion	**22003**
	No loss in conversion	OK
smallint	[Data is always lost]	**22003**
integer		OK
float		OK
double		OK
date		**22007**
time		**22007**
timestamp		**22007**

ABAP Data Type: Time T

DB Data type	Test case [*notes*]	Result
char(6)	ABAP data value width (6) > DB column width	**22001**
	ABAP data value width (6) <= DB column width	OK
varchar	ABAP data value width (6) > DB column width	**22001**

	ABAP data value width (6) <= DB column width	OK
decimal	Loss of figures in conversion	**22003**
	No loss in conversion	OK
numeric	Loss of figures in conversion	**22003**
	No loss in conversion	OK
smallint	[Data is always lost]	**22003**
integer		OK
float		OK
double		OK
date		**22007**
time		**22007**
timestamp		**22007**

ABAP Data Type: Hexadecimal X

DB Data type	**Test case [*notes*]**	**Result**
char	ABAP data value width > DB column width	**22001**
	ABAP data value width = DB column width	OK
	ABAP data value width < DB column width	[undef]
varchar	ABAP data value width > DB column width	**22001**
	ABAP data value width = DB column width	OK
	ABAP data value width < DB column width	[undef]

Select

Database Field Type char, varchar or longvarchar

ABAP Data type	**Test case [*notes*]**	**Result**
Character C	Database data value width > ABAP field width	[rtrunc]
	Database data value width < ABAP field width [*left-justified, filled out with trailing blanks*]	OK
	Database data value width = ABAP field width	OK
Numeric N	Database data value is non-numeric	[undef]

	Database data value width > ABAP field width [*simulated SQL error*]	**22sim**
	Database data value width < ABAP field width [*right-justified, filled out with leading zeros*]	OK
	Database data value width = ABAP field width	OK
Packed P	[*simulated SQL error*]	**22sim**
Integer I	Database data value is non-numeric	**22005**
	Loss of significant figures in conversion	**22003**
	No loss in conversion	OK
Float F	Database data value is non-numeric	**22005**
	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures in conversion	[round]
	No loss in conversion	OK
Date D	Behaves like ABAP type C(8)	-
Time T	Behaves like ABAP type C(6)	-
Hexa X	Database data value width > ABAP field width	[rtrunc]
	Database data value width < ABAP field width [*left-justified, remaining length undefined*]	OK
	Database data value width = ABAP field width	OK

Database Field Type decimal or numeric

ABAP Data type	**Test case [*notes*]**	**Result**
Character C	[*simulated SQL error*]	**22sim**
Numeric N	[*simulated SQL error*]	**22sim**
Packed P	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures in conversion	[round]
	No loss in conversion	OK
Integer I	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures in conversion	[round]
	No loss in conversion	OK
Float F	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures in conversion	[round]
	No loss in conversion	OK

Date D	[*simulated SQL error*]	**22sim**
Time T	[*simulated SQL error*]	**22sim**
Hexa X		[undef]

Database Field Type float or double

ABAP Data type	**Test case [*notes*]**	**Result**
Character C	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures in conversion	[round]
	No loss in conversion	OK
Numeric N	[*simulated SQL error*]	**22sim**
Packed P	[*simulated SQL error*]	**22sim**
Integer I	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures in conversion	[round]
	No loss in conversion	OK
Float F	Loss of significant figures in conversion	**22003**
	Loss of non-significant figures in conversion	[round]
	No loss in conversion	OK
Date D	[*simulated SQL error*]	**22sim**
Time T	[*simulated SQL error*]	**22sim**
Hexa X		[undef]

Database Field Type integer or smallint

ABAP Data type	**Test case [*notes*]**	**Result**
Character C	Loss of significant figures in conversion	**22003**
	No loss in conversion	OK
Numeric N	[*simulated SQL error*]	**22sim**
Packed P	[*simulated SQL error*]	**22sim**
Integer I	[No loss in conversion]	OK
Float F	[No loss in conversion]	OK
Date D	[*simulated SQL error*]	**22sim**
Time T	[*simulated SQL error*]	**22sim**

Hexa X		[undef]

Database Field Type date, time or timestamp

ABAP Data type	Test case [*notes*]	Result
Character C	Database data value width > ABAP field width	**22003**
	Database data value width < ABAP field width [*left-justified, filled out with trailing blanks*]	OK
	Database data value width = ABAP field width	OK
Numeric N	[*simulated SQL error*]	**22sim**
Packed P	[*simulated SQL error*]	**22sim**
Integer I	[*restricted data type attribute violation*]	**07006**
Float F	[*restricted data type attribute violation*]	**07006**
Date D	[*simulated SQL error*]	**22sim**
Time T	[*simulated SQL error*]	**22sim**
Hexa X	[*restricted data type attribute violation*]	**07006**

Logical Databases

Logical databases are special ABAP programs that retrieve data and make it available to application programs. The most common use of logical databases is still to read data from database tables by linking [Page 947] them to executable ABAP programs.

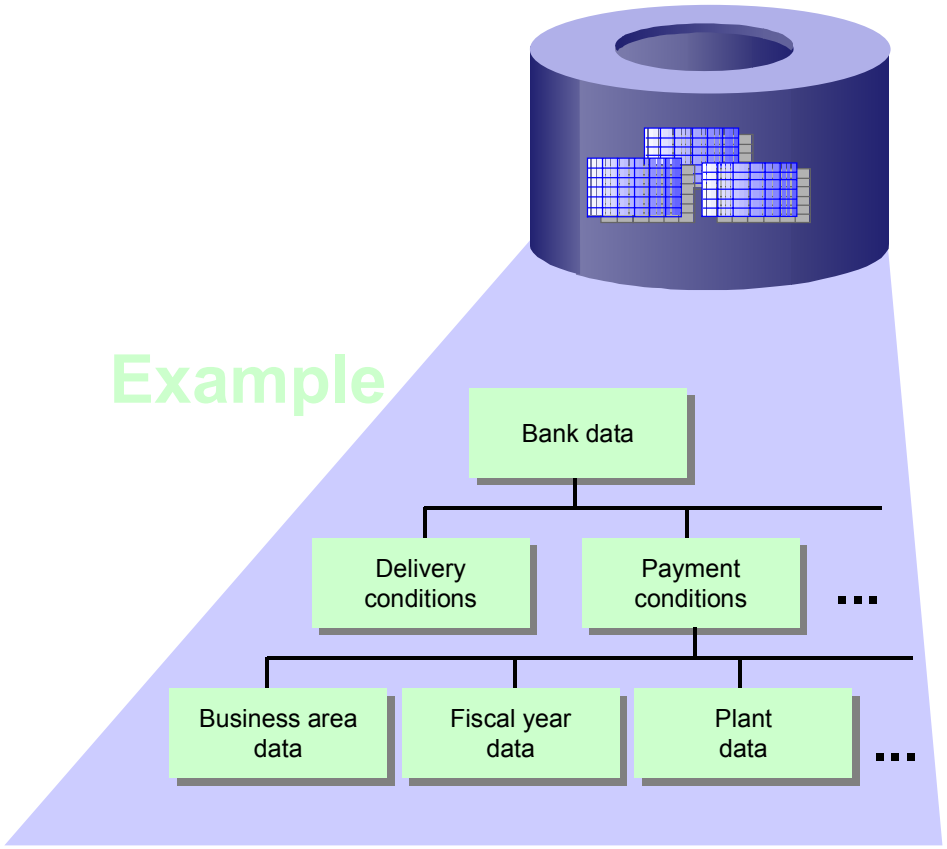
However, from Release 4.5A, it has also been possible to call logical databases using the function module LDB_PROCESS. This allows you to call several logical databases from any ABAP program, nested in any way. It is also possible to call a logical database more than once in a program, if it has been programmed to allow this. This is particularly useful for programs with type 1.

Logical databases contain Open SQL [Page 1041] statements that read data from the database. You do not therefore need to use SQL in your own programs. The logical database reads the program, stores them in the program if necessary, and then passes them line by line to the application program or the function module LDB_PROCESS using an interface work area [Page 130].

Logical Databases - Views of Data

A logical database provides a particular view of database tables in the R/3 System. It is always worth using logical databases if the structure of the data that you want to read corresponds to a view available through a logical database.

The data structure in a logical database is hierarchical. Many tables in the R/3 System are linked to each other using foreign key relationships. Some of these dependencies form tree-like hierarchical structures. Logical databases read data from database tables that are part of these structures.



The diagram illustrates how the R/3 System might represent the structure of a company. A logical database can read the lines of these tables one after the other into an executable program in a sequence which is normally defined by the hierarchical structure. The term logical database is sometimes used to mean not only the program itself, but also the data that it can procure.

Tasks of Logical Databases

As well as allowing you to read data from the database, logical databases also allow you to program other tasks centrally, making your application programs less complicated. They can be used for the following tasks:

- Reading the same data for several programs.

  The individual programs do not then need to know the exact structure of the relevant database tables (and especially not their foreign key relationships). Instead, they can rely on the logical database to read the database entries in the right order during the GET event.

- Defining the same user interface for several programs.

  Logical databases have a built-in selection screen. Therefore, all of the programs that use the logical database have the same user interface.

- Central authorization checks

  Authorization checks for central and sensitive data can be programmed centrally in the database to prevent them from being bypassed by simple application programs.

- Improving performance

  If you want to improve response times, logical databases permit you to take a number of measures to achieve this (for example, using joins instead of nested SELECT statements). These become immediately effective in all of the application programs concerned and save you from having to modify their source code.

Structure of Logical Databases [Page 1166]

Selection Views [Page 1173]

Example of a Logical Database [Page 1175]

Using Logical Databases [Page 1179]

Editing Logical Databases [Page 1191]

Structure of Logical Databases

A logical database is made up of three components (see illustration [Page 60] for further details). They are:

- Structure

  The structure defines the data view of the logical database. It determines the structure of the other components and the behavior of the logical database at runtime. The order in which data is made available to the user depends on the hierarchical structure of the logical database concerned.

- Selections

  The selections define a selection screen, which forms the user interface of the executable programs that use the logical database. Its layout is usually determined by the structure. You can adapt the selections to your own requirements and also add new ones. When you link a logical database to an executable program, the selections of the logical database become part of the standard selection screen of the program (screen number 1000). If you call a logical database using the function module LDB_PROCESS, the selections are filled using interface parameters.

- Database program

  The database program contains the ABAP statements used to read the data and pass it to the user of the logical database. The structure of the database program is a collection of special subroutines. It is determined by the structure and the selections. You can adapt the database program to your own requirements and also extend it.

Other components such as documentation, language-specific texts, and user-defined selection screens extend the functions further.

Structure

The structure of a logical database is usually based on the foreign key relationships between hierarchical tables in the R/3 System. Logical databases have a tree-like structure, which can be defined as follows:

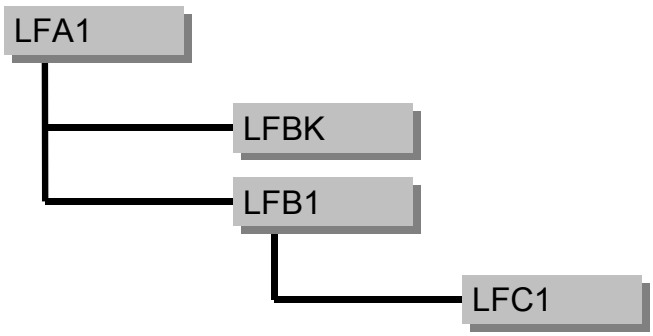
- There is a single node at the highest level. This is known as the root node.
- Each node can have one or several branches.
- Each node is derived from one other node.

The nodes must be structures defined in the ABAP Dictionary or data types from a type group. Normally, these are the structures of database tables which the logical database reads and passes to the user for further evaluation. However, it is also possible, and sometimes useful, to use ABAP Dictionary structures without an underlying database. For technical reasons, the maximum number of nodes allowed in the structure of a logical database is 300.

Any executable ABAP program that has a logical database linked to it can contain a GET [Page 958] statement for each node of the structure. When you run the program, the corresponding event blocks [Page 952] are processed in the sequence prescribed by the hierarchical structure of the logical database. If a program does not contain a GET statement for every node of a logical database, the processing passes through all the nodes that lie in the path from the root to the nodes specified by GET statements.

If you call a logical database using the function module LDB_PROCESS, the depth to which the system reads is controlled by an interface parameter.

Suppose LFA1 is the root node, LFBK and LFB1 are branches of LFA1, and LFC1 is a branch of LFB1.



If the executable program contains GET statements for all nodes, the GET events are executed in the order LFA1, LFBK, LFB1, LFC1. If the program only contains a single GET statement for LFB1, the processing only passes through LFA1 and LFB1.

Selections

The selections in a logical database are defined using the normal statements for defining selection screens [Page 686], that is, PARAMETERS, SELECT-OPTIONS; and SELECTION-SCREEN. In a logical database, you can also use the additions VALUE-REQUEST and HELP-REQUEST to define specific input and value help. You define the selection screen in a special include program known as the selection include.

When you write programs using a logical database, you can also add your own selections. The standard selection screen then contains the database-specific selections, followed by the program-specific selections that you have defined.

When the system generates the selection screen for an executable program, database-specific selection criteria and parameters are only displayed if you have declared an interface work area [Page 130] for them in your program using the NODES or TABLES statement.

Suppose you have a selection include containing the following lines:

```
SELECT-OPTIONS SLIFNR FOR LFA1-LIFNR.
PARAMETERS PBUKRS LIKE LFB1-BUKRS FOR TABLE LFB1.
```

The selection criterion SLIFNR is linked to table LFA1, the parameter PBUKRS to table LFB1. If the TABLES statement in an executable program (report) declares LFA1 but not LFB1, SLIFNR is displayed on the selection screen, but PBUKRS does not appear.

The selection screen of a logical database can contain **dynamic selections** as well as static ones. Dynamic selections are extra, user-defined selections that the user can make as well as using the static selections defined in the selection include. To improve performance, you should

always use this option instead of reading more data than you need and then sorting it out in the application program.

To make dynamic selections available for the node <node> of a logical database, the selection include must contain the following statement:

```
SELECTION-SCREEN DYNAMIC SELECTIONS FOR NODE|TABLE <node>.
```

If the node <node> is requested by the user of the logical databases, the dynamic selections are included in the selection screen. A user can then choose *Dynamic selections* to enter extra selections for the corresponding fields. If you call the logical database using the function module LDB_PROCESS, you can pass a corresponding parameter. You can use these selections in dynamic statements in the logical database program [Page 1201] to read data. The values of the program-specific selection criteria [Page 711] that you defined for a node for which dynamic selections were available are also passed to the logical database. The user can also define the fields for dynamic selections as a selection view [Page 1173] for the logical database.

The selection screen of a logical database is part of the standard selection screen (number 1000) of the executable program to which the logical database is attached. It has a standardized layout - the selection criteria and parameters appear on separate lines in the order in which they were declared. You can change the layout using the SELECTION-SCREEN statement.

The runtime environment generates the selection screen with number 1000 for every program in which the attributes do not contain a different selection screen version. You can prevent certain input fields from the selection screen of a logical database from appearing on the selection screen by defining selection screen versions with a screen number lower than 1000 in the selection include, and entering this version number in the program attributes. By pressing `F4` there, you can get an overview of the selection screen versions defined in the logical database concerned. To define a selection screen version, use the statements SELECTION-SCREEN BEGIN|END OF VERSION. Within these statements you can use SELECTION-SCREEN EXCLUDE to specify fields that you do not want to appear on the selection screen.

If the attributes of an executable program contain the number of a selection screen version, the version is used in the standard selection screen. Although the input fields that you excluded from the selection screen are not displayed, the corresponding selections still exist, and you can still edit them in the program or by calling the function module LDB_PROCESS.

Database Program

The name of the database program of a logical database <ldb> conforms to the naming convention SAPDB<ldb>. It serves as a container for subroutines, which the ABAP runtime environment calls when a logical database is processed. The sequence of the calls and their interaction with the events in executable programs [Page 952] or the function module LDB_PROCESS depends on the structure of the logical database.

A logical database program usually contains the following subroutines:

- FORM LDB_PROCESS_INIT

  Called once only before the logical database is processed. It prepares it to be called more than once by the function module LDB_PROCESS.

- FORM INIT

  Called once only before the selection screen is processed.

- FORM PBO

Called before the selection screen is displayed, each time it is displayed. Consequently, it is only called when you use the logical database with an executable program, not with the function module LDB_PROCESS.

- FORM PAI

  Called when the user interacts with the selection screen. Consequently, it is only called when you use the logical database with an executable program, not with the function module LDB_PROCESS. The interface parameters FNAME and MARK are passed to the subroutine.

  FNAME contains the name of a selection criterion or parameter on the selection screen.

  MARK describes the selection made by the user: MARK = SPACE means that the user has entered a simple single value or range selection. MARK = '*' means that the user has also made entries on the *Multiple Selection* screen.

- FORM LDB_PROCESS_CHECK_SELECTIONS

  Called instead of the subroutine PAI if the logical database is called using the function module LDB_PROCESS without a selection screen. This subroutine can check the selections passed in the function module interface.

- FORM PUT_<node>

  Called in the sequence defined in the structure. Reads the data from the node <node> and uses the

  PUT <node>.

  statement to trigger a corresponding GET event in the ABAP runtime environment. The PUT statement is the central statement in this subroutine: It can only be used within a subroutine of a logical database. The logical database must contain the node <node>, and the subroutine name must begin with PUT_<node>. The PUT statement directs the program flow according to the structure of the logical database. The depth to which the logical database is read is determined by the GET statements in the application program or the interface parameter CALLBACK of the function module LDB_PROCESS.

  First, the subroutine PUT_<root> is executed for the root node. The PUT statement then directs the program flow as follows:

  i) If the database program contains the subroutine AUTHORITY_CHECK_<table>, the first thing the PUT_<node> statement does is to call it.

  ii) Next, the PUT statement triggers a GET event in the runtime environment. If there is a corresponding GET <node> statement in the executable program to which the logical database is linked, the associated event block is processed. If the CALLBACK parameter of the function module LDB_PROCESS is filled accordingly, the corresponding callback routine is called.

  iii) The PUT statement then directs the program flow as follows:

  (a) To the next subroutine of a node that follows directly, if a lower-level node (not necessarily the very next) in the same subtree is requested by GET in the executable program or in the function module.

  (b) To the subroutine of a node at the same level, if the preceding node branches to such a node and if a GET statement exists for such a node in the executable program or the function module.

The PUT statement in that subroutine starts again at step (i). In the subroutine of the lowest node in a subtree to be processed using GET, the program control does not branch further. Instead, the current subroutine is processed further. When a subroutine PUT_<node> has been executed in its entirety, the program flow returns to the PUT statement from which it branched to the subroutine PUT_<node>.

iv) When control has returned from a lower-level subroutine PUT_<node>, the PUT statement triggers the event GET <node> LATE in the runtime environment.

- FORM AUTHORITY_CHECK_<node>

  Called automatically by the PUT <table> statement. In this subroutine, you can specify authorization checks for the appropriate node <table> from the structure of the logical database.

- FORM PUT_<ldb>_SP

  Called when the user makes a selection using a search help to process the key chosen in the search help. <ldba> is the name of the logical database. From this subroutine, you can use the entries in the search help tables to read the relevant entries from the root node <root>. The processing in the program can then be triggered using PUT <root>. The subroutine PUT_<root> is then not called automatically.

- FORM BEFORE_EVENT

  Called before an event, the name of which is passed in the parameter EVENT. Currently, the EVENT field can only contain the value START-OF-SELECTION, to call a subroutine before this event.

- FORM AFTER_EVENT

  Called after an event, the name of which is passed in the parameter EVENT. Currently, the EVENT field can only contain the value END-OF-SELECTION, to call a subroutine after this event.

- FORM <par>_VAL, <selop>_VAL, <selop>-LOW_VAL, <selop>-HIGH_VAL

  Called when the user calls possible values help for the parameter <par> or the selection criterion <selop>. These must belong to the selections in the logical database.

- FORM <par>_HLP, <selop>_HLP, <selop>-LOW_HLP, <selop>-HIGH_HLP

  Called when the user calls input help for the parameter <par> or the selection criterion <selop>. These must belong to the selections in the logical database.

Example

Suppose that in the logical database structure, LFB1 is a branch of LFA1,

and that the following selection criteria are defined in the selection include:

```
SELECT-OPTIONS: SLIFNR FOR LFA1-LIFNR,
                SBUKRS FOR LFB1-BUKRS.
```

A section of the database program would then read:

```
FORM PUT_LFA1.
  SELECT * FROM LFA1
```

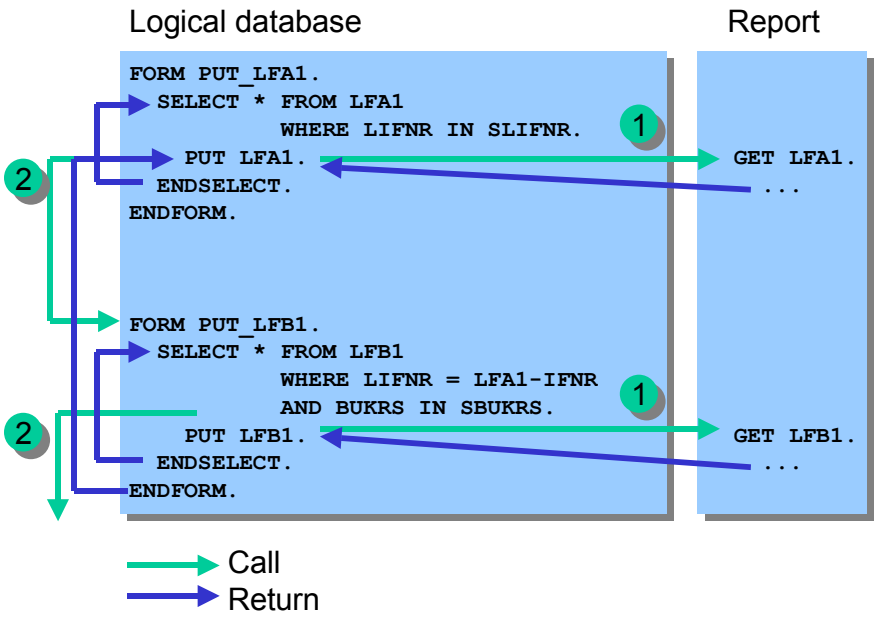
```
        WHERE LIFNR IN SLIFNR.
   PUT LFA1.
  ENDSELECT.
ENDFORM.

FORM PUT_LFB1.
  SELECT * FROM LFB1
        WHERE LIFNR =  LFA1-LIFNR.
        AND   BUKRS IN SBUKRS.
   PUT LFB1.
  ENDSELECT.
ENDFORM.
```

An executable program (report) linked to the logical database could contain the lines:

```
GET LFA1.
  WRITE LFA1-LIFNR.

GET LFB1.
  WRITE LFB1-BUKRS.
```

In this example, the runtime environment calls the routine PUT_LFA1 after the event START-OF-SELECTION. The event GET LFA1 is triggered by the statement PUT LFA1. Once the corresponding event block in the program is complete, PUT LFA1 branches to the subroutine PUT_LFB1. From this subroutine, the event GET LFB1 is triggered in the application program. If LFB1 is the last node to be read, processing resumes with the SELECT loop in PUT_LFB1. Otherwise, the program flow moves to the subroutine PUT_<node> of the next node. At the end of the SELECT loop of the last node, processing resumes in the SELECT loop of the node at the next level up. The example of programming using nested SELECT loops is only used to make the program flow clearer. In a real logical database, you would avoid doing this in order to minimize the number of database accesses [Page 1108].

The following diagram shows the program flow:



In this example, the PUT statements do not branch to the authorization check subroutines.

Selection Views

Selection views are a collection of fields from different database tables. You can define extra selections for fields in a selection view at runtime using dynamic selections. The extra selections are sent to the database and help to minimize the number of database accesses. This works as long as the database tables in the selection include of the logical database are defined for dynamic selections, and that the database program is programmed accordingly.

You can define a selection view from the ABAP Workbench by choosing *Extras* → *Selection views* to open the Logical Database Builder, or by choosing *Further objects* from the initial screen of the Repository Browser. They are identified using a three-character key describing the origin of the selection view. Predefined selection views have the key SAP, customer-defined ones have the key CUS. This way, users can define the best logical database selection views for their requirements.

If you want to use a selection view in a logical database, it must be called STANDARD, and must be assigned to a logical database. Selection views that are not assigned to a particular logical database may have any name, and they can be used freely.

The following rules apply to the structure of dynamic selections on the selection screens of logical databases: You can only use selection views with the name STANDARD and which are assigned to the logical database. If there is a customer-defined selection view (key CUS), it is used. Otherwise, the predefined selection view (key SAP) is used. If there is no selection view with the name STANDARD that is assigned to the logical database, you can construct dynamic selections using any field of any table in the logical database.

Editing Selection Views

When you create or change a selection view in the Logical Database Builder, you need to fill in the following input fields:

Functional Groups

Within a selection view, fields are grouped into functional groups. All of the fields assigned to a functional group are contained in the view. Functional groups allow you to search more efficiently when you use a selection view. You define a functional group by assigning an ID and a descriptive text to it. The functional group ID can consist of any two characters. It is only relevant when you edit the selection view.

Table/Node Name

This field contains the names of the database tables from which you can choose fields for the selection view. In the bottom part of the screen, the system displays the fields of one of the tables. To choose the table, use the *Choose* function or double-click the table or node name.

Fields in the Table or Node

In this field, enter the ID of the functional group in which you want to adopt the field. The fields is then assigned to this functional area and included in the selection view. For a field assigned to a functional group to be able to be used in a logical database, the corresponding node in the logical database must also have been defined for dynamic selections.

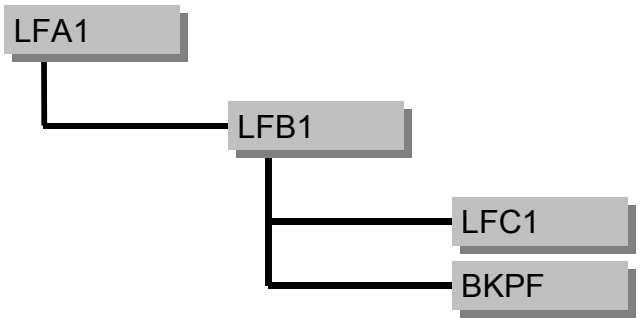
If a field of a table is assigned to a functional group, a checkbox appears to the right of the field description, in which you can mark the field for preselection. These fields then appear as selection criteria on the selection screen for dynamic selections. You can change the preselected

fields by choosing *New field selection* on the selection screen. A selection list appears, containing all of the fields in the selection view for which dynamic selections are supported in the corresponding node of the logical database.

Example of a Logical Database

Let us consider the logical database TEST_LDB.

Structure



Selections in the Selection Include

```
SELECT-OPTIONS: SLIFNR   FOR LFA1-LIFNR,
                SBUKRS   FOR LFB1-BUKRS,
                SGJAHR   FOR LFC1-GJAHR,
                SBELNR   FOR BKPF-BELNR.
```

Database Program

```
*---------------------------------------------------------*
* DATABASE PROGRAM OF THE LOGICAL DATABASE TEST_LDB
*---------------------------------------------------------*
PROGRAM SAPDBTEST_LDB DEFINING DATABASE TEST_LDB.
TABLES: LFA1,
        LFB1,
        LFC1,
        BKPF.

*---------------------------------------------------------*
* Initialize selection screen (process before PBO)
*---------------------------------------------------------*
FORM INIT.
 ....
ENDFORM.                              "INIT

*---------------------------------------------------------*
* PBO of selection screen (always before selection
* screen
*---------------------------------------------------------*
FORM PBO.
 ....
ENDFORM.                              "PBO
```

Example of a Logical Database

```
*---------------------------------------------------------*
* PAI of selection screen (process always after ENTER)
*---------------------------------------------------------*
FORM PAI USING FNAME MARK.
  CASE FNAME.
    WHEN 'SLIFNR'.
     ....
    WHEN 'SBUKRS'.
     ....
    WHEN 'SGJAHR'.
     ....
    WHEN 'SBELNR'.
     ....
  ENDCASE.
ENDFORM.                              "PAI
*---------------------------------------------------------*
* Call event GET LFA1
*---------------------------------------------------------*
FORM PUT_LFA1.
  SELECT * FROM LFA1
         WHERE LIFNR      IN SLIFNR.
    PUT LFA1.
  ENDSELECT.
ENDFORM.                              "PUT_LFA1
*---------------------------------------------------------*
* Call event GET LFB1
*---------------------------------------------------------*
FORM PUT_LFB1.
  SELECT * FROM LFB1
         WHERE LIFNR      =  LFA1-LIFNR
           AND BUKRS      IN SBULRS.
    PUT LFB1.
  ENDSELECT.
ENDFORM.                              "PUT_LFB1
*---------------------------------------------------------*
* Call event GET LFC1
*---------------------------------------------------------*
FORM PUT_LFC1.
  SELECT * FROM LFC1
         WHERE LIFNR      =  LFA1-LIFNR
           AND BUKRS      =  LFB1-BUKRS
           AND GJAHR      IN SGJAHR.
    PUT LFC1.
  ENDSELECT.
ENDFORM.                              "PUT_LFC1
*---------------------------------------------------------*
* Call event GET BKPF
*---------------------------------------------------------*
FORM PUT_BKPF.
  SELECT * FROM BKPF
         WHERE BUKRS      =  LFB1-BUKRS
```

```
        AND BELNR     IN SBELNR
        AND GJAHR     IN SGJAHR.
  PUT BKPF.
 ENDSELECT.
ENDFORM.                    "PUT_BKPF
```

The PROGRAM statement has the addition DEFINING DATABASE TEST_LDB. This defines the database program as belonging to the logical database TEST_LDB.

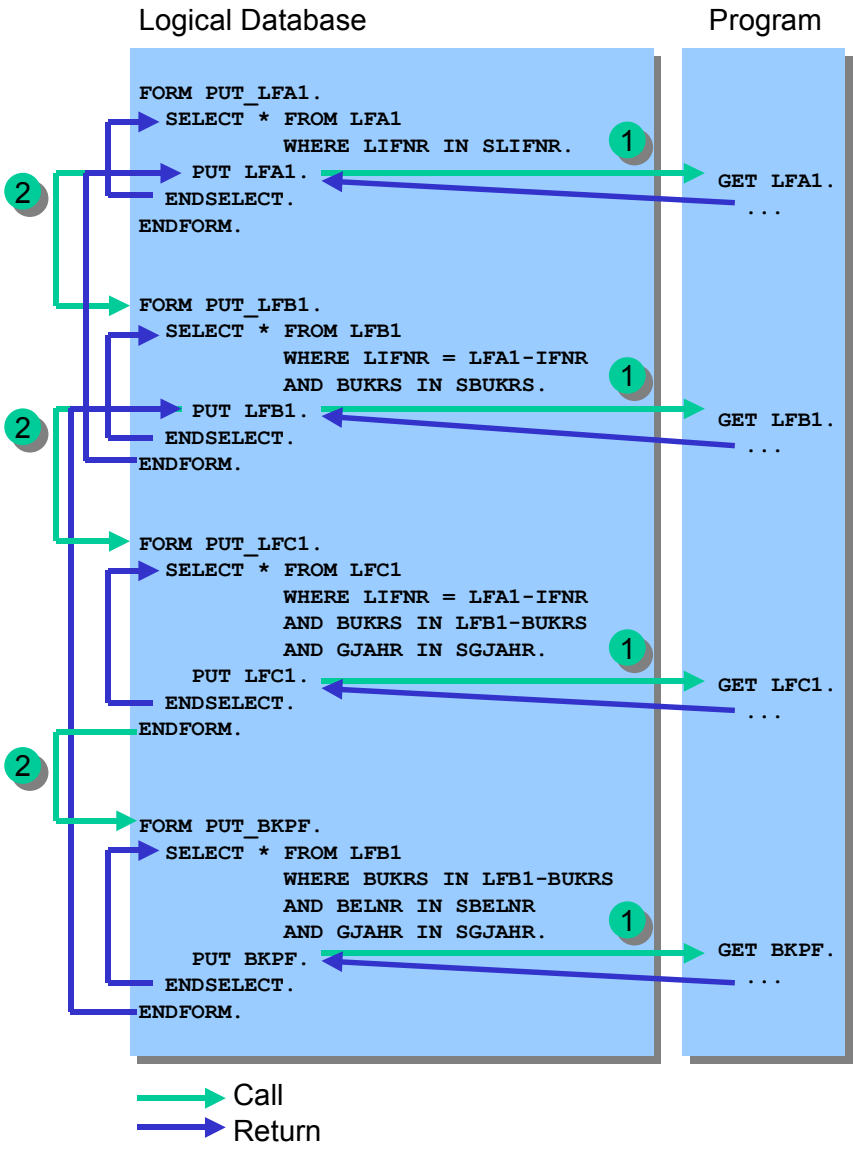
The nodes of the structure are declared with the TABLES statement which generates the appropriate table work areas. You can also use the NODES statement to define database tables as nodes. If a node of a logical database is not a database table, you must use the NODES statement. The interface work areas are **shared** by the database program and the user, and so act as an interface for passing data. The term “user” here can mean either an executable program to which the logical database is linked, or the function module LDB_PROCESS.

The subroutines INIT and PBO initialize the selection screen.

In the PAI subroutine, you can include an authorization check for the user input on the selection screen. Plausibility or value range checks are also possible. If a check fails, you can write an error dialog. The corresponding field on the selection screen is then made ready for input again.

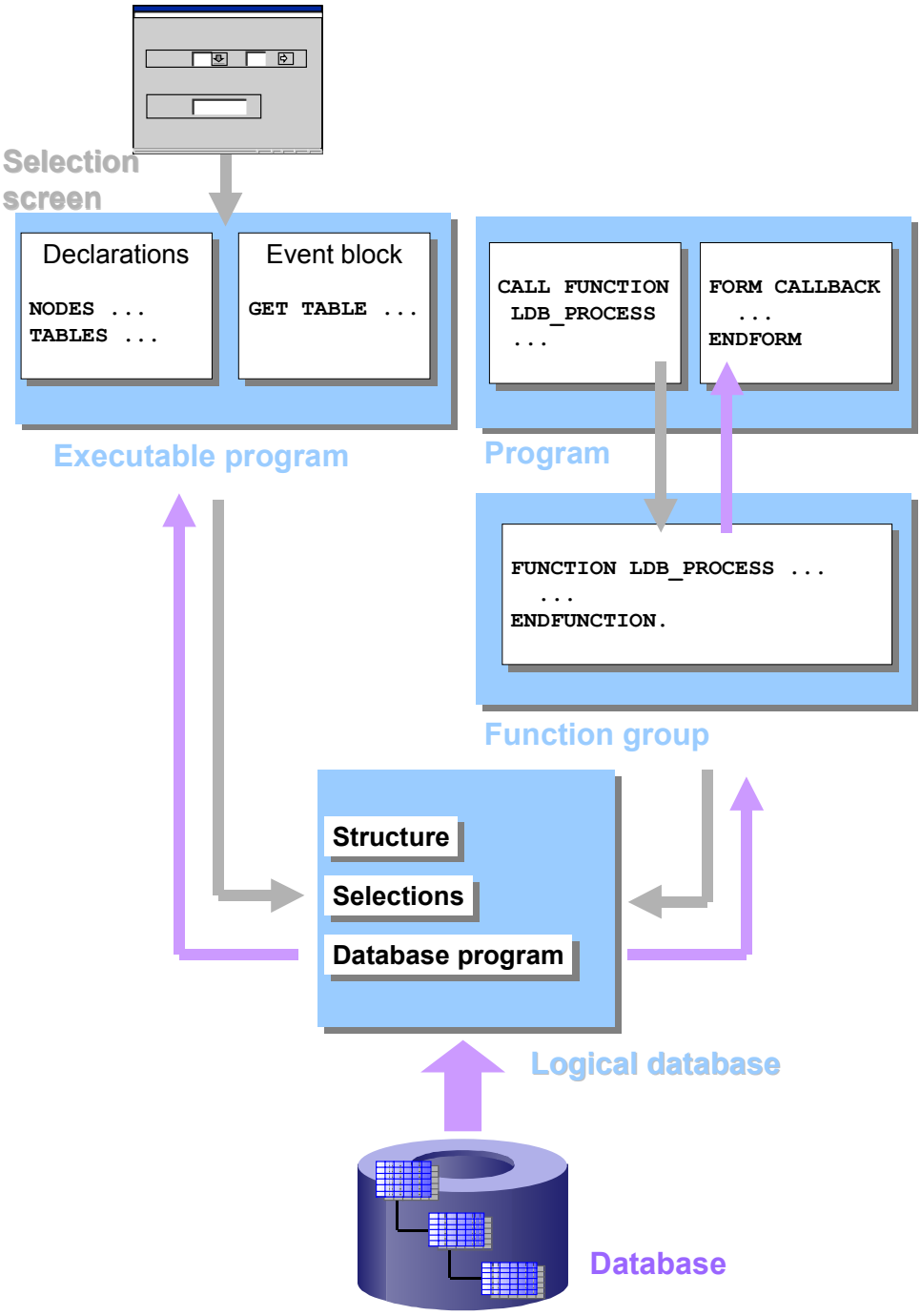
The PUT_<node> subroutines read the database tables according to the selection criteria entered by the user and trigger the relevant events in the executable program. This program is intended only to show the essential structure of a logical database. It does not contain any refinements to improve response times. The order in which the subroutines are called is determined by the structure of the logical database.

Example of a Logical Database



Using Logical Databases

There are two ways of using a logical database: Either by linking it with an executable program, or by using the function module LDB_PROCESS in any ABAP program.



Using Logical Databases

When you link a logical database to an executable program, the user can enter values on the selection screen, and the data read by the logical database is passed back to the program using the interface work areas. If you call the logical database using a function module, the selection screen is not displayed. The calling program does not have to provide interface work areas. Instead, it uses special subroutines called callback routines, which are called by the function module and filled with the required data.

Linking a Logical Database to an Executable Program [Page 1181]

Calling a Logical Database Using a Function Module [Page 1185]

Linking a Logical DB to an Executable Program

When you link an executable program to a logical database by entering the name of the logical database in the program attributes, the subroutines of the logical database program and the event blocks of the executable program form a modularized program for reading and processing data. The individual processing blocks are called in a predefined sequence by the runtime environment (see the diagram in the section Logical Databases and Contexts [Page 60]). The runtime sequence is controlled by the structure, selections, and PUT statements in the logical database, and by the GET statements in the executable program.

Selection Screen

If you specify a logical database in the attributes of an executable program, this affects the standard selection screen of the program. It contains both the selection fields from the logical database and those from the program itself. You can specify which of the logical database selections are relevant for your program, and should therefore appear on the screen, by declaring interface work areas for the relevant nodes.

Runtime Behavior

The following list shows the sequence in which the ABAP runtime environment calls the subroutines of the logical database and the event blocks in the executable program [Page 952]. The runtime environment executes a series of processors (selection screen processor, reporting processor). The ABAP code listed below shows the processing blocks that belong to the individual steps.

1. Initialization before the selection screen is processed.

   Subroutine:

   FORM INIT

   This subroutine is called once only before the selection screen is first displayed.

   Event block:

   INITIALIZATION.

   This event occurs once only before the selection screen is first displayed.

2. PBO of the Selection screen Initialization before each occasion on which the selection screen is displayed (for example, to supply default values for key fields).

   Subroutine:

   FORM PBO.

   This subroutine is called each time the selection screen is sent (before it is displayed).

   Event block:

   AT SELECTION-SCREEN OUTPUT.

   This event is called each time the selection screen is sent (before it is displayed).

3. The selection screen is displayed at the presentation server, and the user can enter data in the input fields.
4. Input help (F4) or field help (F1) requests.

Subroutines:

FORM <par>_VAL.
FORM <selop>_VAL.
FORM <selop>-LOW_VAL.
FORM <selop>-HIGH_VAL.

If the user requests a list of possible entries for database-specific parameters <par> or selection criteria <selop>, these subroutines are called as required.

If the user requests field help for these parameters, the subroutines are called with the ending _HLP instead of _VAL.

Event blocks:

AT SELECTION-SCREEN ON VALUE-REQUEST FOR <par>.
AT SELECTION-SCREEN ON VALUE-REQUEST FOR <selop>-LOW.
AT SELECTION-SCREEN ON VALUE-REQUEST FOR <selop>-HIGH.

If the user requests a list of possible entries for database-specific parameters <par> or selection criteria <selop>, these events are triggered as required.

If the user requests field help for these parameters, the events with the addition ON HELP-REQUEST occurs instead of ON VALUE-REQUEST.

5. PAI of the selection screen. Checks to see whether the user has entered correct, complete, and plausible data. Also contains authorization checks. If an error occurs, you can program a user dialog and make the relevant fields ready for input again.

   Subroutines:

   FORM PAI USING FNAME MARK.

   The interface parameters FNAME and MARK are passed by the runtime environment.

   FNAME contains the name of a selection criterion or parameter on the selection screen.

   If MARK = SPACE, the user has entered a simple single value or range selection.

   If MARK = '*', the user has also entered selections on the *Multiple Selection* screen.

   Using the combination FNAME = '*' and MARK = 'ANY', you can check all entries at once when the user chooses a function or presses ENTER.

   Event blocks:

   AT SELECTION-SCREEN ON <fname>.

   Event for processing a particular input field.

   AT SELECTION-SCREEN ON END OF <fname>.

   Event for processing multiple selections.

   AT SELECTION-SCREEN.

   Event for processing all user input.

6. Processing before reading data.

   Subroutine:

   BEFORE EVENT 'START-OF-SELECTION'.

The logical database can use this subroutine for necessary actions before reading data, for example, initializing internal tables.

Event block:

START-OF-SELECTION.

First event in an executable program after the selection screen has been processed. You can use this event block to prepare the program for processing data.

7. Reading data in the logical database and processing in the executable program.

   Subroutine:

   FORM PUT_<node>

   The logical database reads the selected data of the node <node>.

   Event block:

   GET <table> [LATE].

   This event is triggered by the PUT statement in the above subroutine. This event block allows you to process the data read for <node> in the corresponding interface work area.

8. Processing after reading data.

   Subroutine:

   AFTER EVENT 'END-OF-SELECTION'.

   The logical database can use this subroutine for necessary actions after reading data, for example, releasing memory space.

   Event block:

   END-OF-SELECTION.

   Last reporting event. You can use this event block to process the temporary dataset that you have created (for example, sort it).

9. If a list was generated during the above steps, the list processor in the runtime environment takes control of the program and displays the list.

Suppose TABLE1 is the root node and TABLE2 is its only subordinate node in a logical database. The processing steps for reading and processing data would then have the following hierarchical order:

```
START-OF-SELECTION.
  FORM PUT_TABLE1.
  GET TABLE1.
    FORM PUT_TABLE2.
    GET TABLE2.
  GET TABLE1 LATE.
END-OF-SELECTION.
```

Authorization Checks in Logical Databases

It makes sense to use authorization checks using the AUTHORITY-CHECK [Page 502] statement in the following subroutines in the database program or event blocks of the executable program:

- Subroutines in the database program:
  - PAI
  - AUTHORITY_CHECK_<table>
- Event blocks in the executable program:
  - AT SELECTION-SCREEN
  - AT SELECTION-SCREEN ON <fname>
  - AT SELECTION-SCREEN ON END OF <fname>
  - GET <table>

Whether you place the authorization checks in the database program or in the executable program depends on the following:

- The structure of the logical database.

  For example, you should only check authorizations for company code if you actually read lines containing the company code at runtime.

- Performance

  Avoid repetitive checks (for example, within a SELECT loop).

The separation of database access and application logic allows you to program **all** of your authorization checks centrally in the logical database program. This makes it easier to maintain large programming systems.

Calling a Logical Database Using a Function Module

From Release 4.5A it is possible to call logical databases independently from any ABAP program. Previously it was only possible to link a logical database to an executable program, in which the processing blocks of the logical database and the program were controlled by the ABAP runtime environment.

To call a logical database from another program, use the function module **LDB_PROCESS**. This allows you to use the logical database as a routine for reading data. You can call more than one logical database from the same program. You may also call the same logical database more than once from the same program. In the past, it was only possible to use a logical database more than once or use more than one logical database by calling a further executable program using SUBMIT [Page 1018]. These programs had to be linked to the corresponding logical database, and the data had to be passed to the calling program using ABAP memory [Page 1032] or a similar technique.

When you call a logical database using the function module LDB_PROCESS, its selection screen is not displayed. Instead, you fill the selections using the interface parameters of the function module. The logical database does not trigger any GET events in the calling program, but passes the data back to the caller in callback routines. Calling a logical database using LDB_PROCESS thus decouples the actual data retrieval from the preceding selection screen processing and the subsequent data processing.

There is **no need** to adapt a logical database for use with LDB_PROCESS, except in the following cases: If you do not adapt a logical database, it is not possible to use the function module to call the same logical database more than once. The PAI subroutine is not called when you use LDB_PROCESS. This means that none of the checks for selections programmed in it are performed. You can work around these restrictions by including the subroutines LDB_PROCESS_INIT and LDB_PROCESS_CHECK_SELECTIONS [Page 1219] in the database program.

Runtime Behavior

The subroutines in the logical database are called in the following sequence when you call the function module LDB_PROCESS:

1. LDB_PROCESS_INIT
2. INIT
3. LDB_PROCESS_CHECK_SELECTIONS
4. PUT <node>.

None of the subroutines used to process the selection screen when you link the logical database to an executable program [Page 1181] are called, neither does the runtime environment trigger any reporting events in the calling program. Instead, the PUT statements in the logical database trigger actions in the function module that call callback routines in the calling program. In other words, the function module catches the events that are otherwise processed by the runtime environment.

Parameters of LDB_PROCESS

The function module has the following import parameters:

- LDBNAME

Name of the logical database you want to call.

- VARIANT

  Name of a variant to fill the selection screen of the logical database. The variant must already be assigned to the database program of the logical database. The data is passed in the same way as when you use the WITH SELECTION-TABLE addition in a SUBMIT [Page 1019] statement.

- EXPRESSIONS

  In this parameter, you can pass extra selections for the nodes of the logical database for which dynamic selections are allowed. The data type of the parameter RSDS_TEXPR is defined in the type group RSDS. The data is passed in the same way as when you use the WITH FREE SELECTION addition in a SUBMIT [Page 1019] statement.

- FIELD_SELECTION

  You can use this parameter to pass a list of the required fields for the nodes of the logical database for which dynamic selections are allowed. The data type of the parameter is the deep internal table RSFS_FIELDS, defined in the type group RSFS. The component TABLENAME contains the name of the node and the deep component FIELDS contains the names of the fields that you want to read.

The function module has the following tables parameters:

- CALLBACK

  You use this parameter to assign callback routines to the names of nodes and events. The parameter determines the nodes of the logical database for which data is read, and when the data is passed back to the program and in which callback routine.

- SELECTIONS

  You can use this parameter to pass input values for the fields of the selection screen of the logical database. The data type of the parameter corresponds to the structure RSPARAMS in the ABAP Dictionary. The data is passed in the same way as when you use the WITH SELECTION-TABLE addition in a SUBMIT [Page 1019] statement.

If you pass selections using more than one of the interface parameters, values passed in SELECTIONS and EXPRESSIONS overwrite values for the same field in VARIANT.

Read Depth and Callback Routines

When you link a logical database with an executable program, the GET statements determine the depth to which the logical database is read. When you call the function module LDB_PROCESS, you determine the depth by specifying a node name in the CALLBACK parameter. For each node for which you request data, a callback routine can be executed at two points. These correspond to the GET and GET LATE events in executable programs. In the table parameter CALLBACK, you specify the name of the callback routine and the required execution point for each node. A callback routine is a subroutine in the calling program or another program that is to be executed at the required point.

For the GET event, the callback routine is executed directly after the data has been read for the node, and before the subordinate nodes are processed. For the GET_LATE event, the callback routine is processed after the subordinate nodes have been processed.

The line type of the table parameter CALLBACK is the flat structure LDBCB from the ABAP Dictionary. It has the following components:

- LDBNODE

  Name of the node of the logical database to be read.

- GET

  A flag (contents X or SPACE), to call the corresponding callback routine at the GET event.

- GET_LATE

  A flag (contents X or SPACE), to call the corresponding callback routine at the GET LATE event.

- CB_PROG

  Name of the ABAP program in which the callback routine is defined.

- CB_FORM

  Name of the callback routine.

If you pass an internal table to the CALLBACK parameter, you must fill at least one of the GET or GET_LATE columns with X for each node (you may also fill both with X).

A callback routine is a subroutine that must be defined with the following parameter interface:

```
FORM <subr> USING <node> LIKE LDBCB-LDBNODE
                 <wa>   [TYPE <t>]
                 <evt>
                 <check>.
```

The parameters are filled by the function module LDB_PROCESS. They have the following meaning:

- <node> contains the name of the node.
- <wa> is the work area of the data read for the node. The program that calls the function module LDB_PROCESS and the program containing the callback routine do not have to declare interface work areas using NODES or TABLES. If the callback routine is only used for one node, you can use a TYPE reference to refer to the data type of the node in the ABAP Dictionary. Only then can you address the individual components of structured nodes directly in the subroutine. If you use the callback routine for more than one node, you cannot use a TYPE reference. In this case, you would have to address the components of structured nodes by assigning them one by one [Page 213] to a field symbol.
- <evt> contains G or L, for GET or GET LATE respectively. This means that the subroutine can direct the program flow using the contents of <evt>.
- <check> allows the callback routine to influence how the program is processed further (but only if <evt> contains the value G). The value X is assigned to the parameter when the subroutine is called. If it has the value SPACE when the subroutine ends, this flags that the subordinate nodes of the logical database should not be processed in the function module LDB_PROCESS. This is the same as leaving a GET event block using CHECK [Page 974] in an executable program. If this prevents unnecessary data from being read, it will improve the performance of your program.

Exceptions of LDB_PROCESS

- LDB_ALREADY_RUNNING

  A logical database may not be called if it is still processing a previous call. If this occurs, the exception LDB_ALREADY_RUNNING is triggered.

- LDB_NOT_REENTRANT

  A logical database may only be called repeatedly if its database program contains the subroutine LDB_PROCESS_INIT [Page 1219], otherwise, this exception is triggered.

- LDB_SELECTIONS_NOT_ACCEPTED

  Error handling in the subroutine LDB_PROCESS_CHECK_SELECTIONS [Page 1219] of the database program can trigger this exception. The error message is placed in the usual system fields SY-MSG... [Page 483].

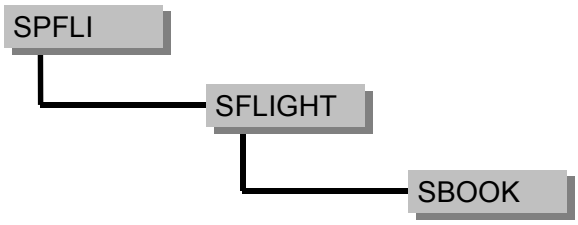
For details of further exceptions, refer to the function module documentation in the Function Builder.

Example

```
TABLES SPFLI.
SELECT-OPTIONS S_CARR FOR SPFLI-CARRID.

TYPE-POOLS: RSDS, RSFS.

DATA: CALLBACK TYPE TABLE OF LDBCB,
      CALLBACK_WA LIKE LINE OF CALLBACK.

DATA: SELTAB TYPE TABLE OF RSPARAMS,
      SELTAB_WA LIKE LINE OF SELTAB.

DATA: TEXPR TYPE RSDS_TEXPR,
      FSEL  TYPE RSFS_FIELDS.

CALLBACK_WA-LDBNODE     = 'SPFLI'.
CALLBACK_WA-GET         = 'X'.
CALLBACK_WA-GET_LATE    = 'X'.
CALLBACK_WA-CB_PROG     = SY-REPID.
CALLBACK_WA-CB_FORM     = 'CALLBACK_SPFLI'.
APPEND CALLBACK_WA TO CALLBACK.

CLEAR CALLBACK_WA.
CALLBACK_WA-LDBNODE     = 'SFLIGHT'.
CALLBACK_WA-GET         = 'X'.
CALLBACK_WA-CB_PROG     = SY-REPID.
CALLBACK_WA-CB_FORM     = 'CALLBACK_SFLIGHT'.
APPEND CALLBACK_WA TO CALLBACK.

SELTAB_WA-KIND = 'S'.
SELTAB_WA-SELNAME = 'CARRID'.

LOOP AT S_CARR.
  MOVE-CORRESPONDING S_CARR TO SELTAB_WA.
  APPEND SELTAB_WA TO SELTAB.
ENDLOOP.
```

```
CALL FUNCTION 'LDB_PROCESS'
     EXPORTING
          LDBNAME                       = 'F1S'
          VARIANT                       = ' '
          EXPRESSIONS                   = TEXPR
          FIELD_SELECTION               = FSEL
     TABLES
          CALLBACK                      = CALLBACK
          SELECTIONS                    = SELTAB
     EXCEPTIONS
          LDB_NOT_REENTRANT             = 1
          LDB_INCORRECT                 = 2
          LDB_ALREADY_RUNNING           = 3
          LDB_ERROR                     = 4
          LDB_SELECTIONS_ERROR          = 5
          LDB_SELECTIONS_NOT_ACCEPTED = 6
          VARIANT_NOT_EXISTENT          = 7
          VARIANT_OBSOLETE              = 8
          VARIANT_ERROR                 = 9
          FREE_SELECTIONS_ERROR         = 10
          CALLBACK_NO_EVENT             = 11
          CALLBACK_NODE_DUPLICATE       = 12
          OTHERS                        = 13.

IF SY-SUBRC <> 0.
  WRITE: 'Exception with SY-SUBRC', SY-SUBRC.
ENDIF.

FORM CALLBACK_SPFLI USING NAME  TYPE LDBN-LDBNODE
                          WA    TYPE SPFLI
                          EVT   TYPE C
                          CHECK TYPE C.
  CASE EVT.
   WHEN 'G'.
      WRITE: / WA-CARRID, WA-CONNID, WA-CITYFROM, WA-CITYTO.
      ULINE.
    WHEN 'L'.
      ULINE.
  ENDCASE.
ENDFORM.

FORM CALLBACK_SFLIGHT USING NAME  TYPE LDBN-LDBNODE
                            WA    TYPE SFLIGHT
                            EVT   TYPE C
                            CHECK TYPE C.
  WRITE: / WA-FLDATE, WA-SEATSOCC, WA-SEATSMAX.
ENDFORM.
```

The program is written to read data using the logical database F1S. The structure of F1S is:



A program-specific selection screen is defined at the beginning of the program. This requires the TABLES statement. Next, the required variables are defined for the interface.

The internal table CALLBACK is filled so that various callback routines are called in the program for the two nodes SPFLI and SFLIGHT. For SPFLI, the routine is to be called for GET and GET_LATE, for SFLIGHT, only at the GET event.

The internal table SELTAB is filled with values for the node SPFLI from the selection table S_CARR from the program-specific selection screen.

The program then calls the function module LDB_PROCESS with these parameters.

The subroutines CALLBACK_SPFLI and CALLBACK_SFLIGHT serve as callback routines. The interface parameter WA is fully typed, so you can address the individual components of the work areas. The events GET and GET LATE are handled differently in CALLBACK_SPFLI.

The beginning of the list output might look like this:

```
Test of LDB_PROCESS
-------------------------------------------------------------------------------
AA  0017 NEW YORK                SAN FRANCISCO
-------------------------------------------------------------------------------
1998/08/28          16          660
1998/09/30          10          660
1998/11/19          18          660
1998/11/22          77          660
1998/11/29          33          660
1998/12/19           2          660
1998/12/21           8          660
-------------------------------------------------------------------------------
AA  0064 SAN FRANCISCO           NEW YORK
-------------------------------------------------------------------------------
1998/09/07         168          280
1998/10/10          17          280
1998/11/29          23          280
1998/12/02          18          280
1998/12/09           2          280
1998/12/29          59          280
1998/12/31          24          280
-------------------------------------------------------------------------------
AZ  0555 ROME                    FRANKFURT
-------------------------------------------------------------------------------
1998/08/28          54          220
1998/09/30           2          220
1998/11/19         122          220
1998/11/22          48          220
1998/11/29          18          220
1998/12/19         187          220
1998/12/21          71          220
```

Editing Logical Databases

You edit logical databases using the Logical Database Builder in the ABAP Workbench. To start the Logical Database Builder, use Transaction SE36 or SLDB, or choose *Tools* → *ABAP Workbench*, followed by *Development* → *Programming environment* → *Logical Database Builder*. You can also start it by forward navigation from the Repository Browser or other ABAP Workbench tools.

In the *Logical database* field, you can enter a name of up to 20 characters. The name may also contain a three to ten-character namespace prefix, enclosed in forward slashes.

On the initial screen, you can create, copy, and delete logical databases. However, you can only delete a logical database if it is not linked to an executable program. Even if a logical database is not linked to any executable programs, it can still be used by a program that calls it using the function module LDB_PROCESS. Before deleting one, you should therefore check that it is not still in use. However, the *Where-used list* function only shows where a logical database is assigned to an executable program.

You can select the individual components of the logical database directly. When you are editing a logical database, two arrow icons appear in the application toolbar that allow you to navigate backwards and forwards between the most important components. You can also use the normal *Goto* function to navigate between components.

Creating a Logical Database

To create a new logical database, you should follow the procedure below. The Logical Database Builder then saves you work by using components that you have already defined to generate proposals for other components. Some of the most important attributes of a logical database are set when you define its structure. When you have defined the structure, the Logical Database Builder automatically generates a proposal for the selection include. The system then generates an input mask for the database program, based on the structure of the logical database and the selections.

These generated proposals allow you to create a working logical database quickly. However, you must program refinements such as authorization checks and performance optimization yourself.

Procedure for Creating a Logical Database

1. Enter a name on the initial screen of the Logical Database Builder and choose *Create*.
2. A dialog box appears. Enter a short text. You can change this later by choosing *Extras* → *Short text* or *Administration info.*
3. Once you have entered the short text, you must define the root node of the logical database. Enter the node name and its attributes. There are three different types of nodes:
   - **Database tables**. The table must be active in the ABAP Dictionary. Tables always have a flat structure. The name of the node must correspond with the name of the table.
   - **Data types from the ABAP Dictionary**: The node may refer to any data type in the ABAP Dictionary [Page 105]. The node name and the name of the data type do not have to be the same. You can use deep data types as nodes.
   - **Data types from type groups**: The node can also refer to a data type from a type group in the ABAP Dictionary. Enter the name of the type group in the corresponding field. You must choose *Other types* before specifying this type. Data types in type groups were the forerunners of real data types in the ABAP Dictionary. Wherever possible, you should use ABAP Dictionary data types. They have all of the semantic properties of their underlying data elements. This is useful, for example, when you use a logical database to create ABAP Queries.

   You can use the *Text from Dictionary* function to adopt the text stored in the ABAP Dictionary for the relevant table or data type.
1. The structure editor of the Logical Database Builder appears. On the left is the name of the root node, followed by a code for the node type: **T** for a database table, **S** for a ABAP Dictionary type, and **C** for a type from a type group. The new logical database now has a structure with a single node.
2. You can now extend the structure as described in Editing the Structure [Page 1194].
3. If you choose *Next screen* (right arrow in the application toolbar), a screen appears on which you can enter a search help for the logical database as described under Editing Search Helps [Page 1196].
4. If you choose *Next screen* (right arrow in the application toolbar), a dialog box appears, asking you whether the system should generate the selections for the logical database.

When you have confirmed the dialog box, a list appears, on which you can select all of the nodes that you want to use for field selections or dynamic selections. The fields that you select are included in the source code generated by the system for the selection include.

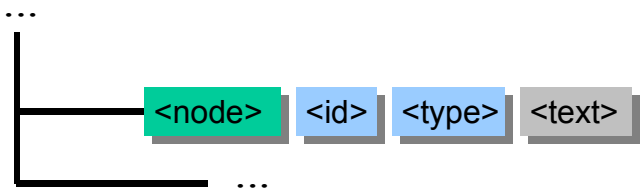
5. The generated selection include is displayed in the ABAP Editor. You can change it as described in Editing Selections [Page 1197].
6. If you choose *Next screen* (right arrow in the application toolbar), a dialog box appears, asking you whether the system should generate the database program for the logical database. The database program is generated from the structure and the selection include. It has a modular structure, consisting of several include programs and all of the necessary subroutines, with proposals for the statements that will read the data.
7. The generated database program is displayed in the ABAP Editor. You can change it as described in Editing the Database Program [Page 1201].
8. If you repeatedly choose *Previous screen* (left arrow in the application toolbar), you can display and change the general attributes of the logical database.
9. Finally, you can maintain optional selection texts and documentation.

Processing the Structure

To display or change the structure of a logical database, choose *Structure* from the initial screen of the Logical Database Builder, or navigate to the structure editor from another component.

Displaying the Structure

The structure editor displays the structure hierarchy. Each node is displayed as follows:



- <node> is the name of the node.
- <id> is the type of the node: **T** for a database table, **S** for a ABAP Dictionary type, and **C** for a type from a type group.
- <type> is the name of the ABAP Dictionary object to which the node refers.
- <text> is the short text for the node.

As usual, you can expand or collapse the subordinate tree structures, and edit individual subtrees. You can display the attributes of a node by double-clicking it. To display the structure of a node, that is, the components of its data type, place the cursor on the node and choose *Display table fields*.

The *PUT routine* function allows you to display the subroutine PUT_<node>. For this to work, the subroutine must be contained in its own include that follows a particular naming convention. For further information, refer to Editing the Database Program [Page 1201].

Changing the Structure

- To rename an existing node, place the cursor on it and choose *Edit* → *Node* → *Change*.
- To create a new node at a subordinate level to the cursor position, or on the same level, choose *Edit* → *Node* → *Create*. Nodes that you create like this in the structure editor consists at first only of its logical name, with which it is declared in the database program using the TABLES or NODES statement. To define the node fully, double-click it. You can then enter its type, the name of the data type from the ABAP Dictionary to which it refers, and its short text.
- To select/deselect a sub-tree, choose *Edit* → *Sub-tree* → *Select/deselect*. To move a selected sub-tree in a structure to a position indicated by the cursor, choose *Edit* → *Sub-tree* → *Reassign*.
- To delete a sub-tree, place the cursor on the node or select it and choose *Edit* → *Sub-tree* → *Delete*.

Editing a Search Help

A search help is a ABAP Dictionary object used to define possible values (F4) help. There are two kinds of search help - elementary and collective. An elementary search help uses a search path to determine the possible entries. A collective search help consists of two or more elementary search helps, and thus provides more than one possible search path.

To display or change the link between a logical database and a search help, choose *Search helps* from the initial screen of the Logical Database Builder or use the navigation function from another component.

Here, you can assign a search help to the logical database by choosing from a list. You can also delete the link between the logical database and an existing search help.

In deciding which search help is appropriate for the logical database, you must consider its content. For example, if you create a logical database that reads creditor records, the creditor number should be one of the output fields of the search help. The contents of the output fields of the search help are available to the logical database at runtime for the actual database access. There are two ways of finding information about the output fields of a search help. You can either use the ABAP Dictionary, or enter a search help and then look at the ensuing list.

To enable the user to use the search help, you must declare a special parameter in the selection include using the addition AS SEARCH PATTERN. The system interprets the user's input on the selection screen and reads the value list from the database. The values are made available to the database program in the internal table <ldb>_SP, and the subroutine PUT_<ldb>_SP is called instead of PUT_<root>. <ldb> is the name of the logical database, and <root> is the name of the root node. This subroutine can use the value list in <ldb>_SP to read the actual data and trigger the GET <root> event using the PUT_<root> statement.

Editing Selections

To display or change the structure of a logical database, choose *Selections* from the initial screen of the Logical Database Builder, or navigate to the selection include from another component.

The name of the selection include is DB<ldb>SEL, where <ldb> is the name of the logical database. You **must not** incorporate this include program in the database program using an INCLUDE statement. Instead, the runtime environment includes the selection include in the database program and the corresponding programs when it generates the logical database.

If you try to edit the selections but no selection include yet exists, the system generates one. This includes SELECT-OPTIONS statements for all of the database tables in the structure (nodes with type T). For each database table, the system proposes selection criteria [Page 703] for all of the fields in its primary key.

The SELECT-OPTIONS statements are commented out, and contain question marks instead of the names of the selection criteria. For each selection criterion that you want to use, you must enter the name and delete the comment character (*).

If a **search help** is specified for the logical database, the system also generates a corresponding PARAMETERS statement with the addition AS SEARCH PATTERN. A group box entitled *Selection using search help* then appears on the selection screen with input fields for the search help ID and the search string. There is also a pushbutton for complex search helps. If you choose this function, you can enter multiple selections for each field.

Finally, SELECTION-SCREEN statements are generated for **dynamic selections** and **field selections** for nodes with type T or S (if this is allowed by the structure definition).

As well as the default elements, you can use the following statements to extend the selection screen:

- Use the PARAMETERS statement and its additions to add input fields for single values [Page 689]. You could use these, for example, to control the flow of the program. In the selection include, you **must** use the addition FOR NODE or FOR TABLE in the PARAMETERS statement. You can use NODE for all node types. You can only use TABLE for nodes with type T. When the selection screen is generated, the system only generates fields for the nodes declared in the executable program in the NODES or TABLES statement, or those requested by the function module LDB_PROCESS.
- Use the SELECTION-SCREEN statement to format the selection screen [Page 718].

- The statement

  SELECTION-SCREEN DYNAMIC SELECTIONS FOR NODE|TABLE <node>.

  allows you to define further nodes for dynamic selections. If the node has type T, you can use TABLE instead of NODE. The user can then decide at runtime the components of the node for which he or she wants to enter selections. Dynamic selections require special handling in the database program.

- The statement

  SELECTION-SCREEN FIELD SELECTION FOR NODE|TABLE <node>.

  allows you to define further nodes for field selections. If the node has type T, you can use TABLE instead of NODE. In an executable program, you can use a field list in the GET statement to specify which fields of the node of the logical database should be read. In the function module LDB_PROCESS, the parameter FIELD_SELECTION must be

passed accordingly. Dynamic selections require special handling in the database program.

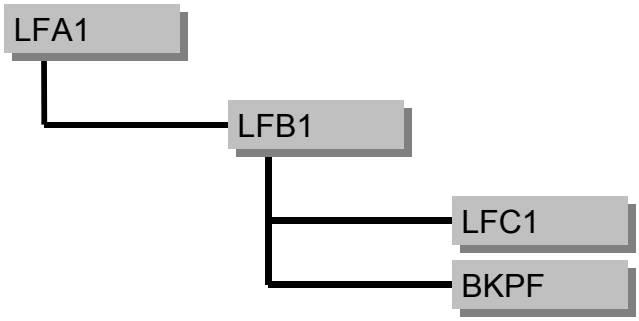
- The statements

  SELECTION-SCREEN BEGIN OF VERSION <dynnr>
  ...
  SELECTION-SCREEN EXCLUDE <f>.
  ...
  SELECTION-SCREEN BEGIN OF VERSION <dynnr>.

  allow you to create different versions of the selection screen with screen number <dynnr> less than 1000. You can hide the input fields of selection criteria or parameters as specified in <f>. This allows an executable program to work with an appropriate selection screen version.

To check the selection include DB<dba>SEL for syntax errors, choose *Check* on the initial screen. The system also checks the syntax of the selection include if you choose *Check* while editing the database program.

Suppose the logical database TEST_LDB has the following **structure:**



The generated default selection include would look like this:

```
*---------------------------------------------------------------*
* Include DBTEST_LDBSEL
* It will be automatically included into the database program
*---------------------------------------------------------------*
* If the source is automatically generated,
* please perform the following steps:
* 1. Replace ? by suitable names (at most 8 characters).
* 2. Activate SELECT-OPTIONS and PARAMETERS (delete stars).
* 3. Save source code.
* 4. Edit database program.
*
* Hint: Syntax-Check is not possible within this Include!
* It will be checked during syntax-check of database program.
*---------------------------------------------------------------*

* SELECT-OPTIONS :  ?  FOR LFA1-LIFNR.
```

```
* Parameter for search pattern selection (Type SY-LDB_SP):
* PARAMETERS p_sp AS SEARCH PATTERN FOR TABLE LFA1.

* SELECT-OPTIONS :
*             ?  FOR LFB1-LIFNR,
*             ?  FOR LFB1-BUKRS.

* SELECT-OPTIONS :
*             ?  FOR LFC1-LIFNR,
*             ?  FOR LFC1-BUKRS,
*             ?  FOR LFC1-GJAHR.

* SELECT-OPTIONS :
*             ?  FOR BKPF-BUKRS,
*             ?  FOR BKPF-BELNR,
*             ?  FOR BKPF-GJAHR.

* Enable DYNAMIC SELECTIONS for selected nodes :
* Enable FIELD SELECTION for selected nodes :
```

If the nodes LFA1 and LFB1 are defined for dynamic selections, and node LFC1 is defined for field selections, the following lines of code will also be generated:

```
SELECTION-SCREEN DYNAMIC SELECTIONS FOR TABLE LFA1.

SELECTION-SCREEN DYNAMIC SELECTIONS FOR TABLE LFB1.

SELECTION-SCREEN FIELD SELECTION FOR TABLE LFC1.
```

The automatically-created selection include could be modified as follows:

```
* Selection criteria:

  SELECT-OPTIONS SLIFNR  FOR LFA1-LIFNR.

  SELECT-OPTIONS SBUKRS  FOR LFB1-BUKRS.

  SELECT-OPTIONS SGJAHR  FOR LFC1-GJAHR.

  SELECT-OPTIONS SBELNR  FOR BKPF-BELNR.

* Self-defined parameters:

  PARAMETERS PSTIDA LIKE SY-DATUM FOR NODE BKPF.

* Dynamic selections for LFA1 and LFB1:

  SELECTION-SCREEN DYNAMIC SELECTIONS FOR NODEE: LFA1, LFB1.

* Field selection for LFB1 and LFC1:

  SELECTION-SCREEN FIELD SELECTION FOR NODE: LFB1, LFC1.
```

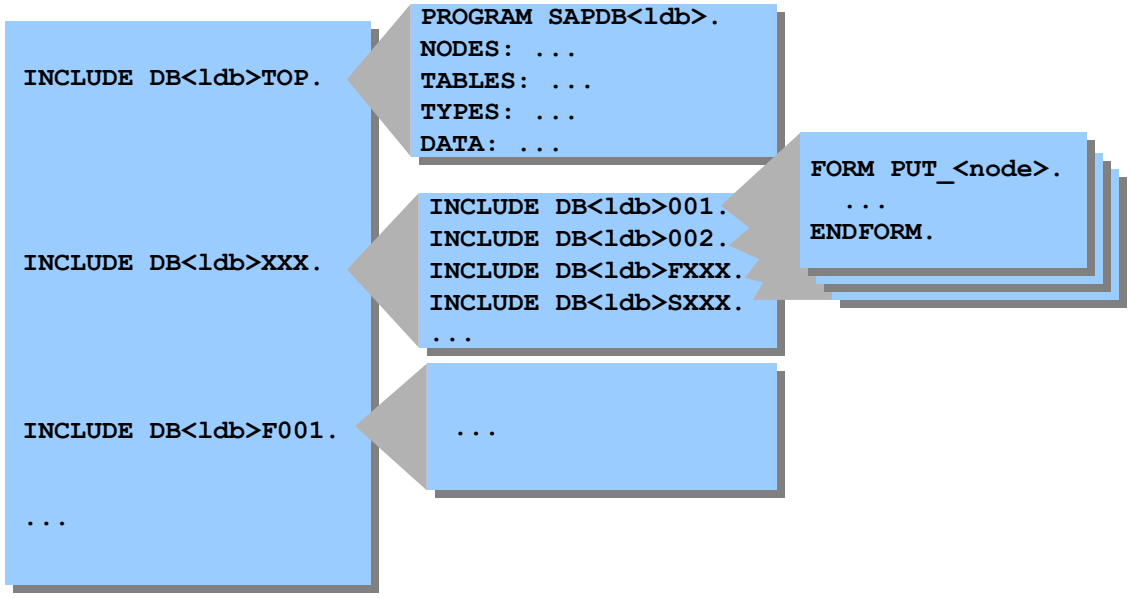
Here, selections are chosen from the available selection criteria and are given names. An additional parameter PSTIDA is declared and linked to the node BKPF. Dynamic selections are defined for the tables LFA1 and LFB1. Field selections are defined for tables LFB1 and LFC1.

Editing the Database Program

To display or change the database program of a logical database, choose *Database program* from the initial screen of the Logical Database Builder, or navigate to the database program from another component. The name of the program is SAPDB<ldb>, where <ldb> is the name of the logical database.

Structure of the Database Program

When you open the database program for editing for the very first time, the system generates it. The generated database program looks like a function group, since it consists of a series of generated include programs [Page 447]. This makes their structure easy to follow, even if the logical database has a large number of nodes. Older logical databases may not use includes and consist instead only of a main program.



Main program SAPDB<ldb> Include programs

The main program SAPDB<ldb> usually contains the following include programs:

- DB<ldb>TOP

  Contains global declarations.

- DB<ldb>XXX.

  Contains further include programs for individual subroutines.

- DB<ldb>F<001>, DB<ldb>F<002> ...

  User-defined include programs for extra functions.

The include program DB<ldb>XXX usually contains the following include programs:

Editing the Database Program

- DB<ldb>001, DB<ldb>002, …

  Contain the subroutines PUT_<node> and AUTHORITY_CHECK_<node> for the individual nodes of the logical database.

- DB<ldb>FXXX

  Contains most of the remaining subroutines for initialization, PBO, and PAI of the selection screen, and so on.

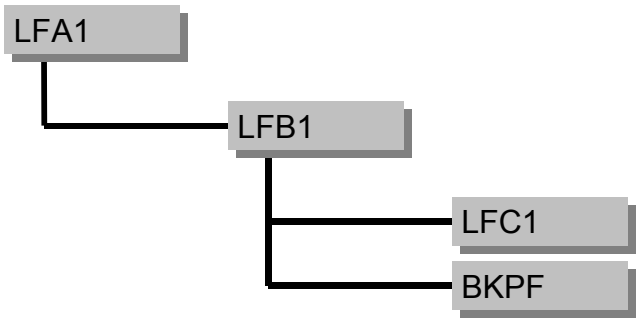
- DB<ldb>SXXX

  Contains the subroutine FORM PUT_<ldb>_SP for processing the search help.

You **must not** change the NODES and TABLES statements, or the prescribed names of the automatically-generated include programs and subroutines. However, you can define extra includes and subroutines, and change the ABAP statements used to read data. User-defined include programs must follow the naming convention DB<ldb>F<nnn> to ensure that they are transported with the logical database.

Full Example of a Generated ABAP Program

Suppose the logical database TEST_LDB has the following structure**:**



All of the nodes are database tables. Suppose the following selections are defined in the selection include:

SELECT-OPTIONS: SLIFNR FOR LFA1-LIFNR.

SELECT-OPTIONS: SBUKRS FOR LFB1-BUKRS.

SELECT-OPTIONS: SGJAHR FOR LFC1-GJAHR.

SELECT-OPTIONS: SBELNR FOR BKPF-BELNR.

The essential lines of the automatically-generated database program and its include programs are listed below. The program also contains some user and performance hints as comment lines, but these are not included here.

Main Program SAPDBTEST_LDB

```
*---------------------------------------------------------------------*
*       DATABASE PROGRAM OF LOGICAL DATABASE
```

```
TEST_LDB
*---------------------------------------------------------------*

INCLUDE DBTEST_LDBTOP .     " header
INCLUDE DBTEST_LDBXXX .     " all system routines
* INCLUDE DBTEST_LDBF001 . " user defined include
```

Include Program DBTEST_LDBTOP

```
PROGRAM SAPDBTEST_LDB DEFINING DATABASE TEST_LDB.

TABLES : LFA1,
         LFB1,
         LFC!,
         BKPF.

 * data: ...          "user defined variables
```

Include Program DBTEST_LDBXXX

```
*---------------------------------------------------------------*
*      Automatically generated file
* contains all necessary system routines of the database
program
*      !!!!! DO NOT CHANGE MANUALLY !!!!!
*---------------------------------------------------------------*

INCLUDE DBTEST_LDBN001 . " Node LFA1
INCLUDE DBTEST_LDBN002 . " Node LFB1
INCLUDE DBTEST_LDBN003 . " Node LFC1
INCLUDE DBTEST_LDBN004 . " Node BKPF
INCLUDE DBTEST_LDBFXXX . " INIT, PBO, PAI
INCLUDE DBTEST_LDBSXXX . " search help
```

Include Program DBTEST_LDB001

```
*---------------------------------------------------------------*
* Call event GET LFA1
*---------------------------------------------------------------*

FORM PUT_LFA1.

* SELECT * FROM LFA1
*   INTO LFA1
*   INTO TABLE ? (choose one!)
*   WHERE LIFNR = ?.

    PUT LFA1.

* ENDSELECT.

ENDFORM.                              "PUT_LFA1

*---------------------------------------------------------------*
* Authority Check for node LFA1
*---------------------------------------------------------------*

* FORM AUTHORITY_CHECK_LFA1.
*   AUTHORITY-CHECK ...
* ENDFORM.                            "AUTHORITY_CHECK_LFA1
```

Include Programs DBTEST_LDB002 and DBTEST_LDB003

Similar to DBTEST_LDB001, but for the nodes LFB1 and LFC1.

Include Program DBTEST_LDB004

```
*---------------------------------------------------------------------*
* Call event GET BKPF
*---------------------------------------------------------------------*

FORM PUT_BKPF.

* STATICS FLAG.
* IF FLAG = SPACE.
*   FLAG = 'X'.
*** Declarations for field selection for node BKPF ***
*   STATICS BKPF_FIELDS TYPE RSFS_TAB_FIELDS.
*   MOVE 'BKPF' TO BKPF_FIELDS-TABLENAME.
*   READ TABLE SELECT_FIELDS WITH KEY BKPF_FIELDS-TABLENAME
*     INTO BKPF_FIELDS.
* ENDIF.

* SELECT (BKPF_FIELDS-FIELDS) INTO CORRESPONDING FIELDS OF
*     BKPF / TABLE ? " (choose one of them)
*     FROM BKPF
*   WHERE BUKRS = LFB1-BUKRS
*     AND BELNR IN SBELNR
*     AND GJAHR = ?.

    PUT BKPF.

ENDFORM.                                    "PUT_BKPF

*---------------------------------------------------------------------*
* Authority Check for node BKPF
*---------------------------------------------------------------------*

* FORM AUTHORITYCHECK_BKPF.
*   AUTHORITY-CHECK ...
* ENDFORM.                                  "AUTHORITY_CHECK_BKPF
```

Include Program DBTEST_LDBFXXX

```
*---------------------------------------------------------------------*
* BEFORE_EVENT will be called before event EVENT
* Possible values for EVENT: 'START-OF-SELECTION'
*---------------------------------------------------------------------*
* FORM BEFORE_EVENT USING EVENT.
*   CASE EVENT.
*    WHEN 'START-OF-SELECTION'
*
*   ENDCASE.
* ENDFORM.                    "BEFORE_EVENT

*---------------------------------------------------------------------*
* AFTER_EVENT will be called after event EVENT
* Possible values for EVENT: 'END-OF-SELECTION'
*---------------------------------------------------------------------*
* FORM AFTER_EVENT USING EVENT.
```

```
*  CASE EVENT.
*    WHEN 'END-OF-SELECTION'
*
*  ENDCASE.
* ENDFORM.                    "AFTER_EVENT
*---------------------------------------------------------------
*
* Initialize global data for multiple processing of
* one logical database.
* Set returncode:
*      0 -> all data are initialized, multiple processing
o.k.
*  other -> no multiple processing allowed
*---------------------------------------------------------------*
FORM  LDB_PROCESS_INIT CHANGING SUBRC LIKE SY-SUBRC.

ENDFORM.                      "LDB_PROCESS_INIT
*---------------------------------------------------------------*
* LDB_PROCESS_CHECK_SELECTIONS is called
* after select-options and parameters are filled
*---------------------------------------------------------------*
FORM  LDB_PROCESS_CHECK_SELECTIONS CHANGING SUBRC LIKE SY-
SUBRC
                                            MSG LIKE
SYMSG.

ENDFORM.                      "LDB_PROCESS_CHECK_SELECTIONS
*-----------------------------------------------------------*
* Initialize selection screen (processed before PBO)
*-----------------------------------------------------------*
FORM INIT.

ENDFORM.                      "INIT.
*-----------------------------------------------------------*
* PBO of selection screen (processed always after ENTER)
*-----------------------------------------------------------*
FORM PBO.

ENDFORM.                      "PBO.
*-----------------------------------------------------------*
* PAI of selection screen (processed always after ENTER)
*-----------------------------------------------------------*
FORM PAI USING FNAME MARK.
* CASE FNAME.
*   WHEN 'SLIFNR  '.
*   WHEN 'SBUKRS  '.
*   WHEN 'SBELNR  '.
*   WHEN 'SGJAHR  '.
*   WHEN '*'.
```

```
* ENDCASE.
ENDFORM.                    "PAI
```

Include Program DBTEST_LDBSXXX

```
************************************************************

* !!! PLEASE DO NOT CHANGE MANUALLY (BEGIN OF BLOCK) !!!!!!
*
*----------------------------------------------------------
*
* Data structures for search pattern selection
*
* !!! PLEASE DO NOT CHANGE MANUALLY (END OF BLOCK) !!!!!!!!
*
************************************************************


*----------------------------------------------------------
*
* PUT_TEST_LDB_SP.
* Processed when search pattern selection is used,
* i.e. user input into PARAMETERS p_sp AS SEARCH PATTERN
* STRUCTURE.
*----------------------------------------------------------
*

* FORM PUT_TEST_LDB_SP.

* ENDFORM.                              "PUT_EXAMPLE_SP
```

The function of most of the subroutines is described in the section Structure of Logical Databases [Page 1166].

The comment symbols (*) before the ABAP statements that are optional can be deleted, and the question marks replaced by appropriate expressions. When you check the syntax of the program, all of the include programs that conform to the naming convention and the selection include are also checked.

In the subroutines PUT_<node>, SELECT statements are generated with conditions in their WHERE clauses for the primary key fields of the node. Note that these statements do not observe the performance rules [Page 1103] for Open SQL statements. In particular, the PUT_<node> subroutines for a subtree a structure form nested SELECTED loops, which you should avoid. Instead, you can buffer the data in internal tables, and pass it to the application program from there using the PUT statement. However, for technical reasons, the PUT <node> statement should always occur in a subroutine whose name begins PUT_<node>.

If the selection contains dynamic selections or field selections for a node, the system generates the corresponding statements in the subroutine PUT_NODE, and adjusts the automatically-generated SELECT statements, as in the example for the node BKPF. The following sections explain how you can process user input in the dynamic selections and column specifications after GET statements.

Generated Data Objects and Subroutines

Some internal tables and other subroutines are automatically generated, and can be used in the program.

- The internal table GET_EVENT contains the nodes of the logical database that are requested by the user in GET statements. The table is generated as follows:

  ```
  DATA: BEGIN OF GET_EVENTS OCCURS 10,
        NODE(10),
        KIND,
      END OF GET_EVENTS.
  ```

  Each line contains the name of a node of the logical database in the NODE field. The KIND field specifies whether and how the node is requested by the user.

  - KIND = 'X': Node is addressed in GET and GET LATE.
  - KIND = 'G': Node is addressed only in GET.
  - KIND = 'L': Node is addressed only in GET LATE.
  - KIND = 'P': Node is addressed neither in GET nor in GET LATE. However, a subordinate node is addressed in GET or GET LATE.
  - KIND = ' ': Node is addressed neither in GET nor in GET LATE. No subordinate node is addressed either.

- The subroutines BEFORE_EVENT, AFTER_EVENT, and PUT_<ldb>_SP are generated as comments in the database program (see example above). You can modify them and activate them by deleting the comment characters (*). BEFORE_EVENT is called before the event specified in the parameter EVENT is processed. AFTER_EVENT is called after the event specified in the parameter EVENT is processed. PUT_<ldb>_SP is called for processing values instead of PUT_<root> if a search help is used for selection. <root> is the root node of the logical database.

Dynamic Selections in the Database Program [Page 1208]

Field Selections in the Database Program [Page 1212]

Search Helps in the Database Program [Page 1215]

Independent Calls and the Database Program [Page 1219]

Dynamic Selections in the Database Program

The statement

SELECTION-SCREEN DYNAMIC SELECTIONS FOR NODE|TABLE <node>.

declares a node <node> of a logical database for dynamic selections in the selection include.

To use the dynamic selections in the SELECT statements of the subroutine PUT_<node>, you must use the data object DYN_SEL. The data object DYN_SEL is automatically generated in the logical database program as follows:

```
TYPE-POOLS RSDS.

DATA DYN_SEL TYPE RSDS_TYPE.
```

You do not have to program these lines yourself. The data object DYN_SEL is available in the database program but not in a connected executable program.

The type RSDS_TYPE of the data object is defined in the type group RSDS as follows:

```
TYPE-POOL RSDS.

* WHERE-clauses ------------------------------

TYPES: RSDS_WHERE_TAB LIKE RSDSWHERE OCCURS 5.

TYPES: BEGIN OF RSDS_WHERE,
         TABLENAME LIKE RSDSTABS-PRIM_TAB,
         WHERE_TAB TYPE RSDS_WHERE_TAB,
       END OF RSDS_WHERE.

TYPES: RSDS_TWHERE TYPE RSDS_WHERE OCCURS 5.

* Expressions Polish notation ----------------

TYPES: RSDS_EXPR_TAB LIKE RSDSEXPR OCCURS 10.

TYPES: BEGIN OF RSDS_EXPR,
         TABLENAME LIKE RSDSTABS-PRIM_TAB,
         EXPR_TAB TYPE RSDS_EXPR_TAB,
       END OF RSDS_EXPR.

TYPES: RSDS_TEXPR  TYPE RSDS_EXPR  OCCURS 10.

* Selections as RANGES-tables -----------------

TYPES: RSDS_SELOPT_T LIKE RSDSSELOPT OCCURS 10.

TYPES: BEGIN OF RSDS_FRANGE,
         FIELDNAME LIKE RSDSTABS-PRIM_FNAME,
         SELOPT_T TYPE RSDS_SELOPT_T,
       END OF RSDS_FRANGE.

TYPES: RSDS_FRANGE_T TYPE RSDS_FRANGE OCCURS 10.

TYPES: BEGIN OF RSDS_RANGE,
         TABLENAME LIKE RSDSTABS-PRIM_TAB,
         FRANGE_T TYPE RSDS_FRANGE_T,
       END OF RSDS_RANGE.

TYPES: RSDS_TRANGE TYPE RSDS_RANGE OCCURS 10.
```

* Definition of RSDS_TYPE

```
TYPES: BEGIN OF RSDS_TYPE,
         CLAUSES TYPE RSDS_TWHERE,
         TEXPR   TYPE RSDS_TEXPR,
         TRANGE  TYPE RSDS_TRANGE,
       END OF RSDS_TYPE.
```

It is a deep structure with the following components:

CLAUSES

CLAUSES contains the dynamic selections entered by the user (or possibly program-internal selection criteria) as internal tables, which you can use directly in dynamic WHERE clauses [Page 1064].

CLAUSES is an internal table that contains another internal table WHERE_TAB as a component. Each line of the CLAUSES-TABLENAME column contains the name of a node designated for dynamic selections. For each of these database tables, the WHERE_TAB tables contain the selection criteria of the dynamic selections. The WHERE_TAB tables have a format that allows you to use them directly in dynamic WHERE clauses.

To use WHERE_TAB in the logical database, you must program the dynamic WHERE clause for each node <node> designated for dynamic selection in the corresponding subroutine PUT_<node>. For this, the corresponding internal table WHERE_TAB for <node> must be read from the data object DYN_SEL. The following example shows how you can use a local data object in the subroutine for this purpose:

Suppose the database table SCARR is the root node of the logical database ZHK and that SPFLI is its only subordinate node.

The selection Include DBZHKSEL contains the following lines:

```
SELECT-OPTIONS S_CARRID FOR SCARR-CARRID.

SELECT-OPTIONS S_CONNID FOR SPFLI-CONNID.

SELECTION-SCREEN DYNAMIC SELECTIONS FOR TABLE SCARR.
```

The subroutine PUT_SCARR of the database program SAPDBZHK uses the dynamic selection as follows:

```
FORM PUT_SCARR.

  STATICS: DYNAMIC_SELECTIONS TYPE RSDS_WHERE,
           FLAG_READ.

  IF FLAG_READ = SPACE.
    DYNAMIC_SELECTIONS-TABLENAME = 'SCARR'.
    READ TABLE DYN_SEL-CLAUSES
               WITH KEY DYNAMIC_SELECTIONS-TABLENAME
               INTO DYNAMIC_SELECTIONS.
    FLAG_READ = 'X'.
  ENDIF.

  SELECT * FROM SCARR
         WHERE  CARRID IN S_CARRID
         AND   (DYNAMIC_SELECTIONS-WHERE_TAB).
```

Dynamic Selections in the Database Program

```
      PUT SCARR.
    ENDSELECT.
  ENDFORM.
```

The line of the internal table DYN_SEL-CLAUSES that contains the value SCARR in column DYN_SEL-CLAUSES-TABLENAME is read into the local structure DYNAMIC_SELECTIONS. The STATICS statements and the FLAG_READ field ensure that table DYN_SEL is only read once during each program execution. The table DYNAMIC_SELECTIONS-WHERE_TAB is used in the dynamic WHERE clause.

Each executable program that uses ZHK as its logical database and contains a NODES or TABLES statement for SCARR or SPFLI now offers dynamic selections for the fields in table SCARR on its selection screen. The dynamic WHERE clause means that the logical database only reads the lines that satisfy the selection conditions on the selection screen and in the dynamic selections.

TEXPR

TEXPR contains the selections of the dynamic selections in an internal format (Polish notation).You can use this format with function modules FREE_SELECTIONS_INIT and FREE_SELECTIONS_DIALOG in order to work with dynamic selections within a program (for more information, see the documentation of these function modules).

TRANGE

TRANGE contains the selections from the dynamic selection as RANGES tables [Page 704] that you can use with the IN operator in a WHERE clause or logical expression.

TRANGE is an internal table that contains another internal table FRANGE_T as a component. Each line in the column TRANGE-TABLENAME contains the name of a node that is designated for dynamic selections. For each of these nodes, the FRANGE_T tables contain the selection criteria of the dynamic selections in the format of RANGES tables. FRANGE_T contains a FIELDNAME column that contains the fields of the node for which the RANGES tables are defined. The other component of FRANGE_T, SELOPT_T, contains the actual RANGES tables.

With TRANGE, you can access the selections of individual database columns directly . Furthermore, it is easier to modify selection criteria stored in the RANGES format than those stored in the WHERE clause format. The following example shows how you can use local data objects of the corresponding subroutine PUT_<node> to work with TRANGE:

Suppose the database table SCARR is the root node of the logical database ZHK and that SPFLI is its single branch.

The selection Include DBZHKSEL contains the following lines:

SELECT-OPTIONS S_CARRID FOR SCARR-CARRID.

SELECT-OPTIONS S_CONNID FOR SPFLI-CONNID.

SELECTION-SCREEN DYNAMIC SELECTIONS FOR TABLE SCARR.

The subroutine PUT_SCARR of the database program SAPDBZHK uses the dynamic selection as follows:

```
FORM PUT_SCARR.
```

```
  STATICS: DYNAMIC_RANGES TYPE RSDS_RANGE,
           DYNAMIC_RANGE1 TYPE RSDS_FRANGE,
           DYNAMIC_RANGE2 TYPE RSDS_FRANGE,
           FLAG_READ.

  IF FLAG_READ = SPACE.

    DYNAMIC_RANGES-TABLENAME = 'SCARR'.
    READ TABLE DYN_SEL-TRANGE
               WITH KEY DYNAMIC_RANGES-TABLENAME
               INTO DYNAMIC_RANGES.

    DYNAMIC_RANGE1-FIELDNAME = 'CARRNAME'.
    READ TABLE DYNAMIC_RANGES-FRANGE_T
               WITH KEY DYNAMIC_RANGE1-FIELDNAME
               INTO DYNAMIC_RANGE1.

    DYNAMIC_RANGE2-FIELDNAME = 'CURRCODE'.
    READ TABLE DYNAMIC_RANGES-FRANGE_T
               WITH KEY DYNAMIC_RANGE2-FIELDNAME
               INTO DYNAMIC_RANGE2.

    FLAG_READ = 'X'.

  ENDIF.

  SELECT * FROM SCARR
         WHERE  CARRID   IN S_CARRID
         AND    CARRNAME IN DYNAMIC_RANGE1-SELOPT_T
         AND    CURRCODE IN DYNAMIC_RANGE2-SELOPT_T.

    PUT SCARR.

  ENDSELECT.

ENDFORM.
```

The line of the internal table DYN_SEL-TRANGE that contains the value 'SCARR' in column DYN_SEL-CLAUSES-TABLENAME is read into the local table DYNAMIC_RANGES. The nested tables DYNAMIC_RANGES-FRANGE_T are read into the local tables DYNAMIC-RANGE1 and DYNAMIC-RANGE2 according to the contents of DYNAMIC_RANGES-FIELDNAME. The STATICS statements and the FLAG_READ field assure that the tables are read only once each time the executable program is executed. After the READ statements, the nested tables SELOPT_T of the local tables contain the RANGES tables for the columns CARRNAME and CURRCODE of the database table SCARR.

The tables SELOPT_T are used in the SELECT statement directly as selection tables. Besides CARRNAME, CURRCODE and the primary key, there are no further columns in the database table SCARR. Therefore, this logical database has the same function as the logical database in the above example using the CLAUSES component.

Field Selections in the Database Program

The statement

SELECTION-SCREEN FIELD SELECTION FOR NODE|TABLE <node>.

declares a node <node> of a logical database for field selections in the selection include.

This means that you can list individual fields in the SELECT statements of the corresponding subroutine PUT_<node>, instead of having to use SELECT * to select all columns. This allows you to minimize the amount of data transferred from the database [Page 1107], which can often improve performance.

For each node for which field selection is allowed, you can specify the columns that you want to read using the FIELD addition to the GET statement in the executable program or in the FIELD_SELECTION parameter of the function module LDB_PROCESS. The database program of the logical database can access the names of the columns in the data object SELECT_FIELDS. The data object SELECT_FIELDS is automatically generated in the logical database program as follows:

TYPE-POOLS RSFS.

DATA SELECT_FIELDS TYPE RSFS_FIELDS.

You do not have to program these lines yourself. The data object SELECT_FIELDS is available in the database program and also in each connected executable program.

The type RSDS_FIELDS of the data object is defined in the type group RSDS as follows:

```
TYPE-POOL RSFS.

* Fields to be selected per table

TYPES: BEGIN OF RSFS_TAB_FIELDS,
         TABLENAME LIKE RSDSTABS-PRIM_TAB,
         FIELDS LIKE RSFS_STRUC OCCURS 10,
       END OF RSFS_TAB_FIELDS.

* Fields to be selected for all tables

TYPES: RSFS_FIELDS TYPE RSFS_TAB_FIELDS OCCURS 10.
```

RSDS_FIELDS is a deep internal table with the components TABLENAME and FIELDS. Each line of the TABLENAME column contains the name of a node that is designated for field selection. The table FIELDS contains the columns specified in the GET statements in the application program for each of these nodes. The FIELDS table have a format that allows you to use them directly in dynamic SELECT lists in SELECT statements.

To use the names of the columns in the logical database, you can specify the dynamic list FIELDS in the SELECT clause of the SELECT statements in the subroutine PUT_<node> for each node <node> for which field selections are allowed. To do this, you must read the corresponding internal table from the internal table SELECT_FIELDS. The following example shows how you can use a local data object of the subroutine for this purpose:

Suppose the database table SCARR is the root node of the logical database ZHK and that SPFLI is its only subordinate node.

The selection Include DBZHKSEL contains the following lines:

```
SELECT-OPTIONS S_CARRID FOR SCARR-CARRID.

SELECT-OPTIONS S_CONNID FOR SPFLI-CONNID.

SELECTION-SCREEN FIELD SELECTION FOR TABLE SPFLI.
```

The subroutine PUT_SCARR of the database program SAPDBZHK uses the field selections as follows:

```
FORM PUT_SPFLI.

  STATICS: FIELDLISTS TYPE RSFS_TAB_FIELDS,
           FLAG_READ.

  IF FLAG_READ = SPACE.
    FIELDLISTS-TABLENAME = 'SPFLI'.
    READ TABLE SELECT_FIELDS WITH KEY FIELDLISTS-TABLENAME
                             INTO FIELDLISTS.
    FLAG_READ = 'X'.
  ENDIF.

  SELECT (FIELDLISTS-FIELDS)
         INTO CORRESPONDING FIELDS OF SPFLI FROM SPFLI
         WHERE CARRID = SCARR-CARRID
           AND CONNID IN S_CONNID.

    PUT SPFLI.

  ENDSELECT.

ENDFORM.
```

The line of the internal table SELECT_FIELDS that contains the value 'SCARR' in column SELECT_FIELD-TABLENAME is read into the local structure FIELDLISTS. The STATICS statements and the FLAG_READ field assure that the table DYN_SEL is read only once each time the executable program is run. The table FIELDLISTS-FIELDS is used in the dynamic SELECT clause.

An executable program to which the logical database HKZ is linked could now, for example, contain the following lines:

```
TABLES SPFLI.

GET SPFLI FIELDS CITYFROM CITYTO.

...
```

The FIELDS option of the GET statement defines which fields besides the primary key the logical database should read from the database table.

Internally, the system fills the table SELECT_FIELDS with the corresponding values. This can be demonstrated by adding the following lines to the program:

```
DATA: ITAB   LIKE SELECT_FIELDS,
      ITAB_L LIKE LINE OF ITAB,
      JTAB   LIKE ITAB_L-FIELDS,
      JTAB_L LIKE LINE OF JTAB.

START-OF-SELECTION.
```

```
ITAB = SELECT_FIELDS.
LOOP AT ITAB INTO ITAB_L.
  IF ITAB_L-TABLENAME = 'SPFLI'.
    JTAB = ITAB_L-FIELDS.
    LOOP AT JTAB INTO JTAB_L.
      WRITE / JTAB_L.
    ENDLOOP.
  ENDIF.
ENDLOOP.
```

These lines display the column names on the list as follows:

```
CITYTO

CITYFROM

MANDT

CARRID

CONNID
```

The fields of the primary key (MANDT, CARRID, CONNID) have been added automatically to the specified columns.

Search Helps in the Database Program

The statement

PARAMETERS <p> AS SEARCH PATTERN FOR TABLE <node>.

defines a framework in the selection include for a search help on the selection screen.

The key fields returned by the search help are placed in the internal table <ldb>_SP, from where they can be used by the database program. The subroutine PUT_<ldb>_SP is then called instead of PUT_<root> (where <ldb> is the name of the logical database).

In the subroutine PUT_<ldb>_SP, the logical database can use the information from the internal table to make the actual database access more efficient. It triggers the event GET <root> with the statement PUT_<root>, where <root> is the name of the root node. It is often useful to call the subroutine PUT_<root> from PUT_<ldb>_SP. PUT_<root> can then select the data and trigger the corresponding GET event in the PUT <root> statement. However, for technical reasons, the PUT <root> statement should always occur in a subroutine whose name begins PUT_<root>. The structure of the internal table <ldb>_SP and other automatically-generated tables is displayed as a comment in the generated source code of the database program. How the tables are used is also documented in the source code.

Let us look at a logical database ZZF with the root node KNA1 that is linked to the search help DEBI.

Let the selection Include DBZHKSEL contain the following lines:

```
SELECT-OPTIONS SKUNNR FOR KNA1-KUNNR.

PARAMETERS P_SP AS SEARCH PATTERN FOR NODE KNA1.
```

The source code of the database program now includes more comment lines which indicate that the following tables and fields were created in addition to those listed under :

Internal table ZZF_SP:

This table has the following data type:

```
DATA: BEGIN OF ZZF_SP OCCURS 1000,
          KUNNR              LIKE KNA1-KUNNR,
        END   OF ZZF_SP.
```

The search help selections for the user generate a hit list in the output fields of the search help. The hit list is available to the database program through the table ZZF_SP.

Internal table SP_FIELDS:

If a collective search help is assigned to the logical database, an elementary search help will normally only fill a selection of the output fields of the collective search help. The program can find out from table SP_FIELDS which fields are filled. Its structure is:

```
DATA: BEGIN OF SP_FIELDS OCCURS 10.
        INCLUDE STRUCTURE RSSPFIELDS.
DATA: END   OF SP_FIELDS.
```

The field SP_FIELDS-SUPPLIED is not equal to SPACE if a value was assigned to the field in SP_FIELDS-FIELDNAME by the search help.

Internal table SP_TABLES:

If the search help contains fields from different tables, the program can find out from the table SP_TABLES which ones are covered by the search help. The structure is:

```
DATA: BEGIN OF SP_TABLES OCCURS 10.
        INCLUDE STRUCTURE RSSPTABS.
DATA: END   OF SP_TABLES.
```

The field SP_TABLES-SUPPLIED is not equal to SPACE if a value was assigned to the table in SP_FIELDS-TABLENAME by the search help.

Field SP_EVENTS:

This is a field with length 200. Each byte in SP_EVENTS stands for a table in the logical database structure (for example, the first character stands for the root node). The contents of the individual positions have the following meaning for the corresponding node:

- 'X': The node is addressed in the application program using the GET statement. The search help has assigned values for the key fields to SP_<ldb>.
- 'R': The node is addressed in the application program using the GET statement. The search help has not assigned values for the key fields to SP_<ldb>.
- 'M': The node is not addressed in the application program using the GET statement, but the search help has assigned values for the key fields to SP_<ldb>.
- ' ': The node is not addressed in the application program using the GET statement, and the search help has not assigned values for the key fields to SP_<ldb>.

If the user selects all suppliers in the search help on the selection screen whose sort field begins with ABC, and this applies to customer numbers 17, 125, and 230, the above tables will be filled as follows:

ZZF_SP:

KUNNR
17
125
230

SP_FIELDS:

FIELDNAME	SUPPLIED
KUNNR	X

SP_TABLES:

TABLENAME	SUPPLIED
KNA1	X

The inactive subroutine PUT_ZZF_SP can, for example, be modified and activated as follows in order to use the entries from the internal table ZZF_SP:

```
FORM PUT_ZZF_SP.
```

```
  CASE SP_EVENTS(1).
    WHEN 'X' OR 'M'.
      PERFORM PUT_KNA1_SP.
    WHEN OTHERS.
      PERFORM PUT_KNA1.
    ENDCASE.

ENDFORM.

FORM PUT_KNA1_SP.

  SELECT * FROM KNA1 FOR ALL ENTRIES IN ZZF_SP
                     WHERE KUNNR = ZZF_SP_KUNNR.
    PUT KNA1.
  ENDSELECT.

ENDFORM.
```

The system uses the table GET_EVENTS to check whether the application program contains a GET statement for KNA1, or whether the search help has assigned values for the key fields. If this is true, the system calls PUT_KNA1_SP. This contains a SELECT loop for KNA1, in which the lines corresponding to the key fields in ZZF_SP are read. The SELECT loop contains the statement PUT KNA1.

Another possibility would be:

```
FORM PUT_ZZF_SP.

        * Fill selection table from ZZF_SP

          IF SP_EVENTS(1) NE SPACE.
            CLEAR SKUNNR. REFRESH SKUNNR.
            MOVE: 'I'  TO SKUNNR-SIGN,
                  'EQ' TO SKUNNR-OPTION.
            LOOP AT ZZF_SP.
              MOVE ZZF_SP-KUNNR TO SKUNNR-LOW.
              APPEND SKUNNR.
            ENDLOOP.
          ENDIF.

        * Select data and call GET KUNNR

          READ TABLE GET_EVENTS WITH KEY 'KNA1'.
          IF SY-SUBRC = 0 AND GET_EVENTS-KIND NE SPACE.
            PERFORM PUT_KUNNR.
          ENDIF.
```

This deletes the selection table SKUNNR for KNA1 and fills it with the values from ZZF_SP. The program uses the table GET_EVENTS to check whether the application program contains a GET statement for KNA1. If so, it calls the subroutine PUT_KUNNR. Here, the data from KNA1 is read according to the selection table SKUNNR, and the PUT KNA1 statement is executed.

If the database access uses a search help, you should of course, use dynamic selections and field selections.

You can also use search helps to improve performance. You can program different database accesses depending on the tables and fields that are used and filled using the internal tables GET_EVENTS, SP_FIELDS, and SP_TABLES. For example, you can use a view or a join and

place the entries in internal tables, which you can process later and then trigger the corresponding GET events.

Independent Calls and the Database Program

If you call logical databases independently using the function module LDB_PROCESS, you can call special subroutines in the database program:

LDB_PROCESS_INIT

If you want to call a logical database more than once in succession, the database program **must** contain the following subroutine:

```
FORM LDB_PROCESS_INIT CHANGING SUBRC LIKE SY-SUBRC.
  ...
  SUBRC = 0.
ENDFORM.
```

This is the first subroutine to be called in the database program. Once the parameter SUBRC has been set to 0, the logical database can perform the initialization routines required to allow it to be called more than once. If the parameter SUBRC is not set to 0, the function module LDB_PROCESS triggers the exception LDB_NOT_REENTRANT.

LDB_PROCESS_CHECK_SELECTIONS

When you call a logical database using LDB_PROCESS, there is no selection screen processing. Instead, the selections are passed to the function module as interface parameters. The PAI subroutine is not called. However, if you still want to check the selections, you can define the following subroutine in the database program:

```
FORM LDB_PROCESS_CHECK_SELECTIONS CHANGING SUBRC LIKE SY-SUBRC
                                  MSG ..LIKE SYMSG.
  ...
  SUBRC = ...
ENDFORM.
```

This subroutine is called after the parameters and selection tables of the selection screen have been filled from the interface parameters of LDB_PROCESS. If the parameter SUBRC is not set to 0, the function module LDB_PROCESS triggers the exception LDB_SELECTIONS_NOT_ACCEPTED. You can assign a message to the structured parameter MSG. This is available to the caller of the function module in the system fields SY-MSG... The eight components of MSG contain the message type, ID, and number, and up to four message variables.

Editing Other Components

This section describes other components of logical databases that you may want to change.

Selection Texts

The selection texts, that is, the texts displayed with the input fields on the selection screen, are normally the names of the selection criteria and parameters. These texts are maintained separately from the logical database as text elements [Ext.]. The user interface of a logical database is therefore independent of the language in which it was created. Instead, it always appears in the logon language of the current user.

Documentation

Logical databases are reusable modules that can be used by a wide range of application programs. You should therefore ensure that they are adequately documented.

Authorization Objects

To create a list of authorization objects that are checked in the database program, choose *Extras* → *Authorization objects*.

Checking Logical Databases

To check whether a logical database is correct and complete, choose *Check* from the initial screen of the Logical Database Builder. The result of the check includes:

- Whether the logical database exists
- Whether its short text exists
- Whether the structure exists
- Whether the selections exist
- Whether the database program exists
- Whether the database program is syntactically correct
- Whether authorization objects have been maintained
- Whether there is any documentation

Improving Performance

Changes to logical databases are immediately visible in all of the programs that use them. This allows you to tune the performance [Page 1103] of the Open SQL statements for all of these programs in one place. One way of improving performance is to allow the user to specify exactly what data should be read from the logical database.

You can use the following techniques to improve performance:

- Static selection criteria and parameters on the selection screen, possibly with default values and value lists defined in the logical database.
- Dynamic selections
- Search helps
- ABAP Dictionary views, or other methods that minimize the number of database reads
  - Early authorization checks - during selection screen processing if possible, not during the data selection itself.

There are no hard and fast rules for how to optimize a logical database, since what you can do depends heavily on the data you want to read. The following are therefore only general points that you can nevertheless observe when optimizing logical databases:

- The numerical relationship between the table contents at different levels of the structure is very important.

  If, for example, one line of a database table at one level of the structure includes exactly one line of the database table at the next level (case A), other optimizations may make more sense for a 1:100 or 1:1000 ratio (case B).

  In case A, you can improve the response time by using database views.

  In case B, you can use internal tables. The data is read by the database into an internal table. The internal table can then be processed within the logical database. You could also use joins or cursor processing.

- Some application programs only use part of the structure of the logical database, while other use all of its nodes. In this case, you have the following options for improving the performance of individual reports:

– You can use the table GET_EVENTS in the logical database program. When you have generated an executable program that uses the logical database, this table contains information about each node in the structure, and whether the corresponding GET statement appears in the program or not.

– In the selections of the logical database, you should use the statement

  SELECTION-SCREEN FIELD SELECTION FOR NODE|TABLE <node>.

  to allow field selections for all suitable nodes <node>. You can then use a field list in the corresponding SELECT statements instead of having to read the entire line.

Using Contexts

Contexts are objects within the ABAP Workbench that enable you to store details about the relationships between data. You use them in your ABAP programs to derive data which is dependent on a small number of key fields.

Contexts allow you to

- store processing logic from application programs in context programs, reducing the complexity of the application.
- make better use of recurring logic.
- use buffering to improve system performance.

What are Contexts? [Page 1224]

The Context Builder in the ABAP Workbench [Page 1225]

Using Contexts in ABAP Programs [Page 1246]

Working With Contexts - Hints [Page 1259]

What are Contexts?

Within the large quantities of data in the R/3 System database, there are always smaller sets of basic data that you can use to derive further information. In a relational database model, for example, these are the key fields of database tables.

When an application program requires further information in order to continue, this is often found in these small amounts of basic data. The relational links in the database are often used to read further data on the basis of this basic data, or further values are calculated from it using ABAP statements.

It is often the case that certain relationships between data are always used in the same form to get further data, either within a single program or in a whole range of programs. This means that a particular set of database accesses or calculations is repeatedly executed, despite the fact that the result already exists in the system. Contexts provide a way of avoiding this unnecessary system load.

Contexts are objects within the ABAP Workbench, consisting of key input fields, the relationships between these fields, and other fields which can be derived from them. Contexts can link these derived fields by foreign key relationships between tables, by function modules, or by other contexts. You define contexts abstractly in the ABAP Workbench (Transaction SE33), and use instances of them as runtime objects in your ABAP programs.

In the ABAP Workbench, contexts consist of fields and modules. Their fields are divided into key fields and derived fields. The modules describe the relationship between the fields.

Technically, contexts are special ABAP programs. You can save them

- in a non-executable program, with the name CONTEXT_S_<context name>.For the example context DEMO_TRAVEL, this would be CONTEXT_S_DEMO_TRAVEL).
- in a generated executable program, with the name CONTEXT_X_<context name>.For the example context DEMO_TRAVEL, this would be CONTEXT_X_DEMO_TRAVEL).

In application programs, you work with instances of a context. You can use more than one instance of the same context. The application program supplies input values for the key fields in the context using the SUPPLY statement, and can query the derived fields from the instance using the DEMAND statement.

Each context has a cross-transaction buffer on the application server. When you query an instance for values, the context program searches first of all for a data record containing the corresponding key fields in the appropriate buffer. If one exists, the data is copied to the instance. If one does not exist, the context program derives the data from the key field values supplied and writes the resulting data record to the buffer.

Whenever, in a program, new key fields are entered into an instance that has already been previously supplied with them, the system invalidates all affected values that have already been derived. These values are only re-supplied the next time they are queried.

The Context Builder in the ABAP Workbench

Normally you will use existing contexts within your R/3 System, and only use the Context Builder to familiarize yourself with them and test them (see Finding and Displaying a Context [Page 1247]). However, if no suitable context already exists in the system, you can use the Context Builder to create your own.

This section describes how to use the Context Builder (Transaction SE33) in the ABAP Workbench.

Creating and Editing a Context [Page 1226]

Testing a Context [Page 1236]

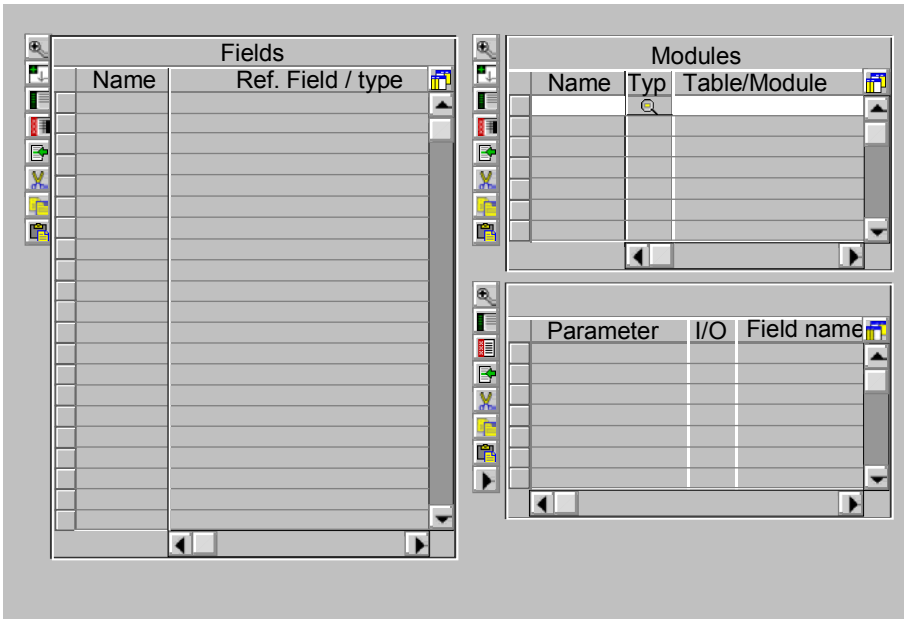
Buffering Contexts [Page 1238]

Creating and Editing a Context

To create a context, call the Context Builder (Transaction SE33) and enter a name for your new context in the *Context* field. In the *Sub-objects* group box, select *Fields and modules*, then choose *Create*.

On the next screen, you enter the attributes for your context:

When you have saved the attributes, choose *Fields and modules*. The context maintenance screen is then displayed.



There are three tables on this screen, in which you enter the Fields [Page 1241], Modules [Page 1243] and Interfaces [Page 1245] in your context. You can enlarge each of these tables by choosing *Enlarge* .

The contents of the individual tables are interdependent, and are filled automatically to a certain extent.

The next three sections use simple examples to demonstrate how you can create contexts using different modules.

Using Tables as Modules [Page 1228]

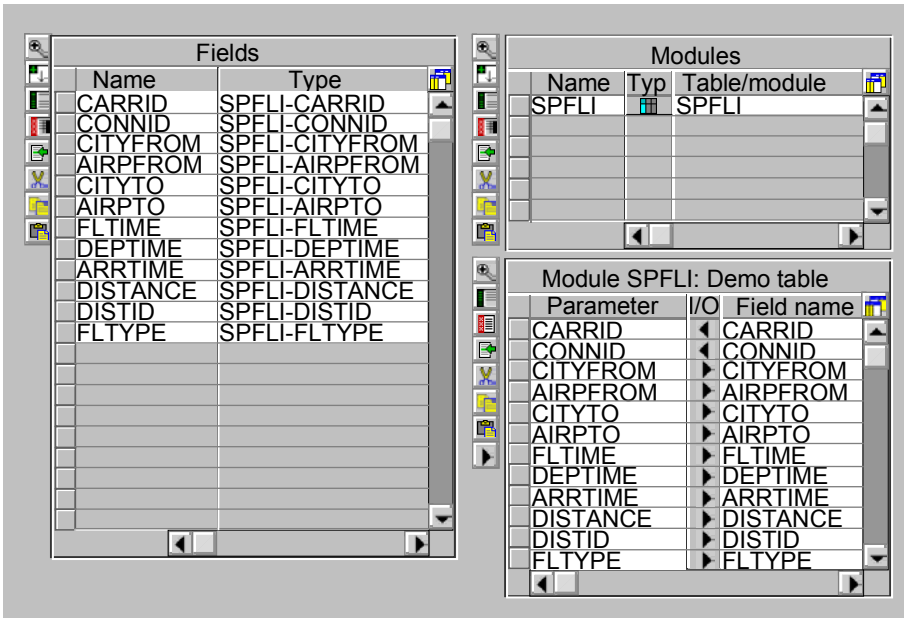
Using Function Modules as Modules [Page 1231]

Using Contexts as Modules [Page 1234]

Using Tables as Modules

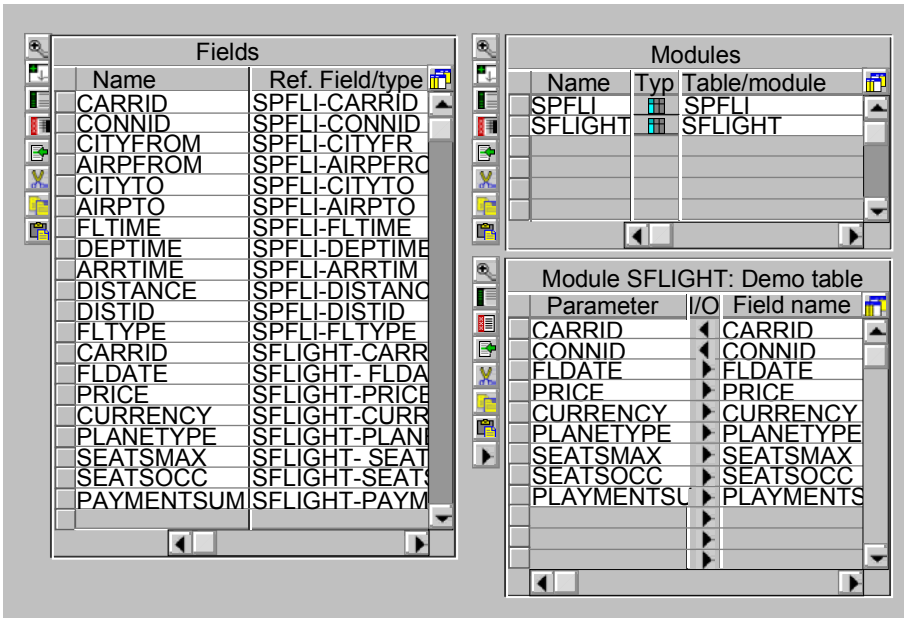
The following example demonstrates how to create a context called DOCU_TEST1, which has two modules with type table. The tables used are SPFLI and SFLIGHT from the data model BC_TRAVEL.

1. Enter SPFLI in the *Name* or *Table/Module* column of the Modules [Page 1243] table and choose *Enter*. The system then fills all of the tables of the context maintenance screen with unique entries as follows:



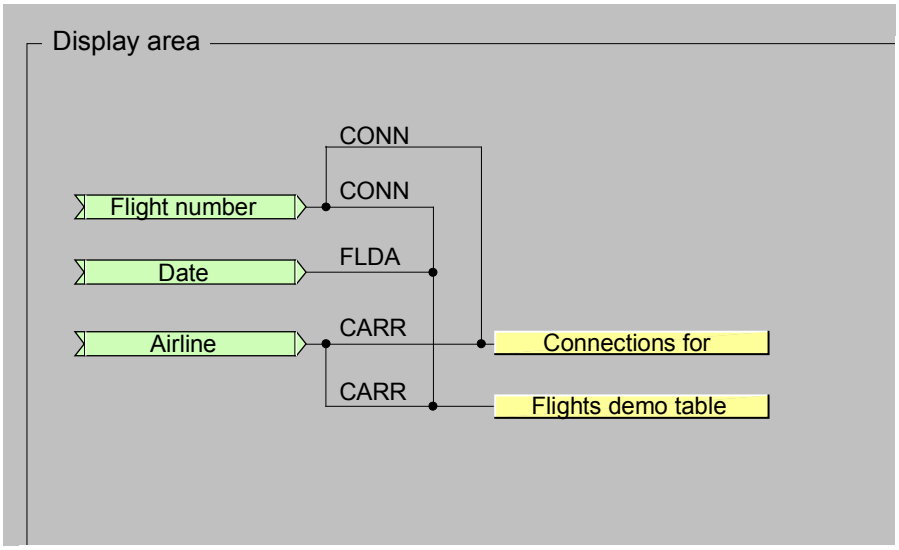
All of the columns of table SPFLI are used as fields in the context. The interface of module SPFLI shows that the key fields CARRID and CONNID are the input parameters.

2. Now enter SFLIGHT in the *Name* or *Table/Module* column of the Modules [Page 1243] table and choose *Enter*. The system now makes the following additional entries on the screen:



All of the columns in SFLIGHT which are not already contained in SPFLI are now also used in the context. The interface for module SFLIGHT shows that the key fields CARRID, CONNID and FLDATE are the input parameters of the new module.

3. Choose *Save*. Save the context.
4. Choose *Check*. The system checks the context for errors.
5. Choose *Network graphic*. The system displays the following graphic showing how values are derived within the context.



6. Generate the context. The system checks and saves your context before generating it.

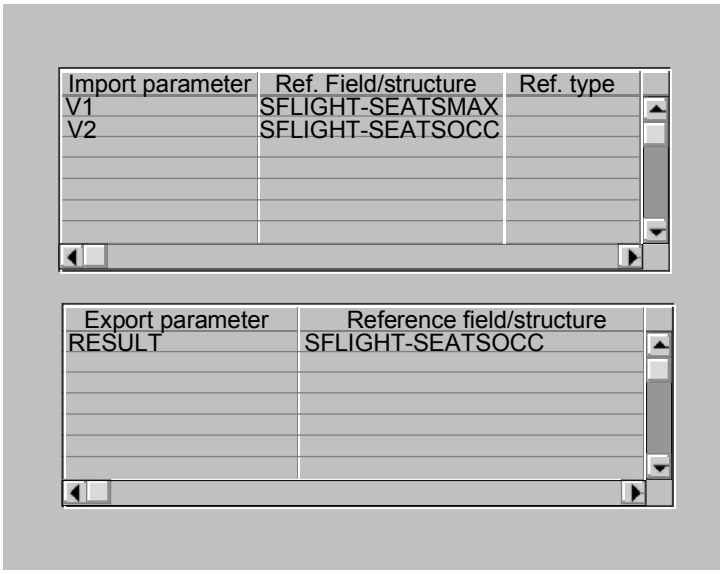
The context is finished and can be tested (see Testing a Context [Page 1236]) and used in programs (see Using Contexts in ABAP Programs [Page 1246]).

Using Function Modules as Modules

The following example demonstrates how to create a context called DOCU_TEST2 which uses a function module as a module.

Function module

The function module is called SUBTRACTION, and is part of the function group TESTCONTEXT. It has the following interface:



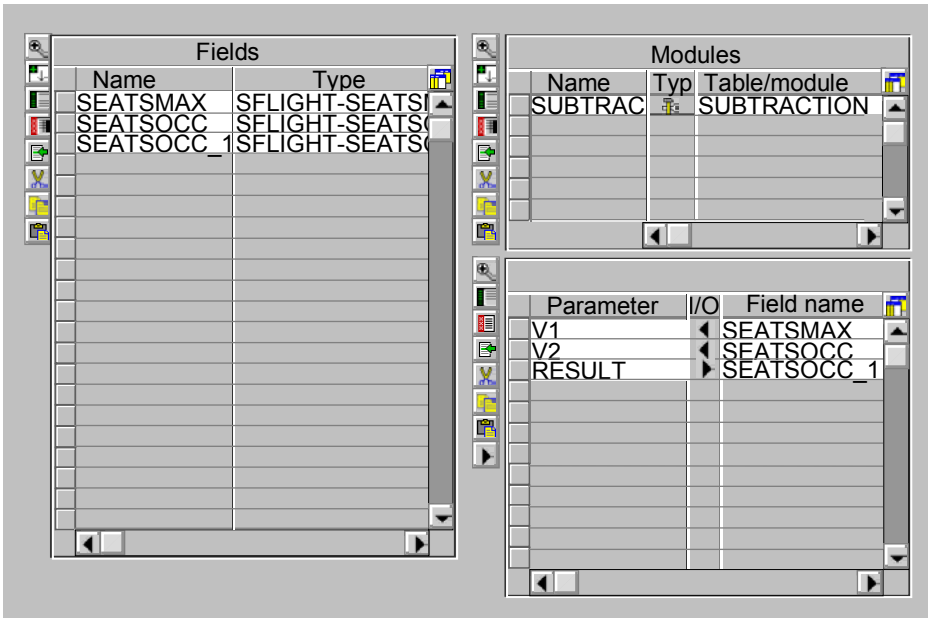
and the following source code:

```
FUNCTION SUBTRACTION.
  RESULT = V1 - V2.
ENDFUNCTION.
```

The function module subtracts import parameter V2 from the other import parameter V1, and returns the value in RESULT.

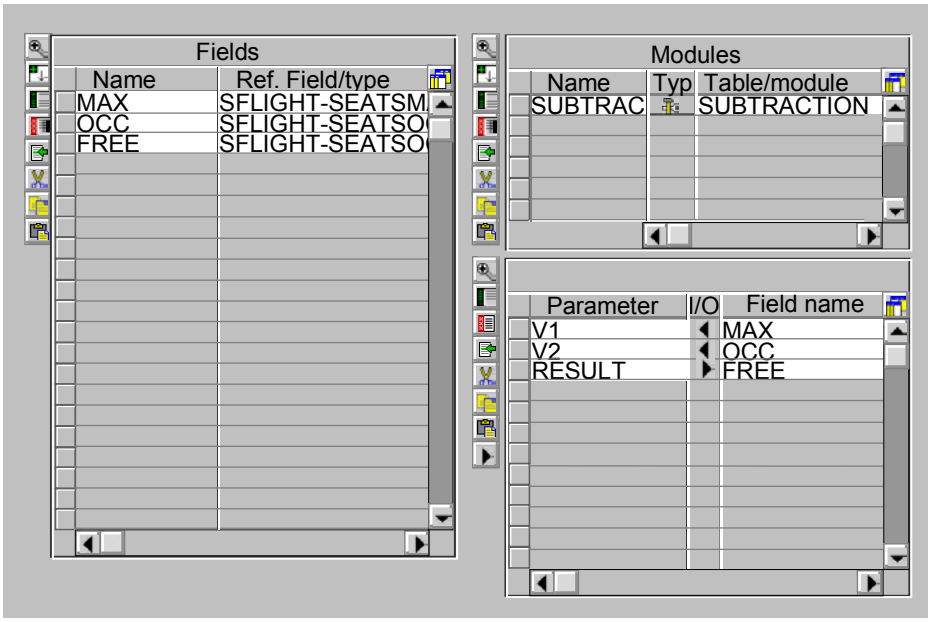
To create the context:

1. Enter SUBTRACTION in the *Name* or *Table/Module* column of the Modules [Page 1243] table and choose *Enter*. The system fills the tables on the context maintenance screen as follows:

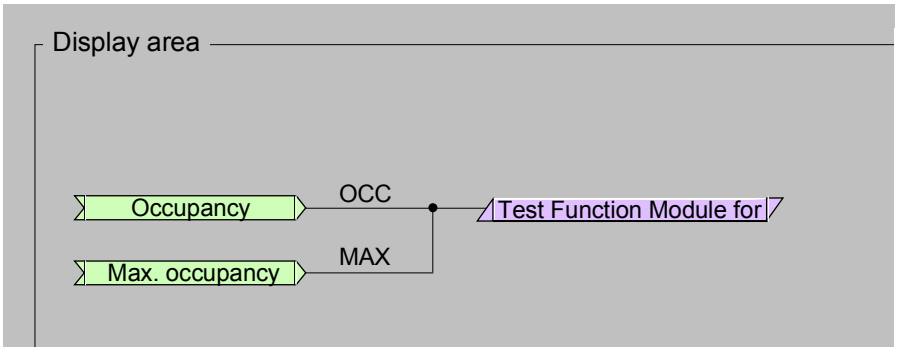


All of the interface parameters for the function module are used as fields in the context. The system generates the names of the context fields. The interface of the SUBTRACTION module shows that the import parameters V1 and V2 are assigned to the key fields SEATSMAX and SEATSOCC, and that the export parameter RESULT is assigned to the dependent field SEATSOCC_1.

2. You do not have to adopt these generated names. You can, instead, overwrite the context field names in the Fields [Page 1241] and Modules [Page 1243] table. For example, as follows:



3. Choose *Save*. Save the context.
4. Choose *Check*. The system checks the context for errors.
5. Choose *Network graphic*. The system displays the following graphic showing how values are derived within the context.



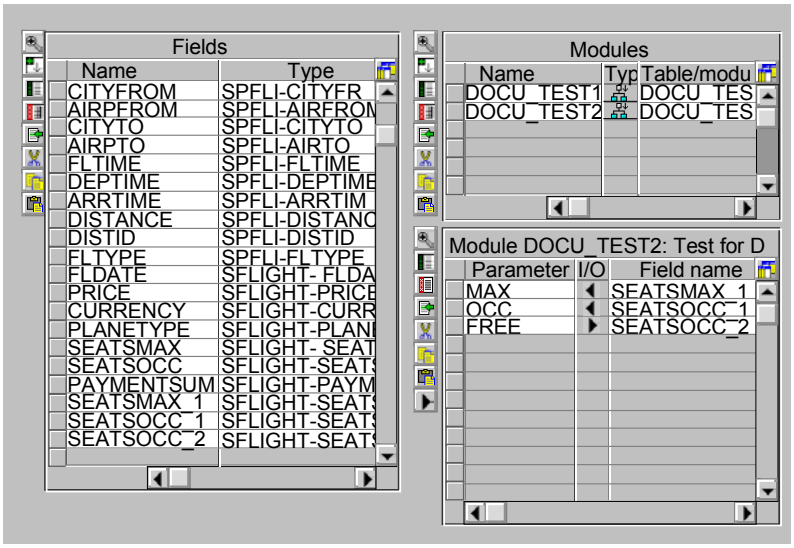
6. Generate the context. The system checks and saves your context before generating it.

The context is finished and can be tested (see Testing a Context [Page 1236]) and used in programs (see Using Contexts in ABAP Programs [Page 1246]).

Using Contexts as Modules

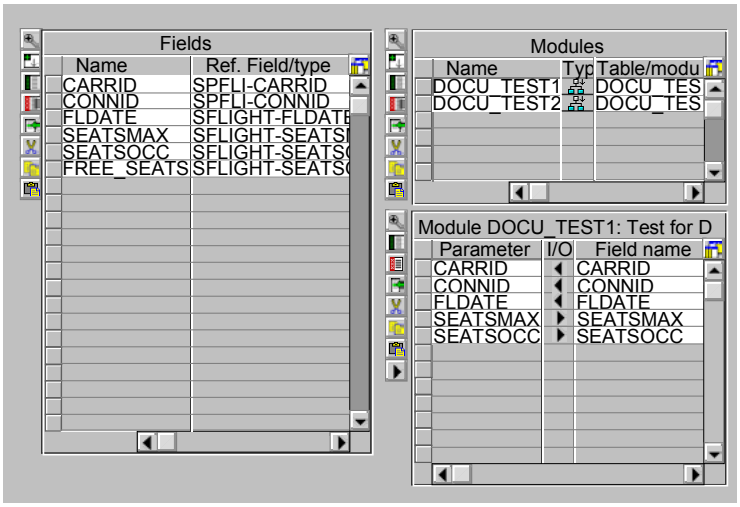
The following example demonstrates how to create a context called DOCU_TEST3, which uses two other contexts, DOCU_TEST1 and DOCU_TEST2, as modules. DOCU_TEST1 and DOCU_TEST2 have already been used in the preceding examples Using Tables as Modules [Page 1228] and Using Function Modules as Modules [Page 1231].

1. Enter DOCU_TEST1 and DOCU_TEST2 in the *Name* or *Table/Module* column of the Modules [Page 1243] table and choose *Enter*. The system fills the tables on the context maintenance screen as follows:



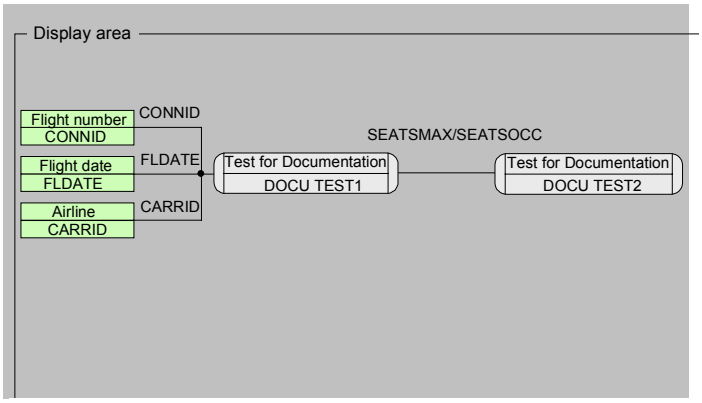
All fields from the two contexts are used as fields in this context. The system generates the names of the context fields.

2. You do not have to use all of the context fields. You can also change the context field names in the Fields [Page 1241] and Modules [Page 1243] tables:



In this example, we have kept the three key fields CARRID, CONNID and FLDATE, and deleted all dependent fields apart from SEATSMAX, SEATSOCC and FREE_SEATS. The output fields from module DOCU_TEST1, SEATSMAX and SEATSOCC are used as input fields for module DOCU_TEST2. The dependent field FREE_SEATS is filled from the output field FREE in DOCU_TEST2.

3. Choose *Save*. Save the context.
4. Choose *Check*. The system checks the context for errors.
5. Choose *Network graphic*. The system displays the following graphic showing how values are derived within the context.



6. Generate the context. The system checks and saves your context before generating it.

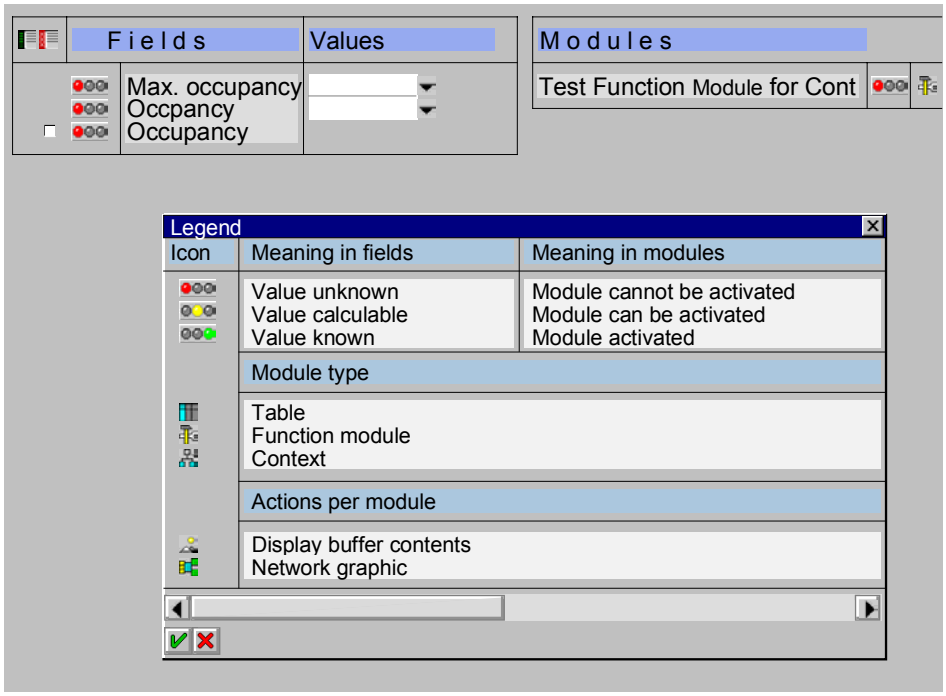
The context is finished and can be tested (see Testing a Context [Page 1236]) and used in programs (see Using Contexts in ABAP Programs [Page 1246]).

Testing a Context

This section describes how to test a context. It uses the context DOCU_TEST2 from the previous example Using Function Modules as Modules [Page 1231].

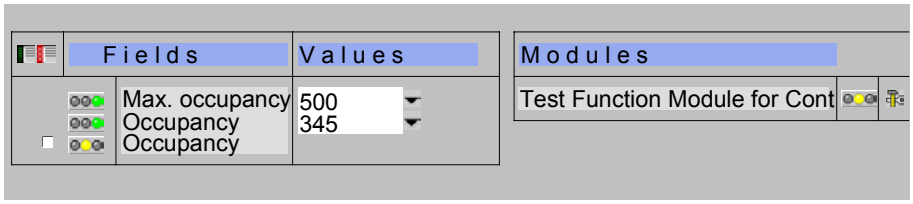
To test a context, call the Context Builder (Transaction SE33) and enter the name of the context you wish to test in the *Context* field. Choose *Test*. If texts appear in a language other than the logon language when you test the context, you must regenerate it once. Afterwards, the translated texts will appear.

The *Context Builder: Testing* screen is then displayed. To display the meanings of the various icons, choose *Legend*.

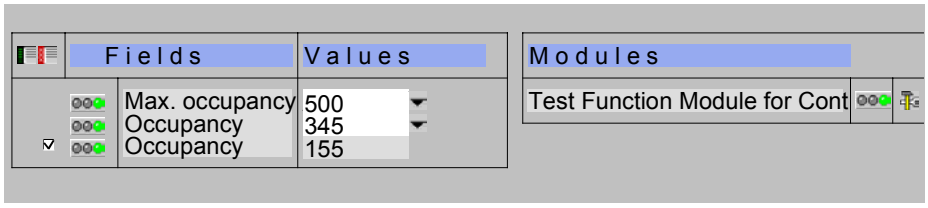


To test the context, enter values in the key fields of the context. Choose *Enter* to accept the input values.

If you want to use a blank character (SPACE) as an input value for a field, click on the traffic light symbol on the left hand side of the field to change the status of the input field.



If you enter values in all of the input fields, the system will be able to calculate all output fields. To activate the context, select the output fields required and choose *Calculate values*.



By choosing individual fields, you can use the *Context Builder: Testing* screen to display the relationships between fields and modules. For example, if you choose the *Test Function Module for Context* module, the following is displayed:

Buffering Contexts

The principal aim of contexts is to avoid the repeated calculation of frequently-used data which is derived from other key data. This is achieved by storing the derived data in the context buffer. Data for each module is stored separately in the buffer.

The main difference between the context buffer and the database buffers in the database interface or the SAP table buffer is that it is only periodically refreshed (every hour on the hour).The system makes no attempt to update it synchronously, or even nearly synchronously. For this reason, you cannot use the buffer for every context, or for every module within a context. Rather, you should only use it for data which does not often change.

In the *Buffer type* column of the Modules [Page 1243] table, you can set the type of buffer you want to use. This can be the context buffer itself (P), a temporary buffer (T), or no buffer at all (N).

The Individual Buffering Types

- Permanent (P)

  This is the default buffer setting, in which the **cross-transaction application buffer** is used. For more information about this buffer, see the keyword documentation for EXPORT/IMPORT TO/FROM SHARED BUFFER. The cross-transaction application buffer can contain up to 128 context entries. These entries are structured using a process similar to LRU (Least Recently Used).

  To display the contents of the context buffer on the current application server, choose *Goto → Display buffer* in the Context Builder. The buffer is reset every hour on the hour. You can also reset it manually in the Context Builder by choosing *Edit → Reset buffer*. This resets the buffer on **all application servers**.

- Temporary (T)

  If you choose this buffer type, the derived data is only buffered within a transaction. **Within** a transaction the buffer can contain up to 1024 entries. These entries are exported in bundles into the cross-transaction application buffer. The results of the various instances of a context are stored in the same buffer within the program.

- No buffer (N)

  If you choose *No buffer*, the derived data is not buffered. If you use this buffering type for all modules in a context, using the context will not improve system performance, but is merely a way of re-using program logic.

Permanent buffering can lead to problems when you are testing your Customizing settings. The Context Builder therefore allows you to de-activate the context buffer (buffering type P) during the current terminal session. To do this, choose *Settings → Buffering*. The system displays a dialog box. Select *Inactive*, and choose *Continue*. You can activate or de-activate the context buffer for a particular user by setting the user parameter BUF to Y or SPACE (default) or N using the *Maintain users* transaction (SU01).

Construction of the Buffer

The context buffer contains two buffer tables for each module:

- The I buffer (internal buffer), which saves each combination of module input parameters queried during the lifetime of the buffer, along with their derived values.

- The E buffer (external buffer), which assigns the derived values in the module to the key values in the context. The E buffer is filled each time a query is made. The first time a query is made using a particular combination of module input parameters, the results are written to the I buffer without any reference to the key values for the context. If, however, the same combination of module input parameters is queried more than once (so that the combination already exists in the I buffer), the system writes the results to the E buffer along with the context key values). It is therefore possible for the results of a particular combination of module input parameters to appear more than once in the E buffer, since different combinations of key fields can lead to the same set of module input parameters.

The various modules within a context are linked by the derivation schema. The input parameters of a module are either key fields or output parameters of another module. The entries in the E buffer for modules which depend on other, previous modules are the same as the I buffer entries for those previous modules. The buffering type of the E buffer for the dependent module is the same as the lowest buffering type of any of the modules on which it depends. Thus, if one of the previous modules has buffering type N, the system will not store any entries in the E buffer of the dependent module.

When you access the context buffer, either during testing or using the DEMAND statement, the system first searches in the E buffer of each module, followed by the I buffer. Access to the E buffer is more direct, and quicker than using the I buffer, which usually needs to be accessed more often.

The inclusion of entries in the E buffer which have been queried more than once is an acceptance of data redundancy as the price for quicker access. The I buffer is a filter for the E buffer, only allowing redundant data when absolutely necessary. Together, the I and E buffers provide a balance of quick access time and high probability of finding an appropriate entry.

When you create your own contexts, you should bear in mind the advantages of this buffering concept. Try to include as many relationships between data objects in a single context for each program, rather than using several contexts in the same program. You can also combine a group of contexts as modules of a single, larger context. Within a context, the system buffers all intermediate results in an optimal form. If you are using separate contexts, there is no link between any of the individual buffers.

Deleting a Buffer

You can delete the contents of a context buffer as follows:

- in the Context Builder (Transaction SE33), choose *Edit → Delete buffer → Local server* to delete the buffer contents for a context on the current application server, or *Edit → Delete buffer → All servers* to delete the buffer contents for a context on all application servers.

  You can use these functions when testing or buffering contexts.

- in ABAP programs, you can use the function modules CONTEXT_BUFFER_DELETE_LOCAL and CONTEXT_BUFFER_DELETE to delete the buffer contents for a context on the local server or all servers respectively.

  You can use these function modules in conjunction with changes to database contents to ensure that the contents of the context buffer are always current.

Buffering Contexts

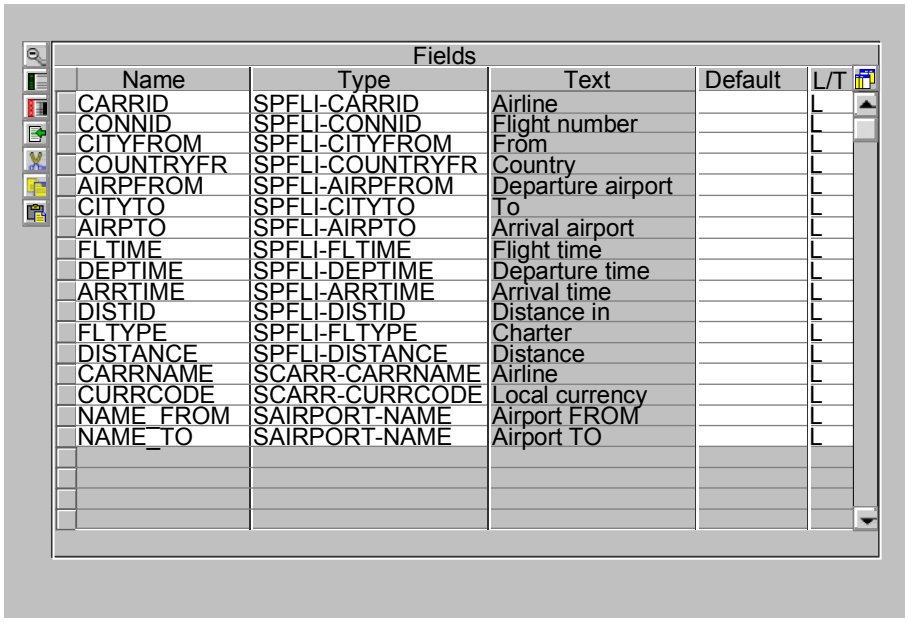
The following is an example which you could use in asynchronous update:

```
DATA CONTEXT_NAME LIKE RS33F-FRMID.

CONTEXT_NAME = 'DOCU_TEST1'.

...

CALL FUNCTION 'CONTEXT_BUFFER_DELETE' IN UPDATE TASK
    EXPORTING
        CONTEXT_ID = CONTEXT_NAME
    EXCEPTIONS
        OTHERS     = 0.

....

COMMIT WORK.
```

In the example, the system calls CONTEXT_BUFFER_DELETE as an update module. You could use it as the last update call following the database changes before the COMMIT WORK statement. For more information about asynchronous update, see Programming Database Updates [Page 1260]

Fields

The following illustration shows the *Fields* table for the sample context DEMO_TRAVEL:



Fields

Name	Type	Text	Default	L/T
CARRID	SPFLI-CARRID	Airline		L
CONNID	SPFLI-CONNID	Flight number		L
CITYFROM	SPFLI-CITYFROM	From		L
COUNTRYFR	SPFLI-COUNTRYFR	Country		L
AIRPFROM	SPFLI-AIRPFROM	Departure airport		L
CITYTO	SPFLI-CITYTO	To		L
AIRPTO	SPFLI-AIRPTO	Arrival airport		L
FLTIME	SPFLI-FLTIME	Flight time		L
DEPTIME	SPFLI-DEPTIME	Departure time		L
ARRTIME	SPFLI-ARRTIME	Arrival time		L
DISTID	SPFLI-DISTID	Distance in		L
FLTYPE	SPFLI-FLTYPE	Charter		L
DISTANCE	SPFLI-DISTANCE	Distance		L
CARRNAME	SCARR-CARRNAME	Airline		L
CURRCODE	SCARR-CURRCODE	Local currency		L
NAME_FROM	SAIRPORT-NAME	Airport FROM		L
NAME_TO	SAIRPORT-NAME	Airport TO		L

These, in turn, can become input parameters for further modules. You can enter new modules in the table directly, or fill it automatically by creating fields in the Modules [Page 1243] table. You can delete all unneeded fields from the context.

All fields in a context must have a reference to the ABAP Dictionary.

To edit the table, use the selection column and the pushbuttons on the left hand side of the table.

The table columns have the following meanings:

- *Name*

  The field names which you use to address the fields in the context.

- *Type*

  The Dictionary reference for the field. This can be a column of a database table or structure or a type in a type group.

- *Text*

  The short text for the field from the ABAP Dictionary.

- *Default value*

Fields

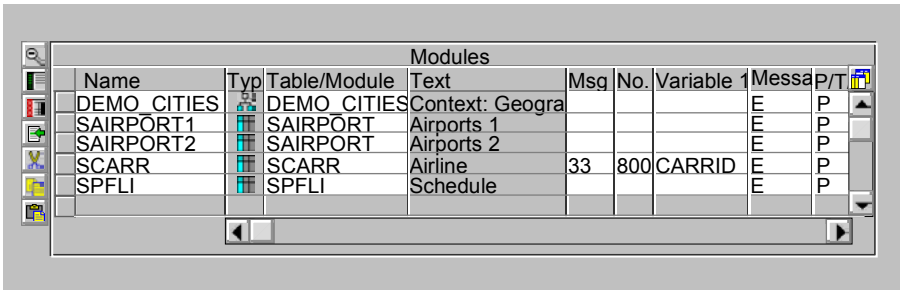
You can enter default values for key fields in this column. If you do this, the key fields of each instance of the context which you create in an application program will already be supplied with these values.

- *Like*

  In this column, you specify the type of reference between the field and the ABAP Dictionary. Enter L if the field refers to a database table or structure (LIKE), or T if the field refers to a type in a type group (TYPE). The system will normally assign the correct value automatically.

Modules

The following illustration shows the *Modules* table for the sample context DEMO_TRAVEL:



Modules

Name	Typ	Table/Module	Text	Msg	No.	Variable 1	Messa	P/T
DEMO_CITIES		DEMO_CITIES	Context: Geogra				E	P
SAIRPORT1		SAIRPORT	Airports 1				E	P
SAIRPORT2		SAIRPORT	Airports 2				E	P
SCARR		SCARR	Airline	33	800	CARRID	E	P
SPFLI		SPFLI	Schedule				E	P

Modules have input and output parameters. Together, they form the Interfaces [Page 1245] of the module. Modules describe the relationship between key fields and their dependent fields. Between them, all of the modules in a context derive all of the dependent fields from the key fields. Each individual module carries out its own part of that task. For example, one module uses some of the key fields as its input parameters to derive some of the dependent fields.

These, in turn, can become input parameters for further modules. You can enter new modules in the table directly, or fill it automatically by creating fields in the Fields [Page 1241] table. To edit the table, use the selection column and the pushbuttons on the left hand side of the table.

The table columns have the following meanings:

- *Name*

  The names of all modules.

- *Type*

  The module type. A module can be

  –  a table:

    The primary key fields of the table are input parameters. All other table fields are output parameters.

  –  a function module:

    The import parameters of the function module are input parameters. The export parameters of the function module are output parameters.

  –  another context:

    The key fields of contexts are input parameters. The dependent fields of contexts are output fields.

  Different modules can have the same type. This allows you to establish the same relationships between different groups of fields.

- *Table/Module*

  The name of the table, function module or context establishing the link between the context fields.

Modules

- *Text*

  A short text describing each module.

  - *Message ID, Number, Variable 1, ... , Variable 4, Type*

    A message, specified by its class (*Message ID*), number and type (*Type*). In the columns *Variable 1*, ... , *Variable 4*, you can specify context fields whose contents will replace the placeholders ('&') in the message. If the module is unable to supply all of the values requested in a DEMAND statement (because, for example, there is no dependent data in the database for the key fields), the system sends this message. You can use message handling in your application program for this message (see Message Handling in Contexts [Page 1253]).

  - *Buffer type*

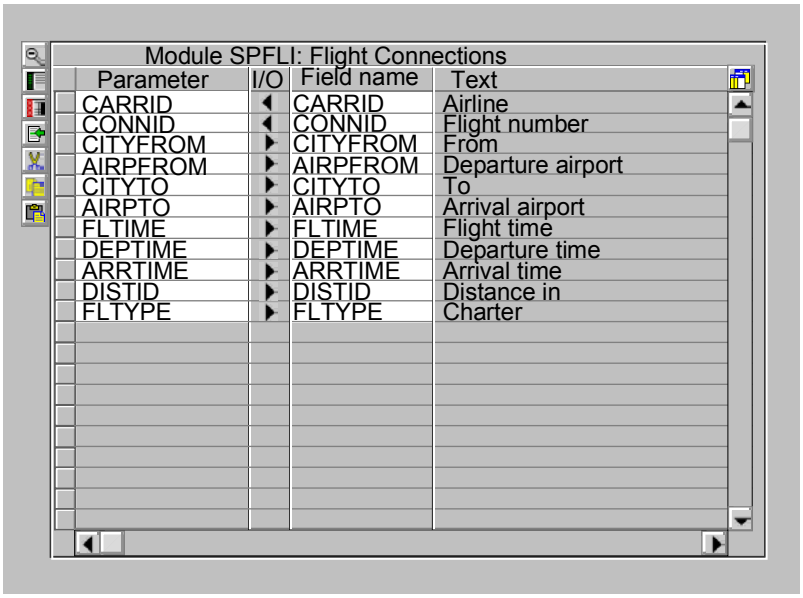
    The buffering type of each context. For more information about buffering, see Buffering Contexts [Page 1238]

Interfaces

When you select an entry in the Modules [Page 1243] table, the system fills the *Interface* table with the corresponding input and output parameters for that module.

When you select a dependent field in the Fields [Page 1241] table, the system finds the module which determines the field and fills the *Interface* table with the corresponding input and output parameters for that module.

For example, the interface for the module SPFLI in the sample context DEMO_TRAVEL looks like this:



Module SPFLI: Flight Connections

Parameter	I/O	Field name	Text
CARRID	◀	CARRID	Airline
CONNID	◀	CONNID	Flight number
CITYFROM	▶	CITYFROM	From
AIRPFROM	▶	AIRPFROM	Departure airport
CITYTO	▶	CITYTO	To
AIRPTO	▶	AIRPTO	Arrival airport
FLTIME	▶	FLTIME	Flight time
DEPTIME	▶	DEPTIME	Departure time
ARRTIME	▶	ARRTIME	Arrival time
DISTID	▶	DISTID	Distance in
FLTYPE	▶	FLTYPE	Charter

Module SPFLI is a database table in the ABAP Dictionary. The *Parameters* column contains its input and output parameters, and the *Field name* column contains the associated fields in the context. The direction of the arrow in the *I/O* column shows which parameters are input parameters and which are output parameters.

Using Contexts in ABAP Programs

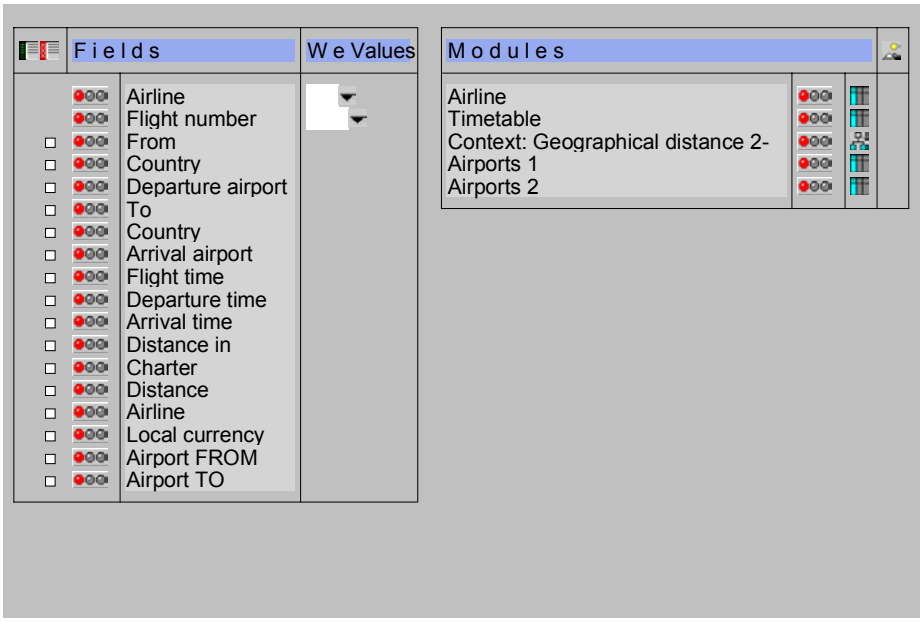
The following sections explain how to use contexts in your ABAP programs. As with logical databases, you will normally only use contexts supplied by SAP.

Finding and Displaying a Context

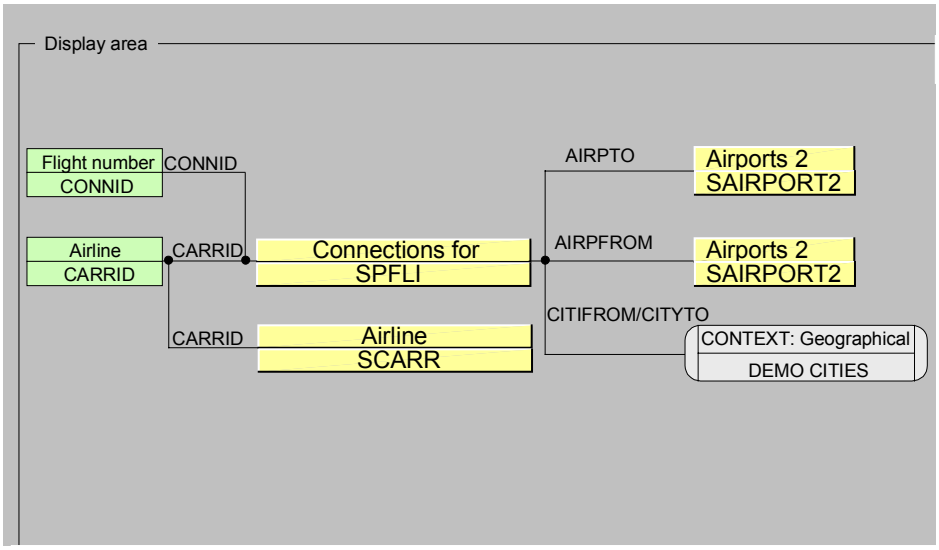
To find an existing context and to familiarize yourself with it, use the initial screen of the Context Builder (Transaction SE33).

- Use the possible entries help for the *Context* field to display a list of existing contexts.
- Use *Test* to display the names of the key fields and dependent fields, and the function of the context (see Testing a Context [Page 1236]).
- Use *Network graphic* to display the derivation schema for the context.
- Use *Display* with the Fields [Page 1241], Modules [Page 1243] and Interfaces [Page 1245] tables to display further information about the context field dependencies.

Enter the name DEMO_TRAVEL in the *Context* field on the initial screen of the Context Builder and choose *Test*.



You will see that the context has two key fields, from which the other fields are derived, and that a total of five modules are used to define the relationships between the fields. Four of these modules are database tables, and one is another context. If you choose *Network graphic*, you will see the names of the key fields (CARRID and CONNID) and their relationships to the modules.



To see the names of all context fields, display the Fields [Page 1241] table in the Context Builder.

You now have the most important information which you need to work with the context. Further information such as message texts and the buffering types for the modules are contained in the modules [Page 1243] table.

Creating an Instance of a Context

As objects within the ABAP Workbench, contexts store the relationships between data, but no data itself. In your ABAP programs, you work with instances of contexts. These are runtime instances of contexts containing the actual data.

To create a context instance, you must first declare the context in the program. You do this using the CONTEXTS statement:

Syntax

CONTEXTS <c>.

The context <c> must already exist as an object in the ABAP Workbench. The statement implicitly generates the special data type CONTEXT_<c>, which you can use to create context instances in a DATA statement:

Syntax

DATA <inst> TYPE CONTEXT_<c>.

This statement generates an instance <inst> of context <c>. You can create as many instances of a context as you like. Once you have created an instance, you can supply it with keywords (see Supplying Context Instances with Key Values [Page 1250]).

```
REPORT RSGCON01.

CONTEXTS DEMO_TRAVEL.

DATA: DEMO_TRAVEL_INST1 TYPE CONTEXT_DEMO_TRAVEL,
      DEMO_TRAVEL_INST2 TYPE CONTEXT_DEMO_TRAVEL.
```

This program extract generates two instances of the context DEMO_TRAVEL.

Supplying Context Instances with Key Values

Once you have created a context instance, you can supply values for its key fields. You do this using the SUPPLY statement:

Syntax

SUPPLY <key_1> = <f_1> ... <key_n> = <f_n> TO CONTEXT <inst>.

This statement supplies the values <f_n> to key fields <key_n> of a context instance <inst>. The fields of the context are contained in the Fields [Page 1241] table.

When you have specified values for the key fields, you can derive the dependent fields of the context instance (see Querying Data from Context Instances [Page 1251]).

If you assign new key fields to a context instance after deriving the dependent fields, the derived values are automatically invalidated, and will be re-calculated the next time you derive dependent values.

```
REPORT RSGCON01.

CONTEXTS DEMO_TRAVEL.

DATA: DEMO_TRAVEL_INST1 TYPE CONTEXT_DEMO_TRAVEL,
      DEMO_TRAVEL_INST2 TYPE CONTEXT_DEMO_TRAVEL.

SUPPLY CARRID = 'LH'
       CONNID = '400'
       TO CONTEXT DEMO_TRAVEL_INST1.

SUPPLY CARRID = 'AA'
       CONNID = '017'
       TO CONTEXT DEMO_TRAVEL_INST2.
```

This program extract generates two instances of the context DEMO_TRAVEL and supplies both with key values.

Querying Data from Context Instances

Once you have supplied a context instance with key values, you can query its dependent values. You do this using the DEMAND statement:

Syntax

DEMAND <val_1> = <f_1> ... <val_n> = <f_n> FROM CONTEXT <inst>
[MESSAGES INTO <itab>].

This statement fills the fields <f_n> with the derived values <val_n> from context instance <inst>. The fields of the context are contained in the Fields [Page 1241] table.

In doing this, the system carries out the following steps:

1. It checks to see whether the context instance already contains valid derived values. This is the case if the values have already been calculated in a previous DEMAND statement and the instance has not since been supplied with new key field values using the SUPPLY statement. In this case, these values are assigned to fields <f_n>.
2. If the context instance does not contain valid values, the system calculates new ones. It looks first in the context buffer (see Buffering Contexts [Page 1238]) for data records with the right key field values for the current context instance. If it finds the corresponding values, the system copies them as valid values into the context instance and assigns them to fields <f_n>.
3. If there are no appropriate values in the context buffer, the system derives the values according to the context definition. The system also searches the context buffer during the derivation for intermediate results, which it uses if they are valid. When it has derived the values, the system saves all results in the context buffer, copies the values to the context instance and assigns them to fields <f_n>.
4. If the system cannot determine the dependent values, it terminates the process, sets fields <f_n> to their initial values and outputs the user message stored in the Modules [Page 1243] table. You can handle these messages in your programs by using the MESSAGES option (see Message Handling In Contexts [Page 1253]).

```
REPORT RSGCON01.
DATA: C_FROM LIKE SPFLI-CITYFROM,
      C_TO   LIKE SPFLI-CITYTO,
      C_TIME LIKE SPFLI-FLTIME.
CONTEXTS DEMO_TRAVEL.
DATA: DEMO_TRAVEL_INST1 TYPE CONTEXT_DEMO_TRAVEL,
      DEMO_TRAVEL_INST2 TYPE CONTEXT_DEMO_TRAVEL.
SUPPLY CARRID = 'LH'
       CONNID = '400'
       TO CONTEXT DEMO_TRAVEL_INST1.
SUPPLY CARRID = 'AA'
       CONNID = '017'
       TO CONTEXT DEMO_TRAVEL_INST2.
DEMAND CITYFROM = C_FROM
       CITYTO   = C_TO
       FLTIME   = C_TIME
       FROM CONTEXT DEMO_TRAVEL_INST1.
```

Querying Data from Context Instances

```
WRITE: / C_FROM, C_TO, C_TIME.
DEMAND CITYFROM = C_FROM
       CITYTO   = C_TO
       FLTIME   = C_TIME
       FROM CONTEXT DEMO_TRAVEL_INST2.
WRITE: / C_FROM, C_TO, C_TIME.
```

This program generates two instances of context DEMO_TRAVEL, supplies them both with key values and reads three dependent values from each of them.

Message Handling in Contexts

If a context cannot derive dependent data when you test it or execute the DEMAND statement, it can send a message to the user.

The way in which messages are handled in contexts depends on the type of module in which the context could not determine data. Table and function module modules handle messages differently. Other contexts used as modules can be broken down to table and function module level.

Message Handling in Table Modules [Page 1254]

Message Handling in Function Module Modules [Page 1256]

Message Handling in Table Modules

If you want to send user messages from table modules, you must first have defined messages for the individual modules in the Modules [Page 1243] table. The system cannot output a message for a table module where none is defined, but will simply reset the corresponding dependent values.

When you query dependent data from context instances using the DEMAND statement, you can decide whether the system should handle the user message or whether you want to handle it yourself in the program.

Message Handling - System

If you want the system to handle the message, use the basic form of the DEMAND statement without the MESSAGES option. The system will then behave as though the following statement occurred after the DEMAND statement:

```
MESSAGE ID 'Message Id' TYPE 'T' NUMBER 'Number'
      WITH 'Variable 1' .... 'Variable 4'.
```

The system takes the arguments for the MESSAGE statement from the corresponding entries in the Modules [Page 1243] table. Note that the system reaction to the various message types depends on the current user dialog (dialog screen, selection screen or list).

Message Handling - Program

If you want to catch the message in your program, use the DEMAND statement using the option MESSAGES INTO <itab>. To do this, you need to define an internal table <itab> with the structure SYMSG. The system deletes the contents of <itab> table at the beginning of the DEMAND statement. Whilst it is processing the context, the system does not output or react to any messages, but writes their ID to <itab>. If there are messages in <itab> following the DEMAND statement, the system sets the return code SY-SUBRC to a value unequal to zero.

This enables you to prevent automatic message handling with contexts and allows you to program your own using the entries in <itab>. This is important, for example, if you want to make particular fields on an entry screen ready for input again. In this case, you must base your message handling on the fields (using the FIELDS statement in screen flow logic, or the ABAP statement AT SELECTION-SCREEN for selection screens), and not on fields in the context.

Suppose the Modules [Page 1243] table in context DOCU_TEST1 (see Using Tables as Modules [Page 1228]) contained the following entries for module SPFLI:

	Name	Message ID	No.	Variable 1	Variable 2	Variable 3	Variable 4	Messa	P/T/
	SPFLI	BCTRAIN	170	CARRID	CONNID			E	P

To demonstrate system message handling, execute the following program:

```
REPORT  RSGCON02.

DATA: C_FROM LIKE SPFLI-CITYFROM,
      C_TO   LIKE SPFLI-CITYTO.
```

```
CONTEXTS DOCU_TEST1.

DATA: CONTEXT_INST TYPE CONTEXT_DOCU_TEST1.

SUPPLY CARRID = 'XX'
       CONNID = '400'
       TO CONTEXT CONTEXT_INST.

DEMAND CITYFROM = C_FROM
       CITYTO   = C_TO
       FROM CONTEXT CONTEXT_INST.

WRITE: / C_FROM, C_TO.
```

The program terminates with the following error message:

```
No entries found for key XX 0400
```

To demonstrate program message handling, execute the following program:

```
REPORT  RSGCON03.

DATA: C_FROM LIKE SPFLI-CITYFROM,
      C_TO   LIKE SPFLI-CITYTO.

DATA ITAB LIKE SYMSG OCCURS 0 WITH HEADER LINE.

CONTEXTS DOCU_TEST1.

DATA: CONTEXT_INST TYPE CONTEXT_DOCU_TEST1.

SUPPLY CARRID = 'XX'
       CONNID = '400'
       TO CONTEXT CONTEXT_INST.

DEMAND CITYFROM = C_FROM
       CITYTO   = C_TO
       FROM CONTEXT CONTEXT_INST MESSAGES INTO ITAB.

WRITE: / C_FROM, C_TO.

LOOP AT ITAB.
  WRITE: / ITAB-MSGTY, ITAB-MSGID, ITAB-MSGNO,
           ITAB-MSGV1, ITAB-MSGV2.
ENDLOOP.
```

The program outputs the following:

```
E  AT  107  XX  0400
```

Instead of terminating with an error message, the program stores the message in the internal table ITAB, where it is available for you to process further.

Message Handling in Function Module Modules

In order for a message to be sent from a function module context module, the function module must contain the necessary exception handling. In other words, the function module must send the message using the MESSAGE ... RAISING statement. If the exception handling in the function module is programmed using the RAISE statement, the system reacts with a runtime error. For more information about exception handling in function modules, see Function Modules [Page 480].

The system sends the message specified in the MESSAGE... RAISING statement. Messages specified for function modules in the Modules [Page 1243] table are ignored.

When you query dependent data from context instances using the DEMAND statement, you can decide whether the system should handle the user message or whether you want to handle it yourself in the program.

Message Handling - System

If you want the system to handle the message, use the basic form of the DEMAND statement without the MESSAGES option. The system will then behave as though the MESSAGE statement in the function module occurred after the DEMAND statement. Note that the system reaction to the various message types depends on the current user dialog (dialog screen, selection screen or list).

Message Handling - Program

If you want to catch the message in your program, use the DEMAND statement using the option MESSAGES INTO <itab>. To do this, you need to define an internal table <itab> with the structure SYMSG. The system deletes the contents of <itab> table at the beginning of the DEMAND statement. Whilst it is processing the context, the system does not output or react to any messages, but writes their ID to <itab>. If there are messages in <itab> following the DEMAND statement, the system sets the return code SY-SUBRC to a value unequal to zero.

This enables you to prevent automatic message handling with contexts and allows you to program your own using the entries in <itab>. This is important, for example, if you want to make particular fields on an entry screen ready for input again. In this case, you must base your message handling on the fields (using the FIELDS statement in screen flow logic, or the ABAP statement AT SELECTION-SCREEN for selection screens), and not on fields in the context.

The function module PERCENT has the following source code:

```
FUNCTION PERCENT.

RESULT = V1 - V2.

IF V1 = 0.
  MESSAGE ID 'AT' TYPE 'E' NUMBER '012' RAISING DIV_ZERO.
ELSE.
  RESULT = RESULT * 100 / V1.
ENDIF.

ENDFUNCTION.
```

Context DOCU_TEST4 uses this function module as its only module, with key fields MAX and OCC for input parameters V1 and V2 and the dependent field PERCENT for the output parameter RESULT.

To demonstrate system message handling, execute the following program:

```
REPORT  RSGCON04.

DATA: INPUT_1 TYPE I,
      INPUT_2 TYPE I,
      PERCENTAGE TYPE I.

CONTEXTS DOCU_TEST4.

DATA: CONTEXT_INST TYPE CONTEXT_DOCU_TEST4.

SUPPLY MAX = 00
       OCC = 20
       TO CONTEXT CONTEXT_INST.

DEMAND PERCENT = PERCENTAGE
       FROM CONTEXT CONTEXT_INST.

WRITE: / 'Percentage: ', PERCENTAGE.
```

The program terminates with the following error message:

```
This is an error message
```

To demonstrate program message handling, execute the following program:

```
REPORT  RSGCON05.

DATA: INPUT_1 TYPE I,
      INPUT_2 TYPE I,
      PERCENTAGE TYPE I.

CONTEXTS DOCU_TEST4.

DATA: CONTEXT_INST TYPE CONTEXT_DOCU_TEST4.

DATA ITAB LIKE SYMSG OCCURS 0 WITH HEADER LINE.

SUPPLY MAX = 00
       OCC = 20
       TO CONTEXT CONTEXT_INST.

DEMAND PERCENT = PERCENTAGE
       FROM CONTEXT CONTEXT_INST MESSAGES INTO ITAB.

WRITE: / 'Percentage: ', PERCENTAGE.

LOOP AT ITAB.
  WRITE: / ITAB-MSGTY, ITAB-MSGID, ITAB-MSGNO,
           ITAB-MSGV1, ITAB-MSGV2.
ENDLOOP.
```

The program outputs the following:

```
PERCENTAGE:  0

E  AT  012
```

Instead of terminating with an error message, the program stores the message in the internal table ITAB, where it is available for you to process further.

Working With Contexts - Hints

Creating Contexts

- You should include only a small number of key fields in your contexts.
- The context graphic should not contain any isolated, unattached fields. In other words, all of the input parameters should be derived hierarchically from the key fields or output parameters of other modules. If this is not the case, you should split the context into two or more smaller contexts. Refer also to the hint in Buffering Contexts [Page 1238]
- You should restrict the number of derived fields to those you really need, because:

– Reading fewer fields reduces the load on the database

– Fewer fields means that the buffer requires less space

– The system can then read the buffer more quickly

– You can always add more fields later if required.

- If you use a table, a function module or a context more than once as a module within a context, the module name should be made up as follows:<object name>_<suffix>. When you test your context, the suffix is automatically displayed after the long text.

Using Contexts

- Since the SUPPLY and DEMAND statements are not runtime-intensive, using them repeatedly causes no problems. In particular,

– You should always use the SUPPLY command as soon as the key values are assigned. This reduces the risk of deriving obsolete values in the DEMAND statement. You do not need to make a preliminary query to see if the contents of the key fields have changed, since the system does this itself in the SUPPLY command.

– You should always use the DEMAND statement immediately before using the derived values. This is to ensure that you always use up-to-date values.

- You should use local data objects as target fields for the DEMAND statement. This reduces the risk of obsolete values being used in error.

Programming Database Updates

This section describes how to program database updates in the R/3 System.

See also the descriptions in the ABAP Editor under Utilities → Help on → ABAP Overview → Description of Syntax and Concepts → Transaction processing.

For detailed information about the statements that you use for database updates, see the keyword documentation in the ABAP Editor.

Transactions and Logical Units of Work

In everyday language, a transaction is a sequence of actions that logically belong together in a business sense and which either procure or process data. It covers a self-contained procedure, for example, generating a list of customers, creating a flight booking, or sending reminders to customers. From the point of view of the user, it forms a logical unit.

The completeness and correctness of data must be assured within this unit. In the middle of a transaction, the data will usually be inconsistent. For example, when you transfer an amount in financial accounting, this must first be deducted from one account before being credited to another. In between the two postings, the data is inconsistent, since the amount that you are posting does not exist in either account. It is essential for application programmers to know that their data is consistent at the end of the transaction. If an error occurs, it must be possible to undo the changes made within a logical process.

In the R/3 System, there are three terms frequently used in this connection:

Database Logical Unit of Work (LUW)

A database LUW is the mechanism used by the database to ensure that its data is always consistent.

SAP LUW

An SAP LUW is a logical unit consisting of dialog steps, whose changes are written to the database in a single database LUW.

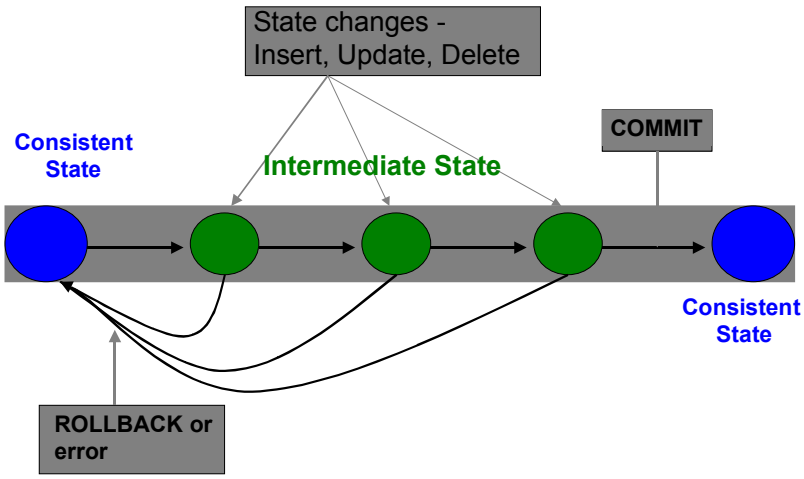
SAP Transaction

An SAP transaction is an application program that you start using a transaction code. It may contain one or more SAP LUWs.

The following sections of this documentation explain these three terms in more detail.

Database Logical Unit of Work (LUW)

From the point of view of database programming, a database LUW is an inseparable sequence of database operations that ends with a database commit. The database LUW is either fully executed by the database system or not at all. Once a database LUW has been successfully executed, the database will be in a consistent state. If an error occurs within a database LUW, all of the database changes since the beginning of the database LUW are reversed. This leaves the database in the state it had before the transaction started.



The database changes that occur within a database LUW are not actually written to the database until after the database commit. Until this happens, you can use a database rollback to reverse the changes. In the R/3 System, database commits and rollbacks can be triggered either implicitly or using explicit commands.

Implicit Database Commits in the R/3 System

A work process can only execute a single database LUW. The consequence of this is that a work process must always end a database LUW when it finishes its work for a user or an external call. There are four cases in which work processes trigger an implicit database commit:

- When a dialog step is completed

  Control changes from the work process back to the SAPgui.

- When a function module is called in another work process (RFC).

  Control passes to the other work process.

- When the called function module (RFC) in the other work process ends.

  Control returns to the calling work process.

- Error dialogs (information, warning, or error messages) in dialog steps.

Control passes from the work process to the SAPgui.

Explicit Database Commits in the R/3 System

There are two ways to trigger an explicit database commit in your application programs:

- Call the function module DB_COMMIT

  The sole task of this function module is to start a database commit.

- Use the ABAP statement COMMIT WORK

  This statement starts a database commit, but also performs other tasks (refer to the keyword documentation for COMMIT WORK).

Implicit Database Rollbacks in the R/3 System

The following cases lead to an implicit database rollback:

- Runtime error in an application program

  This occurs whenever an application program has to terminate because of an unforeseen situation (for example, trying to divide by zero).

- Termination message

  Termination messages are generated using the ABAP statement MESSAGE with the message type A or X. In certain cases (updates), they are also generated with message types I, W, and E. These messages end the current application program.

Explicit Database Rollbacks in the R/3 System

You can trigger a database rollback explicitly using the ABAP statement ROLLBACK WORK. This statement starts a database rollback, but also performs other tasks (refer to the keyword documentation for COMMIT WORK).

From the above, we can draw up the following list of points at which database LUWs begin and end.

A Database LUW Begins

- Each time a dialog step starts (when the dialog step is sent to the work process).
- Whenever the previous database LUW ends in a database commit.
- Whenever the previous database LUW ends in a database rollback.

A Database LUW Ends

- Each time a database commit occurs. This writes all of the changes to the database.
- Each time a database rollback occurs. This reverses all of the changes made during the LUW.

Database LUWs and Database Locks

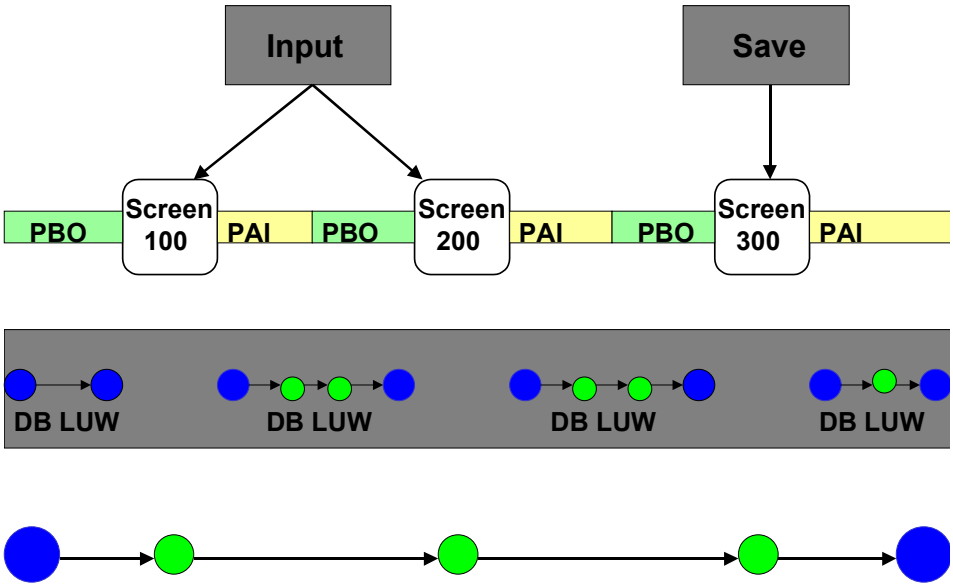
As well as the database changes made within it, a database LUW also consists of database locks. The database system uses locks to ensure that two or more users cannot change the same data simultaneously, since this could lead to inconsistent data being written to the

database. A database lock can only be active for the duration of a database LUW. They are automatically released when the database LUW ends. In order to program SAP LUWs, we need a lock mechanism within the R/3 System that allows us to create locks with a longer lifetime (refer to The R/3 Locking Concept [Page 1270].)

SAP LUW

Die Verbuchungsverwaltung [Ext.]

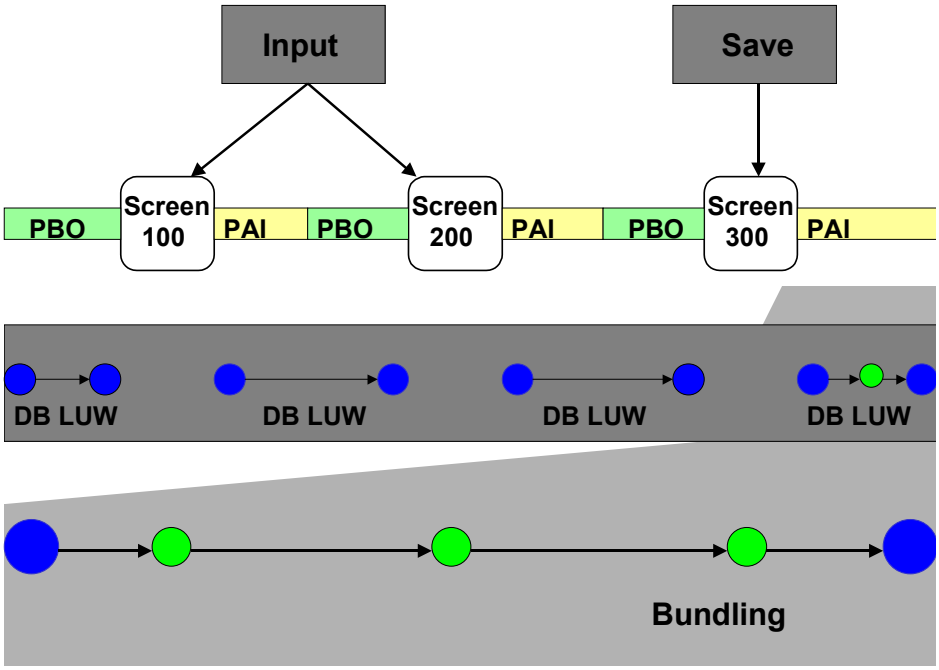
The Open SQL statements INSERT, UPDATE, MODIFY, and DELETE allow you to program database changes that extend over several dialog steps. Even if you have not explicitly programmed a database commit, the implicit database commit that occurs after a screen has been processed concludes the database LUW. The following diagram shows the individual database LUWs in a typical screen sequence:



Under this procedure, you cannot roll back the database changes from previous dialog steps. It is therefore only suitable for programs in which there is no logical relationship between the individual dialog steps.

However, the database changes in individual dialog steps normally depend on those in other dialog steps, and must therefore all be executed or rolled back together. These dependent database changes form logical units, and can be grouped into a single database LUW using the bundling techniques listed below.

A logical unit consisting of dialog steps, whose changes are written to the database in a single database LUW is called an SAP LUW. Unlike a database LUW, an SAP LUW can span several dialog steps, and be executed using a series of different work processes. If an SAP LUW contains database changes, you should either write all of them or none at all to the database. To ensure that this happens, you must include a database commit when your transaction has ended successfully, and a database rollback in case the program detects an error. However, since database changes from a database LUW cannot be reversed in a subsequent database LUW, you must make all of the database changes for the SAP LUW in a single database LUW. To maintain data integrity, you must bundle all of you database changes in the final database LUW of the SAP LUW. The following diagram illustrates this principle:



The bundling technique for database changes within an SAP LUW ensures that you can still reverse them. It also means that you can distribute a transaction across more than one work process, and even across more than one R/3 System. The possibilities for bundling database changes within an SAP LUW are listed below:

The simplest form of bundling would be to process a whole application within a single dialog step. Here, the system checks the user's input and updates the database without a database commit occurring within the dialog step itself. Of course, this is not suitable for complex business processes. Instead, the R/3 Basis system contains the following bundling techniques.

Bundling using Function Modules for Updates

If you call a function module using the CALL FUNCTION... IN UPDATE TASK statement, the function module is flagged for execution using a special update work process. This means that you can write the Open SQL statements for the database changes in the function module instead of in your program, and call the function module at the point in the program where you would otherwise have included the statements. When you call a function module using the IN UPDATE TASK addition, it and its interface parameters are stored as a log entry in a special database table called VBLOG.

The function module is executed using an update work process when the program reaches the COMMIT WORK statement. After the COMMIT WORK statement, the dialog work process is free to receive further user input. The dialog part of the transaction finishes with the COMMIT WORK statement. The update part of the SAP LUW then begins, and this is the responsibility of the update work process. The SAP LUW is complete once the update process has committed or rolled back all of the database changes.

For further information about how to create function modules for use in update, refer to Creating Function Modules for Database Updates [Page 1282]

During the update, errors only occur in exceptional cases, since the system checks for all logical errors, such as incorrect entries, in the dialog phase of the SAP LUW. If a logical error occurs, the program can terminate the update using the ROLLBACK WORK statement. Then, the function modules are not called, and the log entry is deleted from table VBLOG. Errors during the update itself are usually technical, for example, memory shortage. If a technical error occurs, the update work process triggers a database rollback, and places the log entry back into VBLOG. It then sends a mail to the user whose dialog originally generated the VBLOG entry with details of the termination. These errors must be corrected by the system administrator. After this, the returned VBLOG entries can be processed again.

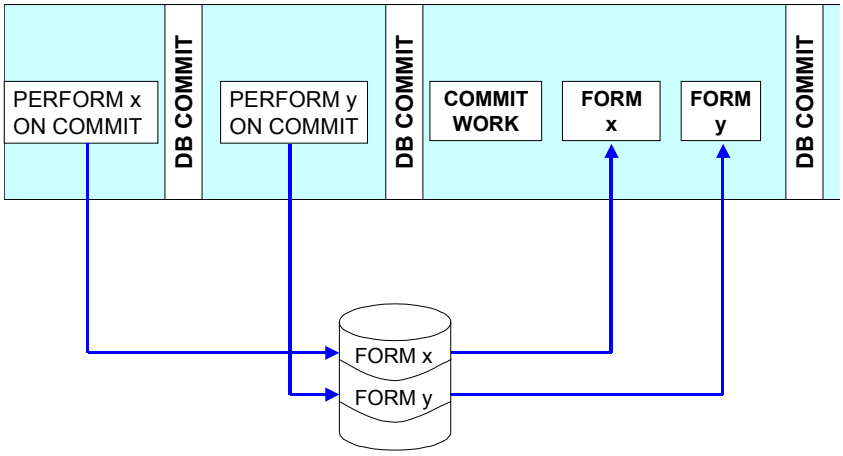
For further information about update administration, see Update Administration

This technique of bundling database changes in the last database LUW of the SAP LUW allows you to update the database asynchronously, reducing the response times in the dialog work process. You can, for example, decouple the update entirely from the dialog work process and use a central update work process on a remote database server.

Bundling Using Subroutines

The statement PERFORM ON COMMIT calls a subroutine in the dialog work process. However, it is not executed until the system reaches the next COMMIT WORK statement. Here, as well, the ABAP statement COMMIT WORK defines the end of the SAP LUW, since all statements in a subroutine called with PERFORM ON COMMIT that make database changes are executed in the database LUW of the corresponding dialog step.

Update in Dialog Work Process



The advantage of this bundling technique against CALL FUNCTION... IN UPDATE TASK is better performance, since the update data does not have to be written into an extra table. The disadvantage, however, is that you cannot pass parameters in a PERFORM... ON COMMIT statement. Data is passed using global variables and ABAP memory. There is a considerable danger of data inconsistency when you use this method to pass data.

SAP LUW

Bundling Using Function Modules in Other R/3 Systems

Function modules that you call using CALL FUNCTION... IN BACKGROUND TASK DESTINATION... are registered for background execution in another R/3 System when the program reaches the next COMMIT WORK statement (using Remote Function Call). After the COMMIT WORK, the dialog process does not wait for these function modules to be executed (asynchronous update). All of the function modules that you register in this way are executed together in a single database LUW. These updates are useful, for example, when you need to maintain identical data in more than one database.

For further details, refer to the keyword documentation.

For more details of RFC processing, refer to the *Remote Communications* section of the *Basis Services* documentation.

SAP Transactions

An SAP LUW is a logical unit consisting of dialog steps, whose changes are written to the database in a single database LUW. In an application program, you end an SAP LUW with either the COMMIT WORK or ROLLBACK WORK statement. An SAP transaction is an application program that you start using a transaction code. It may contain one or more SAP LUWs. Whenever the system reaches a COMMIT WORK or ROLLBACK WORK statement that is not at the end of the last dialog step of the SAP transaction, it opens a new SAP LUW.

If a particular application requires you to write a complex transaction, it can often be useful to arrange logical processes within the SAP transaction into a sequence of individual SAP LUWs. You can structure SAP transactions as follows:

- With one or more SAP LUWs.

  Transactions in this form consist entirely of processing blocks (dialog modules, event blocks, function module calls, and subroutines). You should be careful to ensure that external subroutines or function modules do not lead to COMMIT WORK or ROLLBACK WORK statements accidentally being executed.

- By inserting an SAP LUW

  The ABAP statements CALL TRANSACTION (start a new transaction), SUBMIT (start an executable program), and CALL FUNCTION... DESTINATION (call a function module using RFC) open a new SAP LUW. When you call a program, it always opens its own SAP LUW. However, it does not end the LUW of the SAP transaction that called it. This means that a COMMIT WORK or ROLLBACK WORK statement only applies to the SAP LUW of the called program. When the new LUW is complete, the system carries on processing the first SAP LUW.

- By running two SAP LUWs in parallel

  The CALL FUNCTION... STARTING NEW TASK statement calls a function module asynchronously in a new session. Unlike normal function module calls, the calling transaction carries on with its own processing as soon as the function module has started, and does not wait for it to finish processing. The function call is asynchronous. The called function module can now call its own screens and interact with the user.

The R/3 Lock Concept

Das R/3 Sperrkonzept [Ext.]

Reasons for Setting Locks

Suppose a travel agent want to book a flight. The customer wants to fly to a particular city with a certain airline on a certain day. The booking must only be possible if there are still free places on the flight. To avoid the possibility of overbooking, the database entry corresponding to the flight must be locked against access from other transactions. This ensures that one user can find out the number of free places, make the booking, and change the number of free places without the data being changed in the meantime by another transaction.

Lock Mechanisms in the Database System

The database system automatically sets database locks when it receives change statements (INSERT, UPDATE, MODIFY, DELETE) from a program. Database locks are physical locks on the database entries affected by these statements. You can only set a lock for an existing database entry, since the lock mechanism uses a lock flag in the entry. These flags are automatically deleted in each database commit. This means that database locks can never be set for longer than a single database LUW; in other words, a single dialog step in an R/3 application program.

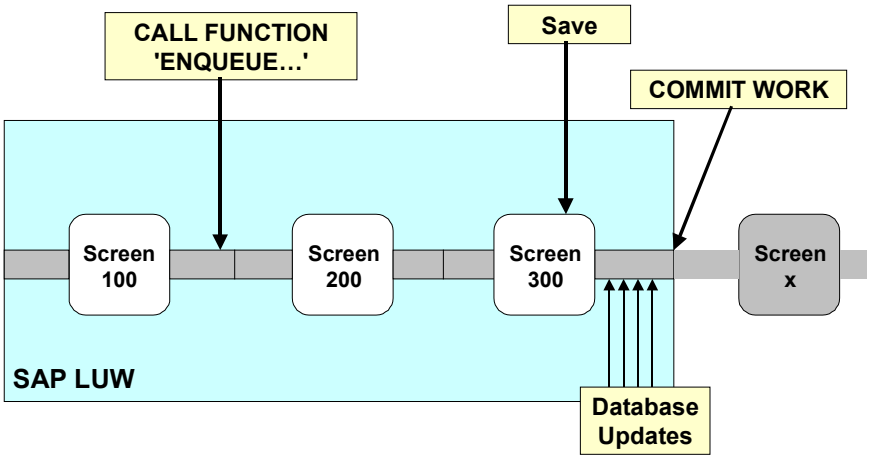
Physical locks in the database system are therefore insufficient for the requirements of an R/3 transaction. Locks in the R/3 System must remain set for the duration of a whole SAP LUW, that is, over several dialog steps. They must also be capable of being handled by different work processes and even different application servers. Consequently, each lock must apply on all servers in that R/3 System.

SAP Locks

To complement the SAP LUW concept, in which bundled database changes are made in a single database LUW, the R/3 System also contains a lock mechanism, fully independent of database locks, that allows you to set a lock that spans several dialog steps. These locks are known as **SAP locks**.

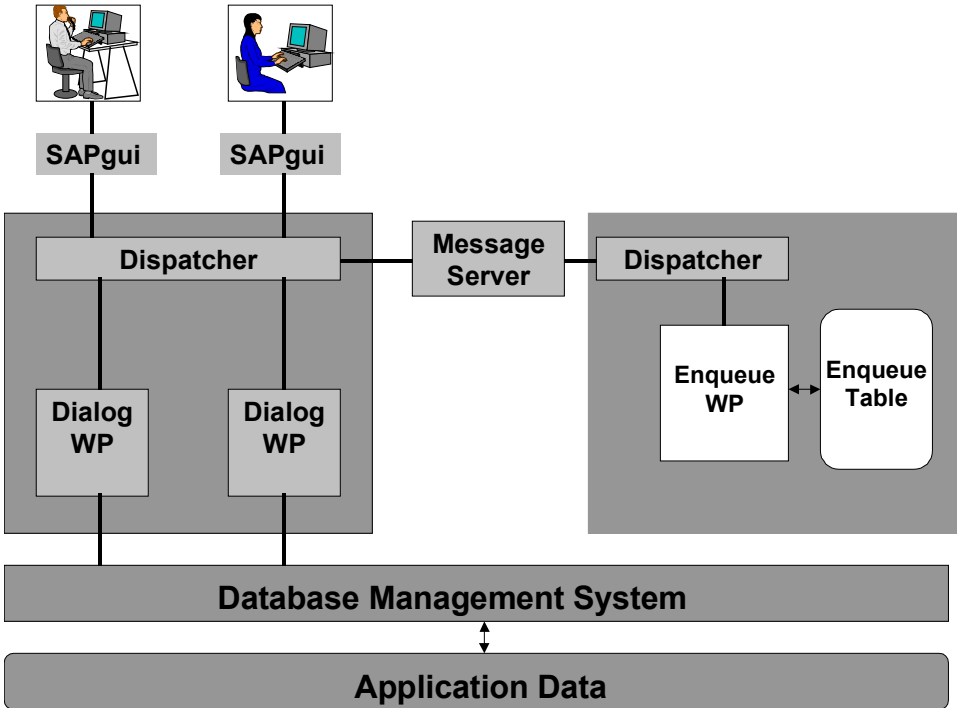
The SAP lock concept is based on **lock objects**. Lock objects allow you to set an SAP lock for an entire application object. An application object consists of one or more entries in a database table, or entries from more than one database table that are linked using foreign key relationships.

Before you can set an SAP lock in an ABAP program, you must first create a lock object in the ABAP Dictionary. A lock object definition contains the database tables and their key fields on the basis of which you want to set a lock. When you create a lock object, the system automatically generates two function modules with the names `ENQUEUE_<lock object name>` and `DEQUEUE_<lock object name>`. You can then set and release SAP locks in your ABAP program by calling these function modules in a CALL FUNCTION statement.



See also: Example Transaction: SAP Locking [Page 1274].

These function modules are executed in a special enqueue work process. Within an R/3 System, enqueue work processes run on a single application server. This server maintains a central lock table for the entire R/3 System in its shared memory.



The enqueue function module sets an SAP lock by writing entries in the central lock table. If the lock cannot be set because the application object (or a part of it) is already locked, this is

reflected in the return code sy-subrc. The following diagram shows the components of the R/3 System that are involved in setting a lock.

Unlike the database, which sets physical locks, the SAP lock mechanism sets logical locks. This means that

- A locked database entry is not physically locked in the database table.

  The lock entry is merely entered as a lock argument in the central R/3 lock table. The lock argument is made up of the primary key field values for the tables in the lock object. These are import parameters of the enqueue function module. The lock is independent of database LUWs. It is released either implicitly when the database update or the SAP transaction ends, or explicitly, using the corresponding dequeue function module. You can use a special parameter in the update function module to set the exact point at which the lock is released during the database update.

- A locked entry does not necessarily have to exist in a database table.

  You can, for example, set a lock as a precaution for a database entry that is not written to the database until the update at the end of the SAP LUW.

- The effectiveness of the locks depends on cooperative application programming.

  Since there are no physical locks in the database tables themselves, all programs that use the same application objects must look in the central table themselves for any locks. There is no mechanism that automatically prevents a program from ignoring the locks in the lock table.

Lock Types

There are two types of lock in the R/3 System:

- Shared lock

  Shared locks (or read locks) allow you to prevent data from being changed while you are reading it. They prevent other programs from setting an exclusive lock (write lock) to change the object. It does not, however, prevent other programs from setting further read locks.

- Exclusive lock

  Exclusive locks (or write locks) allow you to prevent data from being changed while you are changing it yourself. An exclusive lock, as its name suggests, locks an application object for exclusive use by the program that sets it. No other program can then set either a shared lock or an exclusive lock for the same application object.

Lock Duration

When you set a lock, you should bear in mind that if it remains set for a long time, the availability of the object to other transactions is reduced. Whether or not this is acceptable depends on the nature of the task your program is performing.

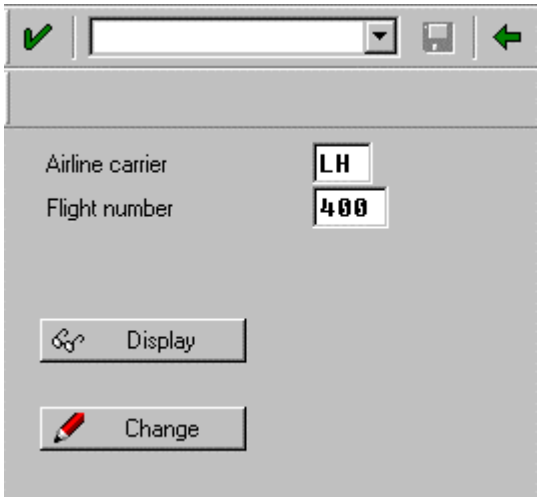
Remember in particular that setting too many shared locks without good reason can have a considerable effect on programs that work with database tables. If several programs running concurrently all set a shared lock for the same application object in the system, it can make it almost impossible to set an exclusive lock, since the program that needs to set that lock will be unable to find any time when there are no locks at all set for that object. Conversely, a single exclusive lock prevents all other programs from reading the locked object.

At the end of an SAP LUW, you should release all locks. This either happens automatically during the database update, or explicitly, when you call the corresponding dequeue function module. Locks that are not linked to a database update are released at the end of the SAP transaction.

Example Transaction: SAP Locking

Übernahme [Ext.]

The following example transaction shows how you can lock and unlock database entries using a lock object. It lets the user request a given flight (on screen 100) and display or update it (on screen 200). If the user chooses *Change*, the table entry is locked; if he or she chooses *Display*, it is not.



The example uses the lock object ESFLIGHT and its function modules ENQUEUE_ESFLIGHT and DEQUEUE_ESFLIGHT to lock and unlock the object.

For more information about creating lock objects and the corresponding function modules, refer to the *Lock objects* section of the *ABAP Dictionary* documentation.

The PAI processing for screen 100 in this transaction processes the user input and prepares for the requested action (*Change* or *Display*). If the user chooses *Change*, the program locks the relevant database object by calling the corresponding ENQUEUE function.

```
MODULE USER_COMMAND_0100 INPUT.
  CASE OK_CODE.
    WHEN 'SHOW'....
    WHEN 'CHNG'.
*   <...Authority-check and other code...>
      CALL FUNCTION 'ENQUEUE_ESFLIGHT'
      EXPORTING
        MANDT     = SY-MANDT
        CARRID     = SPFLI-CARRID
        CONNID     = SPFLI-CONNID
      EXCEPTIONS
        FOREIGN_LOCK  = 1
        SYSTEM_FAILURE = 2
        OTHERS     = 3.
   IF SY-SUBRC NE 0.
     MESSAGE ID   SY-MSGID
         TYPE  'E'
         NUMBER SY-MSGNO
         WITH  SY-MSGV1 SY-MSGV2 SY-MSGV3 SY-MSGV4.
```

```
      ENDIF.
     ...
```

The ENQUEUE function module can trigger the following exceptions:

FOREIGN_LOCK determines whether a conflicting lock already exists. The system variable SY-MSGV1 contains the name of the user that owns the lock.

The SYSTEM_FAILURE exception is triggered if the enqueue server is unable to set the lock for technical reasons.

At the end of a transaction, the locks are released automatically. However, there are exceptions if you have called update routines within the transaction. You can release a lock explicitly by calling the corresponding DEQUEUE module. As the programmer, you must decide for yourself the point at which it makes most sense to release the locks (for example, to make the data available to other transactions).

If you need to use the DEQUEUE function module call several times in a program, it makes good sense to write it in a subroutine, which you then call as required.

The subroutine UNLOCK_FLIGHT calls the DEQUEUE function module for the lock object ESFLIGHT:

```
FORM UNLOCK_FLIGHT.
     CALL FUNCTION 'DEQUEUE_ESFLIGHT'
          EXPORTING
              MANDT     = SY-MANDT
              CARRID    = SPFLI-CARRID
              CONNID    = SPFLI-CONNID
          EXCEPTIONS
              OTHERS    = 1.
     SET SCREEN 100.
ENDFORM.
```

You might use this for the BACK and EXIT functions in a PAI module for screen 200 in this example transaction. In the program, the system checks whether the user leaves the screen without having saved his or her changes. If so, the PROMPT_AND_SAVE routine sends a reminder, and gives the user the opportunity to save the changes. The flight can be unlocked by calling the UNLOCK_FLIGHT subroutine.

```
MODULE USER_COMMAND_0200 INPUT.
  CASE OK_CODE.
    WHEN 'SAVE'....
    WHEN 'EXIT'.
      CLEAR OK_CODE.
      IF OLD_SPFLI NE SPFLI.
         PERFORM PROMPT_AND_SAVE.
      ENDIF.
      PERFORM UNLOCK_FLIGHT.
      LEAVE TO SCREEN 0.
    WHEN 'BACK'....
```

Update Techniques

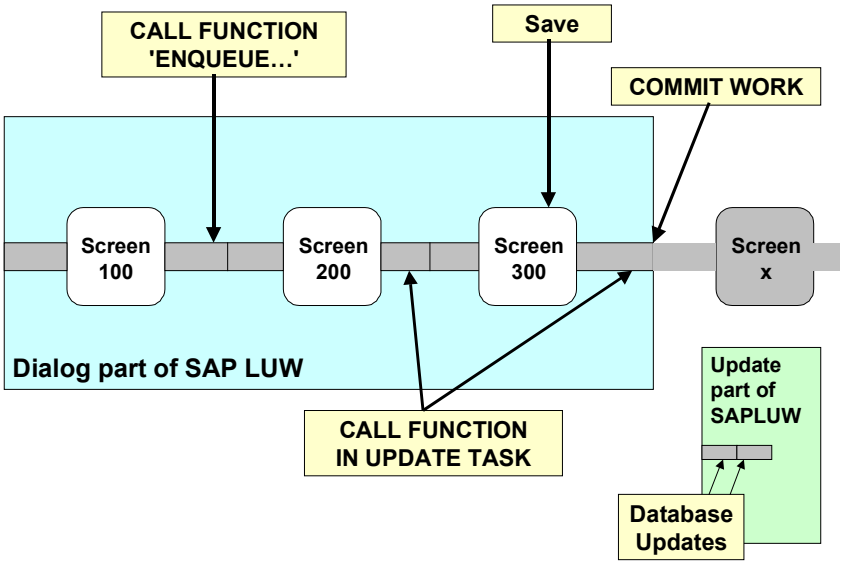
The main update technique for bundling database changes in a single database LUW is to use CALL FUNCTION... IN UPDATE TASK. This section describes various ways of updating the database.

A program can send an update request using COMMIT WORK

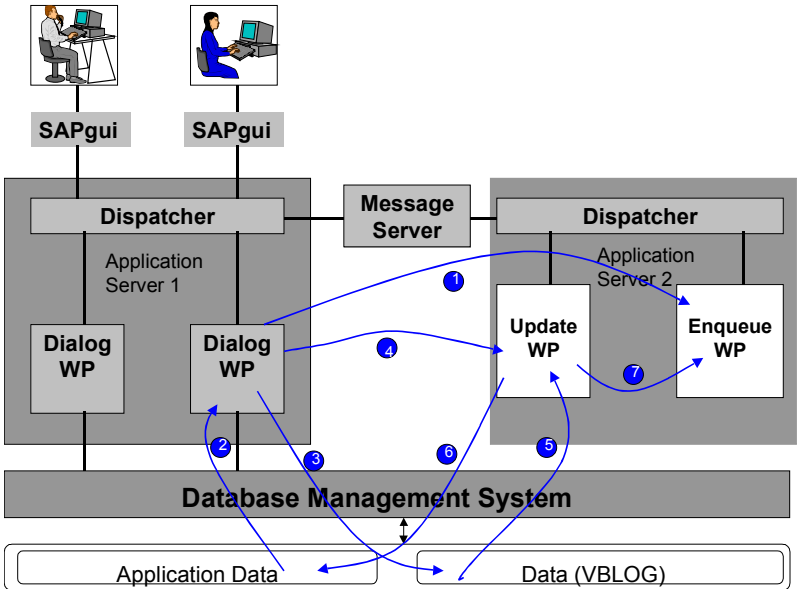
- To the update work process, where it is processed asynchronously. The program does not wait for the work process to finish the update (Asynchronous Update [Page 1277]).
- For asynchronous processing in two steps (Updating Asynchronously in Steps [Page 1279].)
- To the update work process, where it is processed synchronously. The program waits for the work process to finish the update (Synchronous Update [Page 1280]).
- To its own work process locally. In this case, of course, the program has to wait until the update is finished (Local Update [Page 1281].)

Asynchronous Update

A typical R/3 installation contains dialog work processes and at least one update work process. The update work processes are responsible for updating the database. When an ABAP program reaches a COMMIT WORK statement, any function modules from CALL FUNCTION... IN UPDATE TASK statements are released for processing in an update work process. The dialog process does not wait for the update to finish. This kind of update is called asynchronous update.



The following diagram shows a typical asynchronous update:



Asynchronous Update

For example, suppose a user wants to change an entry in a database table, or add a new one. He or she enters the necessary data, and then starts the update process by choosing *Save*. This starts the following procedure in the ABAP program:

1. Firstly, the program locks the database entry against other users, using the enqueue work process (or the message server in the case of a distributed system). This generates an entry in the lock table. The user is informed whether the update was successful, or whether the lock could not be set because of other users.
2. If the lock is set, the program reads the entry that is to be changed and modifies it. If the user has created a new entry, the program checks whether a record with the same key values already exists.
3. In the current dialog work process, the program calls a function module using CALL FUNCTION... IN UPDATE TASK, and this writes the change details as an entry in table VBLOG.
4. When the program is finished (maybe after further dialog steps), a COMMIT WORK statement starts the final part of the SAP LUW. The work process that is processing the current dialog step starts an update work process.
5. Based on the information passed to it from the dialog work process, the update work process reads the log entries belonging to the SAP LUW from table VBLOG.
6. The update work process passes this data to the database for updating, and analyzes the return message from the database. If the update was successful, the update work process triggers a database commit after the last database change and deletes the log entries from table VBLOG. If an error occurred, the update work process triggers a database rollback, leaves the log entries in table VBLOG, flags them as containing errors, and sends a SAPoffice message to the user, who should then inform the system administrator.
7. The corresponding entries in the lock table are reset by the update work process.

Asynchronous update is useful when response time from the transaction is critical, and the database updates themselves are so complex that they justify the extra system load of logging them in VBLOG. If you are running a transaction in a background work process, asynchronous update offers no advantages.

Updating Asynchronously in Steps

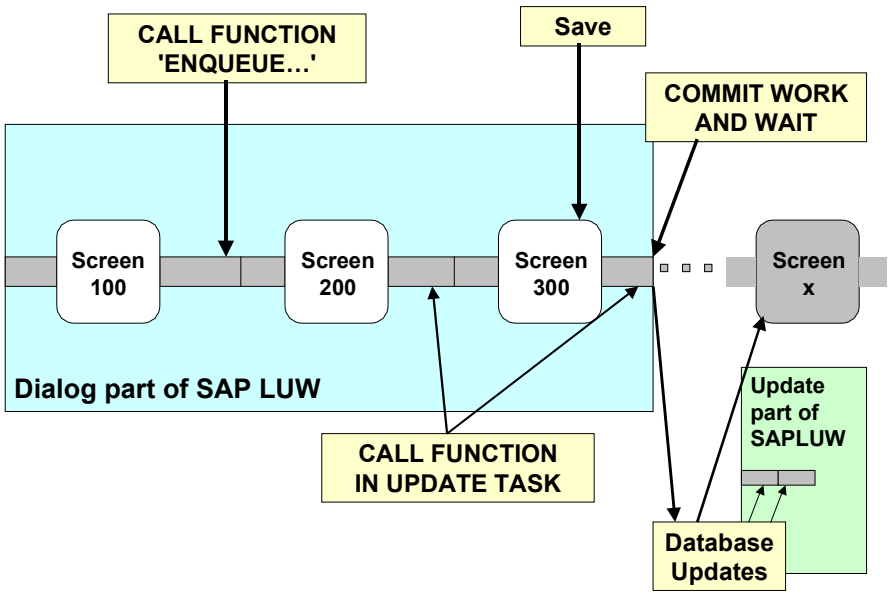
When you process a VBLOG entry asynchronously, you can do it in two update steps. This allows you to divide the contents of the update into primary and secondary steps. The primary step is called V1, the secondary step V2. The V1 and V2 steps of a log entry are processed in separate database LUWs. The entries in the lock table are usually deleted once the V1 step has been processed. There is no locking for the V2 step. Dividing up the update process allows you to separate time-critical updates that require database locks from less critical data updates that do not need locks. V2 steps receive lower priority from the dispatcher than V1 steps. This ensures that the time- and lock-critical updates are processed quickly, even when the system is busy.

If an error occurs during the V1 processing, the database rollback applies to all V1 steps in the log entry. The entire entry is replaced in table VBLOG. If an error occurs during V2 processing, all of the V2 steps in the log entry are replaced in table VBLOG, but the V1 updates are not reversed.

The system marks rolled-back function modules as error functions in the update task log. The error can then be corrected and the function restarted later. To access the update task log, choose *Tools* → *Administration* → *Monitoring* → *Update*. For further information about update administration, see the *Managing Updating* section of the *BC System Services* documentation.

Synchronous Update

In synchronous update, you do not submit an update request using CALL FUNCTION... IN UPDATE TASK. Instead, you use the ABAP statement COMMIT WORK AND WAIT. When the update is finished, control passes back to the program. Synchronous update works in the same way as bundling update requests in a subroutine (PERFORM ON COMMIT). This kind of update is useful when you want to use both asynchronous and synchronous processing without having to program the bundles in two separate ways. The following diagram illustrates the synchronous update process:



Use this type of update whenever the changed data is required immediately. For example, you may want to link SAP LUWs together where one LUW depends on the results of the previous one.

Local Update

In a local update, the update program is run by the same work process that processed the request. The dialog user has to wait for the update to finish before entering further data. This kind of update is useful when you want to reduce the amount of access to the database. The disadvantage of local updates is their parallel nature. The updates can be processed by many different work processes, unlike asynchronous or synchronous update, where the update is serialized due to the fact that there are fewer update work processes (and maybe only one).

You switch to local update using the ABAP statement SET UPDATE TASK LOCAL. This statement sets a “local update switch”. When it is set, the system interprets CALL FUNCTION IN UPDATE TASK as a request for local update. The update is processed in the same work process as the dialog step containing the COMMIT WORK. The transaction waits for the update to finish before continuing.

As an example, suppose you have a program that uses asynchronous update that you normally run in dialog mode. However, this time you want to run it in the background. Since the system response time is irrelevant when you are running the program in the background, and you only want the program to continue processing when the update has actually finished, you can set the SET UPDATE TASK LOCAL switch in the program. You can then use a system variable to check at runtime whether the program is currently running in the background.

By default, the local update switch is not set, and it is reset after each COMMIT WORK or ROLLBACK WORK. You therefore need to include a SET UPDATE TASK LOCAL statement at the beginning of each SAP LUW.

If you reset data within the local update, the ROLLBACK WORK statement applies to both the dialog and the update part of the transaction, since no new SAP LUW is started for the update.

Creating Update Function Modules

To create a function module, you first need to start the Function Builder. Choose *Tools* → *ABAP Workbench*, *Function Builder*. For more information about creating function modules, refer to the ABAP Workbench Tools [Ext.] documentation.

To be able to call a function module in an update work process, you must flag it in the Function Builder. When you create the function module, set the *Process Type* attribute to one of the following values:

- *Update with immediate start*

  Set this option for high priority (“V1”) functions that run in a shared (SAP LUW). These functions can be restarted by the update task in case of errors.

- *Update w. imm. start, no restart*

  Set this option for high priority (“V1”) functions that run in a shared (SAP LUW). These functions may not be restarted by the update task.

- *Update with delayed start*

  Set this option for low priority (“V2”) functions that run in their own update transactions. These functions can be restarted by the update task in case of errors.

To display the attributes screen in the Function Builder, choose *Goto* → *Administration*.

Defining the Interface

Function modules that run in the update task have a limited interface:

- Result parameters or exceptions are not allowed since update-task function modules cannot report on their results.

- You must specify input parameters and tables with reference fields or reference structures defined in the ABAP Dictionary.

Calling Update Functions

Synchronous or Asynchronous Processing?

Function modules that run in the update task can run synchronously or asynchronously. You determine this by the form of the commit statement you use:

- COMMIT WORK

  This is the standard form, which specifies asynchronous processing. Your program does not wait for the requested functions to finish processing.

- COMMIT WORK AND WAIT

  This form specifies synchronous processing. The commit statement waits for the requested functions to finish processing. Control returns to your program after all high priority (V1) function modules have run successfully.

  The AND WAIT form is convenient for switching old programs to synchronous processing without having to re-write the code. Functionally, using AND WAIT for update-task updates is just the same as dialog-task updates with PERFORM ON COMMIT.

Parameter Values at Execution

In ABAP, you can call update-task function modules in two different ways. The way you choose determines what parameter values are used when the function module is actually executed. Parameter values can be set either at the time of the CALL FUNCTION statement, or at the time of the COMMIT WORK. The following sections explain.

Calling Update Functions Directly [Page 1284]

Adding Update Task Calls to a Subroutine [Page 1285].

The examples in these sections show asynchronous commits with COMMIT WORK.

Calling Update Functions Directly

To call a function module directly, use CALL FUNCTION IN UPDATE TASK directly in your code.

```
CALL FUNCTION 'FUNCTMOD' IN UPDATE TASK EXPORTING...
```

The system then logs your request and executes the function module when the next COMMIT WORK statement is reached. The parameter values used to execute the function module are those current at the time of the call.

```
a = 1.
CALL FUNCTION 'UPD_FM' IN UPDATE TASK EXPORTING PAR = A...
a = 2.
CALL FUNCTION 'UPD_FM' IN UPDATE TASK EXPORTING PAR = A...
a = 3.
COMMIT WORK.
```

Here, the function module UPD_FM is performed twice in the update task: the first time, with value 1 in PAR, the second time with value 2 in PAR.

Adding Update Task Calls to a Subroutine

You can also put the CALL FUNCTION IN UPDATE TASK into a subroutine and call the subroutine with:

```
PERFORM SUBROUT ON COMMIT.
```

If you choose this method, the subroutine is executed at the commit. Thus the request to run the function in the update task is also logged during commit processing. As a result, the parameter values logged with the request are those current at the time of the commit.

```
a = 1.
PERFORM F ON COMMIT.
a = 2.
PERFORM F ON COMMIT.
a = 3.
COMMIT WORK.

FORM f.
 CALL FUNCTION 'UPD_FM' IN UPDATE TASK EXPORTING PAR = A.
ENDFORM.
```

In this example, the function module UPD_FM is carried out with the value 3 in PAR. The update task executes the function module only once, despite the two PERFORM ON COMMIT statements. This is because a given function module, logged with the same parameter values, can never be executed more than once in the update task. The subroutine itself, containing the function module call, may not have parameters.

The method described here is not suitable for use inside dialog module code. However, if you do need to use a dialog module, refer to Dialog Modules That Call Update Modules.

Special LUW Considerations

In the update task queue, the system identifies all function modules belonging to the same transaction [Page 1261] (SAP LUW) by assigning them a common update key. At the next COMMIT WORK, the update task reads the queue and processes all requests with the predefined update key.

If your program calls an update-task function module, the request to execute the module (or the subroutine calling it) is provided with the update key of the current LUW and placed in the queue.

The following sections explain what happens to LUWs when update function modules are included in other modules (transactions or dialog modules) that are called by other programs.

Transactions That Call Update Task Functions [Page 1287]

Dialog Modules That Call Update Task Functions [Page 1288]

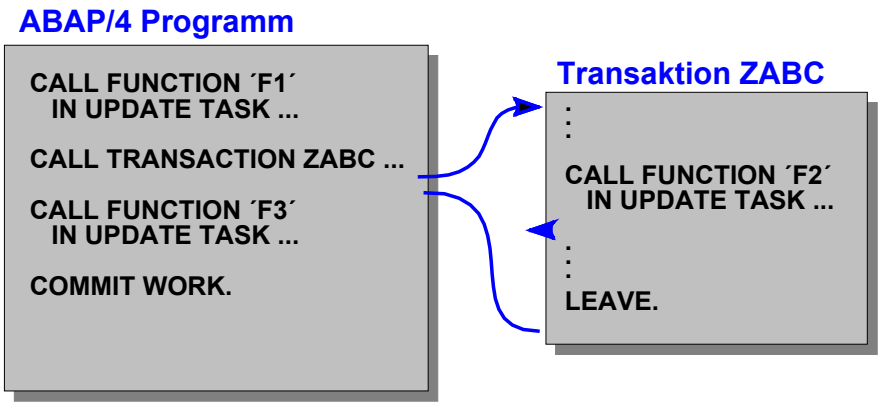
Transactions That Call Update Function Modules

If your program calls another program that itself calls an update function module, you should be aware of the following:

When the new program is called, a new SAP LUW begins, and a new update key is generated. This key is used to identify all update-task operations requested during the called program.

When returning from the program, the LUW of the calling program is restored together with the old update key.

If the called program does not contain its own COMMIT WORK, the database update requests are not processed, and the update function modules are not called. In the following example, F1, F2, and F3 are update function modules:



Here, F1 and F3 are executed in the update task, because the COMMIT WORK for the main program triggers their execution. However, since transaction ZABC contains no COMMIT WORK statement, the function F2 is never executed by the update task.

Dialog Modules that Call Update Function Modules

Unlike transactions and executable programs (reports), dialog modules do not start a new SAP LUW. Calls to update-task function modules from a dialog module use the same update key as the ones in the calling program. The result is that calls to update function modules from a dialog module are executed only if a COMMIT WORK statement occurs in the calling program.

If you place a COMMIT WORK in a dialog module, it does commit changes to the database (for example, with UPDATE).However, it does not start the update task. The function modules are not actually executed until a COMMIT WORK statement occurs in the calling program.

If you use dialog modules, try to avoid including calls to update function modules in subroutines called with PERFORM ON COMMIT. In general, any occurrence of PERFORM ON COMMIT (with or without update-task function calls) in a dialog module can cause problems.

This is because dialog modules have their own roll area, which disappears when the module finishes. Consequently, all local data (including data used for parameter values when calling an update function module) disappears as soon as the commit in the main program is reached.

If you must call an update function module in a subroutine in a dialog module, you must ensure that the values of the actual parameters still exist when the update-task function actually runs. To do this, you can store the required values with EXPORT TO MEMORY and then import them back into the main program (IMPORT FROM MEMORY) before the COMMIT WORK statement.

Error Handling for Bundled Updates

Runtime errors can occur during execution of bundled updates. How are they handled? In general, COMMIT WORK processing occurs in the following order:

1. All dialog-task FORM routines logged with PERFORM ON COMMIT are executed.
2. All high-priority (V1) update-task function modules are executed.

   The end of V1-update processing marks the end of the . If you used COMMIT WORK AND WAIT to trigger commit processing, control returns to the dialog-task program.
3. All low-priority (V2) update-task function modules are triggered.
   All background-task function modules are triggered.

Runtime errors can occur either in the system itself, or because your program issues an termination message (MESSAGE type 'A'). Also, the ROLLBACK WORK statement automatically signals a runtime error. The system handles errors according to where they occur:

- **in a FORM routine** (called with PERFORM ON COMMIT)
  - Updates already executed for the current update transaction are rolled back.
  - No other FORM routines will be started.
  - No further update-task or background-task functions will be started.
  - An error message appears on the screen.
- **in a V1 update-task function module** (requested IN UPDATE TASK)
  - Updates already executed for V1 functions are rolled back.
  - All further update-task requests (V1 or V2) are thrown away.
  - All background-task requests are thrown away.
  - Updates already executed for FORM routines called with PERFORM ON COMMIT are **not** rolled back.
  - An error message appears on the screen, if your system is set up to send them
- **in a V2 update-task function module** (requested IN UPDATE TASK)
  - Updates already executed for the current V2 function are rolled back.
  - All update-task requests (V2) still to be executed are carried out.
  - All background-task requests still to be executed are carried out.
  - No updates for previously executed V1 or V2 function are rolled back.
  - No updates previously executed for FORM routines (called with ON COMMIT) are rolled back.
  - An error message appears on the screen, if your system is set up to send them
- **in a background-task function module** (requested IN BACKGROUND TASK DESTINATION)
  - Background-task updates already executed for the current DESTINATION are not rolled back.

Error Handling for Bundled Updates

- All further background-task requests for the same DESTINATION are thrown away.
- No other previously-executed updates are not rolled back.
- No error message appears on the screen.

  If your program detects that an error in remote processing has occurred, it can decide whether to resubmit the requests at a later time.

  For further information about RFC processing, refer to the Remote Communications [Ext.] documentation.

ABAP Objects

General

Object Orientation in ABAP

Using ABAP Objects

What is Object Orientation?

Object orientation (OO), or to be more precise, object-oriented programming, is a problem-solving method in which the software solution reflects objects in the real world.

A comprehensive introduction to object orientation as a whole would go far beyond the limits of this introduction to ABAP Objects. This documentation introduces a selection of terms that are used universally in object orientation and also occur in ABAP Objects. In subsequent sections, it goes on to discuss in more detail how these terms are used in ABAP Objects. The end of this section contains a list of **further reading**, with a selection of titles about object orientation.

Objects

An object is a section of source code that contains data and provides services. The data forms the attributes of the object. The services are known as methods (also known as operations or functions). Typically, methods operate on private data (the attributes, or state of the object), which is only visible to the methods of the object. Thus the attributes of an object cannot be changed directly by the user, but only by the methods of the object. This guarantees the internal consistency of the object.

Classes

Classes describe objects. From a technical point of view, objects are runtime instances of a class. In theory, you can create any number of objects based on a single class. Each instance (object) of a class has a unique identity and its own set of values for its attributes.

Object References

In a program, you identify and address objects using unique object references. Object references allow you to access the attributes and methods of an object.

In object-oriented programming, objects usually have the following properties:

Encapsulation

Objects restrict the visibility of their resources (attributes and methods) to other users. Every object has an **interface**, which determines how other objects can interact with it. The implementation of the object is encapsulated, that is, invisible outside the object itself.

Polymorphism

Identical (identically-named) methods behave differently in different classes. Object-oriented programming contains constructions called interfaces. They enable you to address methods with the same name in different objects. Although the form of address is always the same, the implementation of the method is specific to a particular class.

Inheritance

You can use an existing class to derive a new class. Derived classes inherit the data and methods of the superclass. However, they can overwrite existing methods, and also add new ones.

Uses of Object Orientation

Below are some of the advantages of object-oriented programming:

- Complex software systems become easier to understand, since object-oriented structuring provides a closer representation of reality than other programming techniques.
- In a well-designed object-oriented system, it should be possible to implement changes at class level, without having to make alterations at other points in the system. This reduces the overall amount of maintenance required.
- Through polymorphism and inheritance, object-oriented programming allows you to reuse individual components.
- In an object-oriented system, the amount of work involved in revising and maintaining the system is reduced, since many problems can be detected and corrected in the design phase.

Achieving these goals requires:

- Object-oriented programming languages

  Object-oriented programming techniques do not necessarily depend on object-oriented programming languages. However, the efficiency of object-oriented programming depends directly on how object-oriented language techniques are implemented in the system kernel.

- Object-oriented tools

  Object-oriented tools allow you to create object-oriented programs in object-oriented languages. They allow you to model and store development objects and the relationships between them.

- Object-oriented modeling

  The object-orientation modeling of a software system is the most important, most time-consuming, and most difficult requirement for attaining the above goals. Object-oriented design involves more than just object-oriented programming, and provides logical advantages that are independent of the actual implementation.

This section of the ABAP User's Guide provides an overview of the object-oriented extension of the ABAP language. We have used simple examples to demonstrate how to use the new features. However, these are not intended to be a model for object-oriented design. More detailed information about each of the ABAP Objects statements is contained in the keyword documentation in the ABAP Editor. For a comprehensive introduction to object-oriented software development, you should read one or more of the titles listed below.

Further Reading

There are many books about object orientation, object-oriented programming languages, object-oriented analysis and design, project management for OO projects, patterns and frameworks, and so on. This is a small selection of good books covering the most important topics:

- **Scott Ambler, The Object Primer, SIGS Books & Multimedia (1996), ISBN: 1884842178**

  A very good introduction to object orientation for programmers. It provides comprehensive explanations of all essential OO concepts, and contains a procedure model for learning OO quickly and thoroughly. It is easy to read and practical, but still theoretically-founded.

What is Object Orientation?

- **Grady Booch, Object Solutions: Managing the Object-Oriented Project, Addison-Wesley Pub Co (1995), ISBN: 0805305947**

  A good book about all of the non-technical aspects of OO that are equally important for effective object-oriented programming. Easy to read and full of practical tips.

- **Martin Fowler, UML Distilled: Applying the Standard Object Modeling Language, Addison-Wesley Pub Co (1997), ISBN: 0201325632**

  An excellent book about UML (Unified Modeling Language - the new standardized OO language and notation for modeling). Assumes knowledge and experience of object orientation.

- **Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley Pub Co (1998), ISBN: 0201634988**

  Provides a pattern, showing how recurring design problems can be solved using objects. This is the first big pattern book, containing many examples of good OO design.

- **James Rumbaugh, OMT Insights: Perspectives on Modeling from the Journal of Object-Oriented Programming, Prentice Hall (1996), ISBN: 0138469652**

  A collection of articles addressing the many questions and problems of OO analysis and design, implementation, dependency management, and so on. Highly recommended.

Notes

If you are new to object-orientation, you should read Scott Ambler's 'The Object Primer' and then acquire some practical experience of your own. You should definitely use the CRC techniques described by Ambler and Fowler for object-oriented analysis and design. After this, you should learn UML, since this is the universal OO analysis and design notation. Finally, you should read at least one book about patterns.

At the beginning of a large OO project, the question immediately arises as to the sequence in which things should be done, which phases should be finished at what time, how to divide up and organize the development work, how to minimize risks, how to assemble a good team, and so on and so forth. Many of the best practices of project management have had to be redefined for the object-oriented world, and the opportunities that this has produced are significant. For further information about how to use OO in project management, see Grady Brooch's book 'Object solutions', or the chapter entitles 'An outline development process' from Martin Fowler's book.

There are, of course, many other good books about object orientation. The above list does not claim either to be complete, or necessarily to recommend the best books available.

What are ABAP Objects?

ABAP Objects is a new concept in R/3 Release 4.0. The term has two meanings. On the one hand, it stands for the entire ABAP runtime environment. On the other hand, it represents the object-oriented extension of the ABAP language.

The Runtime Environment

The new name ABAP Objects for the entire ABAP runtime environment is an indication of the way in which SAP has, for some time, been moving towards object orientation, and of its commitment to pursuing this line further. The ABAP Workbench allows you to create R/3 Repository objects such as programs, authorization objects, lock objects, Customizing objects, and so on. Using function modules, you can encapsulate functions in separate programs with a defined interface. The Business Object Repository (BOR) allows you to create SAP Business Objects for internal and external use (DCOM/CORBA). Until now, object-oriented techniques have been used exclusively in system design, and have not been supported by the ABAP language.

The Object-oriented Language Extension

ABAP Objects is a complete set of object-oriented statements that has been introduced into the ABAP language. This object-oriented extension of ABAP builds on the existing language, and is fully compatible with it. You can use ABAP Objects in existing programs, and can also use “conventional” ABAP in new ABAP Objects programs.

ABAP Objects supports object-oriented programming. Object orientation (OO), also know as the object-oriented paradigm, is a programming model that unites data and functions in objects. The rest of the ABAP language is primarily intended for structured programming, where data is stored in a structured form in database tables and function-oriented programs access and work with it.

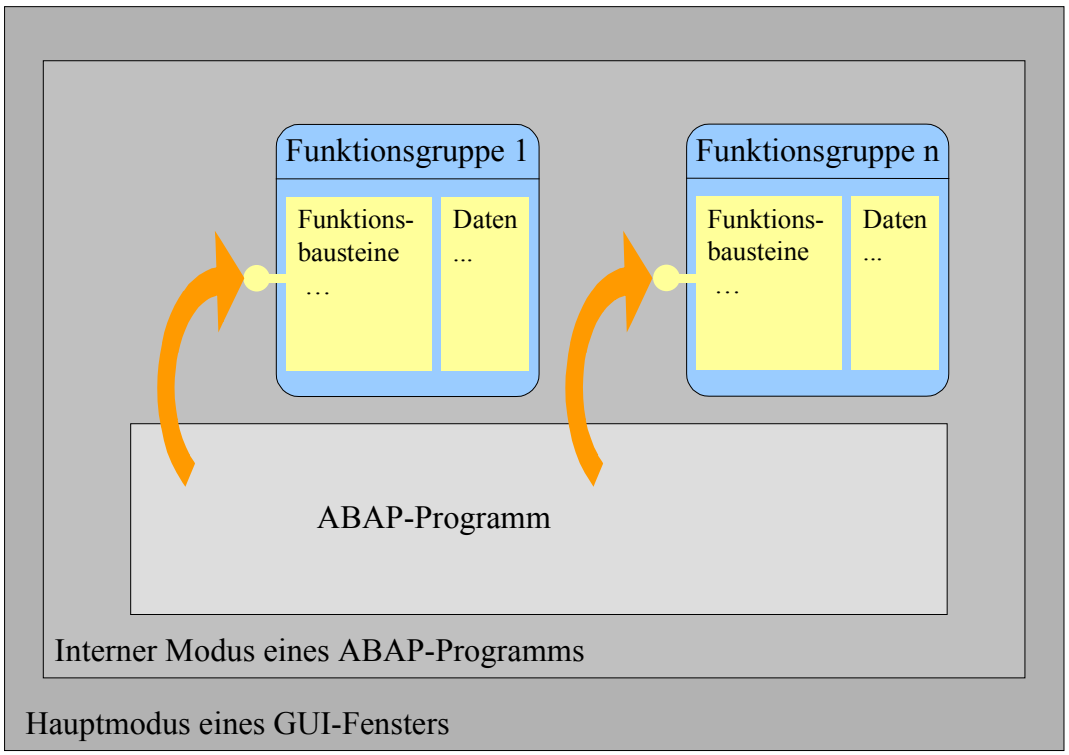
The object-oriented enhancement of ABAP is based on the models of Java and C++. It is compatible with external object interfaces such as DCOM and CORBA. The implementation of object-oriented elements in the kernel of the ABAP language has considerably increased response times when you work with ABAP Objects. SAP Business Objects and GUI objects - already object-oriented themselves - will also profit from being incorporated in ABAP Objects.

From Function Groups to Objects

At the center of any object-oriented model are objects, which contain attributes (data) and methods (functions). Objects should enable programmers to map a real problem and its proposed software solution on a one-to-one basis. Typical objects in a business environment are, for example, ‘customer’, ‘Order’, or ‘Invoice’. From Release 3.1 onwards, the Business Object Repository (BOR) has contained examples of such objects. The object model of ABAP Objects, the object-oriented extension of ABAP, is compatible with the object model of the BOR.

Before R/3 Release 4.0, the nearest equivalent of objects in ABAP were function modules and function groups. Suppose we have a function group for processing orders. The attributes of an order correspond to the global data of the function group, while the individual function modules represent actions that manipulate that data (methods). This means that the actual order data is encapsulated in the function group, and is never directly addressed, but instead only through the function modules. In this way, the function modules can ensure that the data is consistent.

When you run an ABAP program, the system starts a new internal session. The internal session has a memory area that contains the ABAP program and its associated data. When you call a function module, an **instance** of its function group plus its data, is loaded into the memory area of the internal session. An ABAP program can load several instances by calling function modules from **different** function groups.

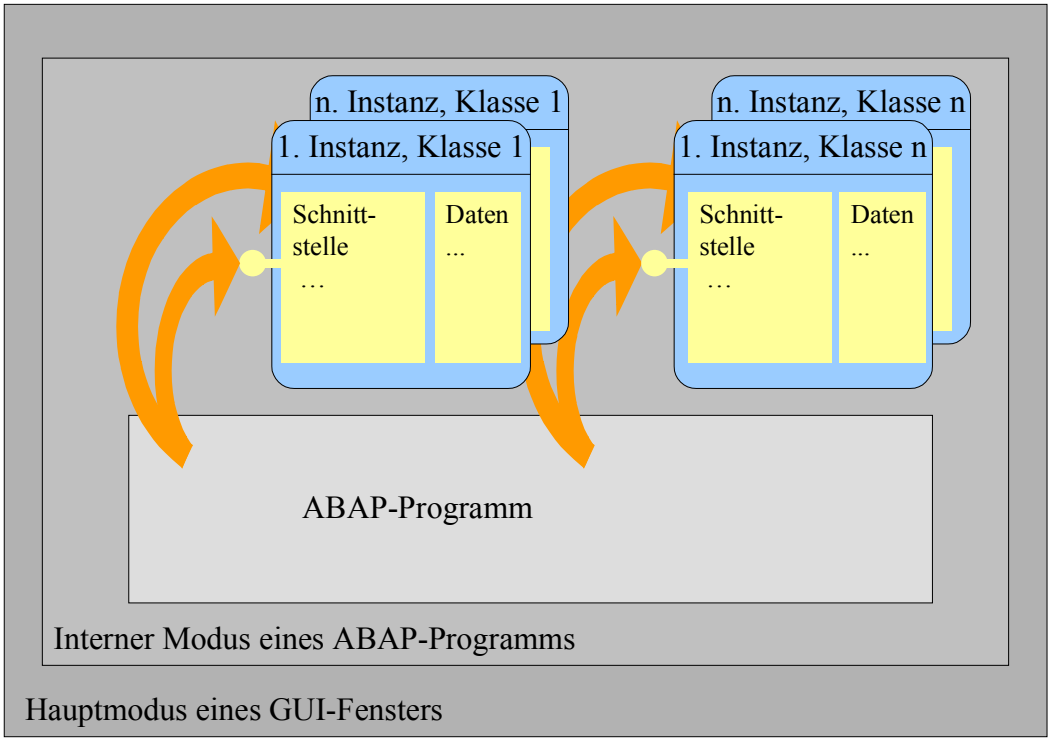


The instance of a function group in the memory area of the internal session almost represents an object in the sense of object orientation. (See also the definition in the section What is Object Orientation? [Page 1292].. When you call a function module, the calling program uses the instance of a function group, based on its description in the Function Builder. The program cannot access the data in the function group directly, but only through the function module. The function modules and their parameters are the **interface** between the function group and the user.

The main difference between real object orientation and function groups is that although a program can work with the instances of several function groups at the same time, it cannot work with several instances of a single function group. Suppose a program wanted to use several independent counters, or process several orders at the same time. In this case, you would have to adapt the function group to include instance administration, for example, by using numbers to differentiate between the instances.

In practice, this is very awkward. Consequently, the data is usually stored in the calling program, and the function modules are called to work with it (structured programming). One problem is, for example, that all users of the function module must use the same data structures as the function group itself. Changing the internal data structure of a function group affects many users, and it is often difficult to predict the implications. The only way to avoid this is to rely heavily on interfaces and a technique that guarantees that the internal structures of instances will remain hidden, allowing you to change them later without causing any problems.

This requirement is met by object orientation. ABAP Objects allows you to define data and functions in classes instead of function groups. Using classes, an ABAP program can work with any number of instances (objects) based on the **same** template.



From Function Groups to Objects

Instead of loading a single instance of a function group into memory implicitly when a function module is called, the ABAP program can now generate the instances of classes explicitly using the new ABAP statement CREATE OBJECT. The individual instances represent unique objects. You address these using object references. The object references allow the ABAP program to access the interfaces of the instances.

The following sections contain more information about classes, objects, interfaces, and object references in ABAP Objects.

Example

The following example shows the object-oriented aspect of function groups in the simple case of a counter.

Suppose we have a function group called COUNTER:

```
FUNCTION-POOL COUNTER.

DATA COUNT TYPE I.

FUNCTION SET_COUNTER.
* Local Interface IMPORTING VALUE(SET_VALUE)
  COUNT = SET_VALUE.
ENDFUNCTION.

FUNCTION INCREMENT_COUNTER.
  ADD 1 TO COUNT.
ENDFUNCTION.

FUNCTION GET_COUNTER.
* Local Interface: EXPORTING VALUE(GET_VALUE)
  GET_VALUE = COUNT.
ENDFUNCTION.
```

The function group has a global integer field COUNT, and three function modules, SET_COUNTER, INCREMENT_COUNTER, and GET_COUNTER, that work with the field. Two of the function modules have input and output parameters. These form the data interface of the function group.

Any ABAP program can then work with this function group. For example:

```
DATA NUMBER TYPE I VALUE 5.

CALL FUNCTION 'SET_COUNTER' EXPORTING SET_VALUE = NUMBER.

DO 3 TIMES.
  CALL FUNCTION 'INCREMENT_COUNTER'.
ENDDO.

CALL FUNCTION 'GET_COUNTER' IMPORTING GET_VALUE = NUMBER.
```

After this section of the program has been processed, the program variable NUMBER will have the value 8.

The program itself cannot access the COUNT field in the function group. Operations on this field are fully encapsulated in the function module. The program can only communicate with the function group by calling its function modules.

Classes

LIKE-Zusatz [Page 116]

Classes are templates for objects. Conversely, you can say that the type of an object is the same as its class. A class is an abstract description of an object. You could say that it is a set of instructions for building an object. The attributes of objects are defined by the **components** of the class, which describe the state and behavior of objects.

Local and Global Classes

Classes in ABAP Objects can be declared either globally or locally. You define global classes and interfaces in the Class Builder (Transaction SE24) in the ABAP Workbench. They are stored centrally in class pools [Page 1357] in the class library in the R/3 Repository. All of the ABAP programs in an R/3 System can access the global classes. Local classes are defined within an ABAP program. Local classes and interfaces can only be used in the program in which they are defined. When you use a class in an ABAP program, the system first searches for a local class with the specified name. If it does not find one, it then looks for a global class. Apart from the visibility question, there is no difference between using a global class and using a local class.

There is, however, a significant difference in the way that local and global classes are designed. If you are defining a local class that is only used in a single program, it is usually sufficient to define the outwardly visible components so that it fits into that program. Global classes, on the other hand, must be able to be used anywhere. This means that certain restrictions apply when you define the interface of a global class, since the system must be able to guarantee that any program using an object of a global class can recognize the data type of each interface parameter.

The following sections describe how to define local classes and interfaces in an ABAP program. For information about how to define local classes and interfaces, refer to the Class Builder [Ext.] section of the ABAP Workbench Tools documentation.

Defining Local Classes

Local classes consist of ABAP source code, enclosed in the ABAP statements CLASS ... ENDCLASS. A complete class definition consists of a declaration part and, if required, an implementation part. The declaration part of a class <class> is a statement block:

```
CLASS <class> DEFINITION.
...
ENDCLASS.
```

It contains the declaration for all **components** (attributes, methods, events) of the class. When you define local classes, the declaration part belongs to the global program data. You should therefore place it at the beginning of the program.

If you declare methods in the declaration part of a class, you must also write an implementation part for it. This consists of a further statement block:

```
CLASS <class> IMPLEMENTATION.
...
ENDCLASS.
```

The implementation part of a class contains the implementation of all **methods** of the class. The implementation part of a local class is a processing block. Subsequent coding that is not itself part of a processing block is therefore not accessible.

Structure of a Class

The following statements define the structure of a class:

- A class contains components
- Each component is assigned to a visibility section
- Classes implement methods

The following sections describe the structure of classes in more detail.

Class Components

The components of a class make up its contents. All components are declared in the declaration part of the class. The components define the attributes of the objects in a class. When you define the class, each component is assigned to one of the three **visibility sections**, which define the external **interface** of the class. All of the components of a class are visible within the class. All components are in the same namespace. This means that all components of the class must have names that are unique within the class.

There are two kinds of components in a class - those that exist separately for each object in the class, and those that exist only once for the whole class, regardless of the number of instances. Instance-specific components are known as instance components. Components that are not instance-specific are called static components.

In ABAP Objects, classes can define the following components. Since all components that you can declare in classes can also be declared in interfaces, the following descriptions apply equally to interfaces.

Attributes

Attributes are internal data fields within a class that can have any ABAP data type. The state of an object is determined by the contents of its attributes. One kind of attribute is the reference variable. Reference variables allow you to create and address objects. Reference variables can be defined in classes, allowing you to access objects from within a class.

Instance Attributes

The contents of instance attributes define the instance-specific state of an object. You declare them using the DATA statement.

Static Attributes

The contents of static attributes define the state of the class that is valid for all instances of the class. Static attributes exist once for each class. You declare them using the CLASS-DATA statement. They are accessible for the entire runtime of the class.

All of the objects in a class can access its static attributes. If you change a static attribute in an object, the change is visible in all other objects in the class.

Methods

Methods are internal procedures in a class that define the behavior of an object. They can access all of the attributes of a class. This allows them to change the data content of an object. They also have a parameter interface, with which users can supply them with values when calling them, and receive values back from them The private attributes of a class can only be changed by methods in the same class.

Classes

The definition and parameter interface of a method is similar to that of function modules. You define a method <met> in the definition part of a class and implement it in the implementation part using the following processing block:

METHOD <meth>.

...

ENDMETHOD.

You can declare local data types and objects in methods in the same way as in other ABAP procedures (subroutines and function modules). You call methods using the CALL METHOD statement.

Instance Methods

You declare instance methods using the METHODS statement. They can access all of the attributes of a class, and can trigger all of the events of the class.

Static Methods

You declare static methods using the CLASS-METHODS statement. They can only access static attributes and trigger static events.

Special Methods

As well as normal methods, which you call using CALL METHOD, there are two special methods called CONSTRUCTOR and CLASS_CONSTRUCTOR, which are automatically called when you create an object (CONSTRUCTOR) or when you first access the components of a class (CLASS_CONSTRUCTOR).

Events

Objects or classes can use events to trigger event handler methods in other objects or classes. In a normal method call, one method can be called by any number of users. When an event is triggered, any number of event handler methods can be called. The link between the trigger and the handler is not established until runtime. In a normal method call, the calling program determines the methods that it wants to call. These methods must exist. With events, the handler determines the events to which it wants to react. There does not have to be a handler method registered for every event.

The events of a class can be triggered in the methods of the same class using the RAISE EVENT statement. You can declare a method of the same or a different class as an event handler method for the event <evt> of class <class> using the addition FOR EVENT <evt> OF <class>.

Events have a similar parameter interface to methods, but only have output parameters. These parameters are passed by the trigger (RAISE EVENT statement) to the event handler method, which receives them as input parameters.

The link between trigger and handler is established dynamically in a program using the SET HANDLER statement. The trigger and handlers can be objects or classes, depending on whether you have instance or static events and event handler methods. When an event is triggered, the corresponding event handler methods are executed in all registered handling classes.

Instance Events

You declare instance events using the EVENTS statement. An instance event can only be triggered in an instance method.

Static Events

You declare static events using the CLASS-EVENTS statement. All methods (instance and static methods) can trigger static events. Static events are the only type of event that can be triggered in a static method.

See also Triggering and Handling Events [Page 1343].

Types

You can define your own ABAP data types within a class using the TYPES statement. Types are not instance-specific, and exist once only for all of the objects in a class.

Constants

Constants are special static attributes. You set their values when you declare them, and they can then no longer be changed. You declare them using the CONSTANTS statement. Constants are not instance-specific, and exist once only for all of the objects in a class.

Visibility Sections

You can divide the declaration part of a class into up to three visibility areas:

```
CLASS <class> DEFINITION.
  PUBLIC SECTION.
...
  PROTECTED SECTION.
...
  PRIVATE SECTION.
...
ENDCLASS.
```

These areas define the external visibility of the class components, that is, the interface between the class and its users. Each component of a class must be assigned to one of the visibility sections.

Public Section

All of the components declared in the public section are accessible to all users of the class, and to the methods of the class and any classes that inherit from it. The public components of the class form the **interface between the class and its users**.

Protected Section

All of the components declared in the protected section are accessible to all methods of the class and of classes that inherit from it. Protected components form a special **interface between a class and its subclasses**. Since inheritance is not active in Release 4.5B, the protected section currently has the same effect as the private section.

Private Section

Components that you declare in the private section are only visible in the methods of the same class. The private components are **not part of the external interface of the class**.

Encapsulation

The three visibility areas are the basis for one of the important features of object orientation - encapsulation. When you define a class, you should take great care in designing the public

components, and try to declare as few public components as possible. The public components of global classes may not be changed once you have released the class.

For example, public attributes are visible externally, and form a part of the interface between an object and its users. If you want to **encapsulate** the **state** of an object fully, you cannot declare any public attributes. As well as defining the visibility of an attribute, you can also protect it from changes using the READ-ONLY addition.

See also:

Overview Graphic [Page 1305]

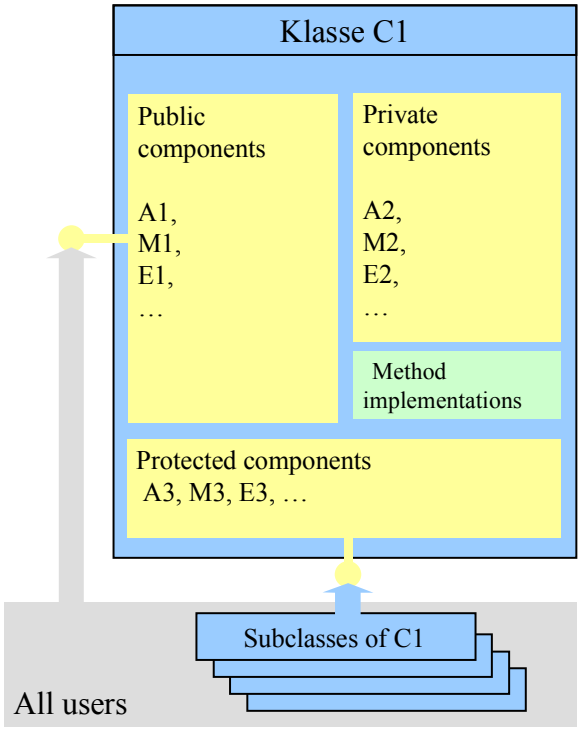
Example [Page 1306]

Overview Graphic

Classes

```
CLASS C1 DEFINITION.
  PUBLIC SECTION.
    DATA:     A1 ...
    METHODS:  M1 ...
    EVENTS:   E1 ...
  PROTECTED SECTION.
    DATA:     A2 ...
    METHODS:  M2 ...
    EVENTS:   E2 ...
  PRIVATE SECTION.
    DATA:     A3 ...
    METHODS:  M3 ...
    EVENTS:   E3 ...
ENDCLASS.
```

```
CLASS C1 IMPLEMENTATION.

  METHOD M1. ... ENDMETHOD.

  METHOD M2. ... ENDMETHOD.

  METHOD M3. ... ENDMETHOD.

ENDCLASS.
```



The left-hand side of the illustration shows the declaration and implementation parts of a local class C1. The right-hand side illustrates the structure of the class with the components in their respective visibility areas, and the implementation of the methods.

The public components of the class form the **interface between the class and its users**. The protected components are an interface to the subclasses of C1. The private components are not visible externally, and are fully encapsulated in the class. The methods in the implementation part have unrestricted access to all components of the class.

Classes - Introductory Example

Von Funktionsgruppen zu Objekten [Page 1296]

The following simple example uses ABAP Objects to program a counter. For comparison, see also the example [Page 1299] in From Function Groups to Objects

```
CLASS C_COUNTER DEFINITION.

  PUBLIC SECTION.
    METHODS: SET_COUNTER IMPORTING VALUE(SET_VALUE) TYPE I,
             INCREMENT_COUNTER,
             GET_COUNTER EXPORTING VALUE(GET_VALUE) TYPE I.

  PRIVATE SECTION.
    DATA COUNT TYPE I.

ENDCLASS.

CLASS C_COUNTER IMPLEMENTATION.

  METHOD SET_COUNTER.
    COUNT = SET_VALUE.
  ENDMETHOD.

  METHOD INCREMENT_COUNTER.
    ADD 1 TO COUNT.
  ENDMETHOD.

  METHOD GET_COUNTER.
    GET_VALUE = COUNT.
  ENDMETHOD.

ENDCLASS.
```

The class C_COUNTER contains three public methods - SET_COUNTER, INCREMENT_COUNTER, and GET_COUNTER. Each of these works with the private integer field COUNT. Two of the methods have input and output parameters. These form the data interface of the class. The field COUNT is not outwardly visible.

The example in the section Working with Objects [Page 1307] shows how you can create instances of the class C_COUNTER.

Object Handling

Datenreferenzen [Page 219]

Interfaces [Page 1337]

Vererbung [Page 1327]

Interfaces [Page 1337]

LIKE-Zusatz [Page 116]

Objects

Objects are instances of classes. Each object has a unique identity and its own attributes. All transient objects reside in the context of an internal session (memory area of an ABAP program). Persistent objects in the database are not yet available. A class can have any number of objects (instances).

Object References

To access an object from an ABAP program, you use object references. Object references are pointers to objects. In ABAP, they are always contained in reference variables.

Reference variables

Reference variables contain references. A reference variable is either initial or contains a reference to an existing object. The identity of an object depends on its reference. A reference variable that points to an object knows the identity of that object. Users cannot access the identity of the object directly.

Reference variables in ABAP are treated like other elementary data objects. This means that a reference variable can occur as a component of a structure or internal table as well as on its own.

Data Types for References

ABAP contains a predefined data type for references, comparable to the data types for structures or internal tables. The full data type is not defined until the declaration in the ABAP program. The data type of a reference variable determines how the program handles its value (that is, the object reference). There are two principal types of references: Class references and interface references (see Interfaces [Page 1337]).

You define class references using the

... TYPE REF TO <class>

addition in the TYPES or DATA statement, where <class> refers to a class. A reference variable with the type class reference is called a class reference variable, or class reference for short.

A class reference <cref> allows a user to create an instance (object) of the corresponding class, and to address a visible component <comp> within it using the form

cref->comp

Creating Objects

Before you can create an object for a class, you need to declare a reference variable with reference to that class. Once you have declared a class reference variable <obj> for a class <class>, you can create an object using the statement

CREATE OBJECT <cref>.

This statement creates an instance of the class <class>, and the reference variable <cref> contains a reference to the object.

Addressing the Components of Objects

Programs can only access the instance components of an object using references in reference variables. The syntax is as follows (where <ref> is a reference variable):

- To access an attribute <attr>: <ref>-><attr>
- To call a method <meth>: CALL METHOD <ref>-><meth>

You can access static components using the class name as well as the reference variable. It is also possible to address the static components of a class before an object has been created.

- Addressing a static attribute <attr>: <class>=><attr>
- Calling a static method <meth>: CALL METHOD <class>=><meth>

Within a class, you can use the self-reference ME to access the individual components:

- To access an attribute <attr> in the same class: ME-><attr>
- To call a method <meth> in the same class: CALL METHOD ME-><meth>

Self references allow an object to give other objects a reference to it. You can also access attributes in methods from within an object even if they are obscured by local attributes of the method.

Creating More Than One Instance of a Class

In a program, you can create any number of objects from the same class. The objects are fully independent of each other. Each one has its own identity within the program and its own attributes. Each CREATE OBJECT statement generates a new object, whose identity is defined by its unique object reference.

Assigning References

You can assign references to different reference variables using the MOVE statement. In this way, you can make the references in several reference variables point to the same object. When you assign a reference to a different reference variable, their types must be either compatible or convertible.

When you use the MOVE statement or the assignment operator (=) to assign reference variables, the system must be able to recognize in the syntax check whether an assignment is possible. The same applies when you pass reference variables to procedures as parameters. If you write the statement

<cref1> = <cref2>

the two class references <cref1> and <cref2> must have the same type, that is, they must either refer to the same class, or the class of <cref1> must be the predefined empty class OBJECT.

The class OBJECT has no components, and has the same function for reference variables as the data type ANY has for normal variables. Reference variables with the type OBJECT can function as containers for passing references. However, you cannot use them to address objects.

Object Lifetime

An object exists for as long as it is being used in the program. An object is in use by a program for as long as at least one reference points to it, or at least one method of the object is registered as an event handler.

As soon as there are no more references to an object, and so long as none of its methods are registered as event handlers, it is deleted by the automatic memory management (garbage collection). The ID of the object then becomes free, and can be used by a new object.

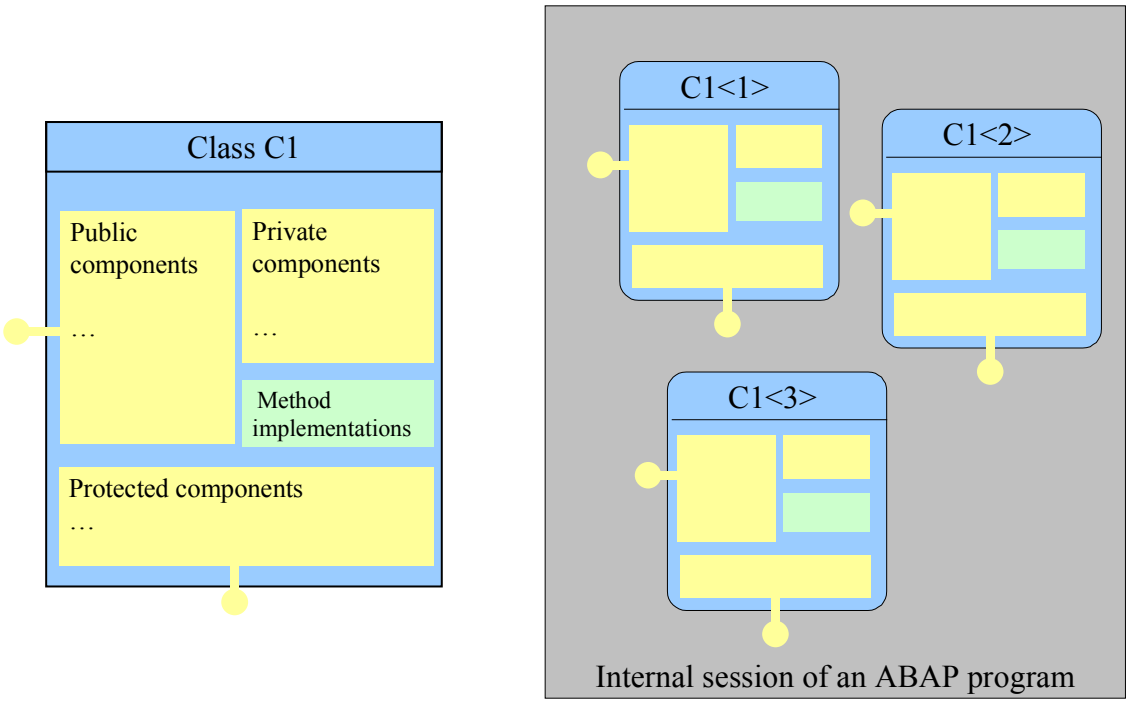
See also:

Overview Graphic [Page 1310]

Example [Page 1311]

Overview Graphic

Objects as Instances of a Class



Class ⟶ Instances of the class

The above illustration shows a class C1 on the left, with its instances represented in the internal session of an ABAP program on the right. To distinguish them from classes, instances are drawn with rounded corners. The instance names above use the same notation as is used for reference variables in the Debugger.

Objects - Introductory Example

Klassen [Page 1300]

The following example shows how to create and use an instance of the class C_COUNTER that we created in the previous section (see the example [Page 1306] under Classes and Class Components):

 [Ext.]

Declaring and Calling Methods

GET REFERENCE [Page 222]

This section contains explains how to work with methods in ABAP Objects. For precise details of the relevant ABAP statements, refer to the corresponding keyword documentation in the ABAP Editor. The example [Page 1315] shows how to declare, implement, and call methods.

Declaring Methods

You can declare methods in the declaration part of a class or in an interface. To declare instance methods, use the following statement:

METHODS <meth> IMPORTING.. [VALUE(]<i_i>[)] TYPE type [OPTIONAL]..
EXPORTING.. [VALUE(]<e_i>[)] TYPE type [OPTIONAL]..
CHANGING.. [VALUE(]<c_i>[)] TYPE type [OPTIONAL]..
RETURNING VALUE(<r>)
EXCEPTIONS.. <e_i>..

and the appropriate additions.

To declare static methods, use the following statement:

CLASS-METHODS <meth>...

Both statements have the same syntax.

When you declare a method, you also define its parameter interface using the additions IMPORTING, EXPORTING, CHANGING, and RETURNING. The additions define the input, output, and input/output parameters, and the return code. They also define the attributes of the interface parameters, namely whether a parameter is to be passed by reference or value (VALUE), its type (TYPE), and whether it is optional (OPTIONAL, DEFAULT). Unlike in function modules, the default way of passing a parameter in a method is by reference. To pass a parameter by value, you must do so explicitly using the VALUE addition. The return value (RETURNING parameter) must always be passed explicitly as a value. This is suitable for methods that return a single output value. If you use it, you cannot use EXPORTING or CHANGING parameters.

As in function modules, you can use exception parameters (EXCEPTIONS) to allow the user to react to error situations when the method is executed.

Implementing Methods

You must implement all of the methods in a class in the implementation part of the class in a

METHOD <meth>.
...
ENDMETHOD.

block. When you implement the method, you do not have to specify any interface parameters, since these are defined in the method declaration. The interface parameters of a method behave like local variables within the method implementation. You can define additional local variables within a method using the DATA statement.

As in function modules, you can use the RAISE <exception> and MESSAGE RAISING statements to handle error situations.

When you implement a static method, remember that it can only work with the static attributes of your class. Instance methods can work with both static and instance attributes.

Calling Methods

To call a method, use the following statement:

CALL METHOD <meth> EXPORTING... <i_i> =.<f_i>...
IMPORTING... <e_i> =.<g_i>...
CHANGING ... <c_i> =.<f_i>...
RECEIVING r = h
EXCEPTIONS... <e_i> = rc_i...

The way in which you address the method <method> depends on the method itself and from where you are calling it. Within the implementation part of a class, you can call the methods of the same class directly using their name <meth>.

CALL METHOD <meth>...

Outside the class, the visibility of the method depends on whether you can call it at all. Visible instance methods can be called from outside the class using

CALL METHOD <ref>-><meth>...

where <ref> is a reference variable whose value points to an instance of the class. Visible instance methods can be called from outside the class using

CALL METHOD <class>=><meth>...

where <class> is the name of the relevant class.

When you call a method, you must pass all non-optional input parameters using the EXPORTING or CHANGING addition in the CALL METHOD statement. You can (but do not have to) import the output parameters into your program using the IMPORTING or RECEIVING addition. Equally, you can (but do not have to) handle any exceptions triggered by the exceptions using the EXCEPTIONS addition. However, this is recommended.

You pass and receive values to and from methods in the same way as with function modules, that is, with the syntax:

... <Formal parameter> = <Actual parameter>

after the corresponding addition. The interface parameters (formal parameters) are always on the left-hand side of the equals sign. The actual parameters are always on the right. The equals sign is not an assignment operator in this context; it merely serves to assign program variables to the interface parameters of the method.

If the interface of a method consists only of a single IMPORTING parameter, you can use the following shortened form of the method call:

CALL METHOD <method>(f).

The actual parameter <f> is passed to the input parameters of the method.

If the interface of a method consists only of IMPORTING parameters, you can use the following shortened form of the method call:

CALL METHOD <method>(....<i_i> =.<f_i>...).

Each actual parameter <f_i> is passed to the corresponding formal parameter <i_i>.

Event Handler Methods

Event handler methods are special methods that cannot all be called using the CALL METHOD statement. Instead, they are triggered using events. You define a method as an event handler method using the addition

... FOR EVENT <evt> OF <cif>...

in the METHODS or CLASS-METHODS statement.

The following special rules apply to the interface of an event handler method:

- The interface may only consist of IMPORTING parameters.
- Each IMPORTING parameter must be an EXPORTING parameter of the event <evt>.
- The attributes of the parameters are defined in the declaration of the event <evt> (EVENTS statement) and are adopted by the event handler method.

See also Triggering and Handling Events [Page 1343]

Constructors

Constructors are special methods that cannot be called using CALL METHOD. Instead, they are called automatically by the system to set the starting state of a new object or class. There are two types of constructors - instance constructors and static constructors. Constructors are methods with a predefined name. To use them, you must declare them explicitly in the class.

The instance constructor of a class is the predefined instance method CONSTRUCTOR. You declare it in the public section as follows:

METHODS CONSTRUCTOR
    IMPORTING.. [VALUE(]<i_i>[)] TYPE type [OPTIONAL]..
    EXCEPTIONS.. <e_i>.

and implement it in the implementation section like any other method. The system calls the instance constructor once for each instance of the class, directly **after** the object has been created in the CREATE OBJECT statement. You can pass the input parameters of the instance constructor and handle its exceptions using the EXPORTING and EXCEPTIONS additions in the CREATE OBJECT statement.

The static constructor of a class is the predefined static method CLASS_CONSTRUCTOR. You declare it in the public section as follows:

CLASS-METHODS CLASS_CONSTRUCTOR.

and implement it in the implementation section like any other method. The static constructor has no parameters. The system calls the static constructor once for each class, **before** the class is accessed for the first time. The static constructor **cannot** therefore access the components of its own class.

The methods example [Page 1315] shows how to use instance and static constructors.

Methods in ABAP Objects - Example

The following example shows how to declare, implement, and use methods in ABAP Objects.

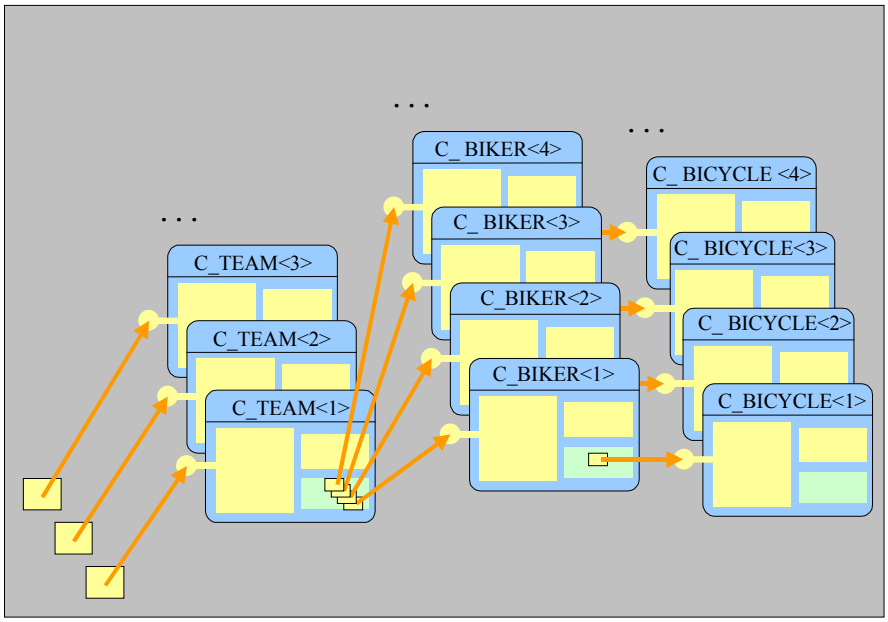
Overview

This example uses three classes called C_TEAM, C_BIKER, and C_BICYCLE. A user (a program) can create objects of the class C_TEAM. On a selection screen, the class C_TEAM asks for the number of members of each team.

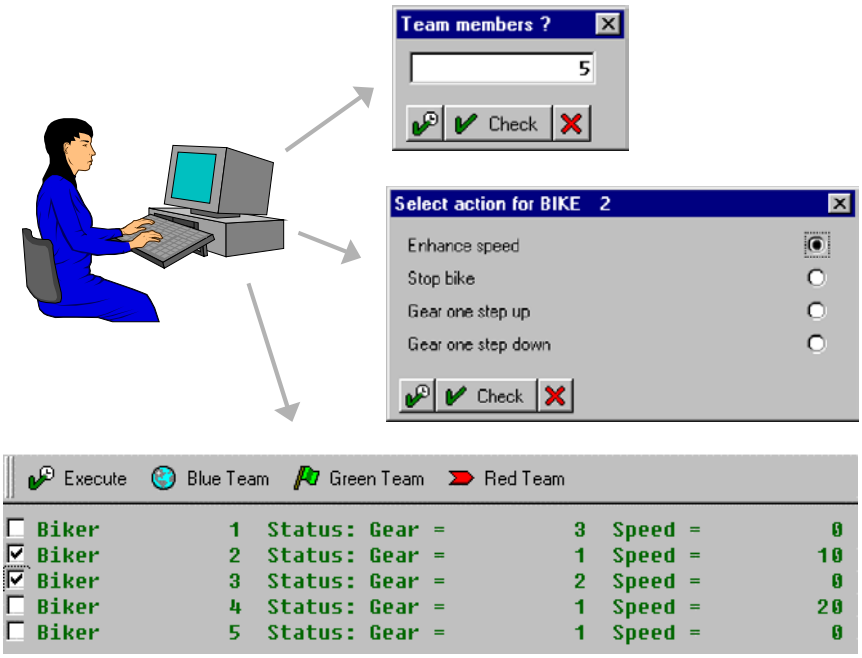
Each object in the class C_TEAM can create as many instances of the class C_BIKER as there are members in the team. Each instance of the class C_BIKER creates an instances of the class C_BICYCLE.

Each instance of the class C_TEAM can communicate with the program user through an interactive list. The program user can choose individual team members for actions. The instances of the class C_BIKER allow the program user to choose the action on a further selection screen.

Referenzvariablen und Instanzen:



Benutzerinteraktion:



Constraints

The ABAP statements used for list processing are not yet fully available in ABAP Objects. However, to produce a simple test output, you can use the following statements:

- WRITE [AT] /<offset>(<length>) <f>
- ULINE
- SKIP
- NEW-LINE

Note: The behavior of formatting and interactive list functions in their current state are not guaranteed. Incompatible changes could occur in a future release.

Declarations

This example is implemented using **local** classes, since selection screens belong to an ABAP program, and cannot be defined or called in global classes. Below are the definitions of the two selection screens and three classes:

```
*********************************************************************
* Global Selection Screens
*********************************************************************

SELECTION-SCREEN BEGIN OF: SCREEN 100 TITLE TIT1, LINE.
  PARAMETERS MEMBERS TYPE I DEFAULT 10.
SELECTION-SCREEN END OF: LINE, SCREEN 100.

*---------------------------------------------------------------------

SELECTION-SCREEN BEGIN OF SCREEN 200 TITLE TIT2.
  PARAMETERS: DRIVE    RADIOBUTTON GROUP ACTN,
              STOP     RADIOBUTTON GROUP ACTN,
              GEARUP   RADIOBUTTON GROUP ACTN,
              GEARDOWN RADIOBUTTON GROUP ACTN.
SELECTION-SCREEN END OF SCREEN 200.

*********************************************************************
* Class Definitions
*********************************************************************

CLASS: C_BIKER DEFINITION DEFERRED,
       C_BICYCLE DEFINITION DEFERRED.

*---------------------------------------------------------------------

CLASS C_TEAM DEFINITION.

  PUBLIC SECTION.

  TYPES: BIKER_REF TYPE REF TO C_BIKER,
         BIKER_REF_TAB TYPE STANDARD TABLE OF BIKER_REF
                                           WITH DEFAULT KEY,
```

```
          BEGIN OF STATUS_LINE_TYPE,
            FLAG(1)  TYPE C,
            TEXT1(5) TYPE C,
            ID       TYPE I,
            TEXT2(7) TYPE C,
            TEXT3(6) TYPE C,
            GEAR     TYPE I,
            TEXT4(7) TYPE C,
            SPEED    TYPE I,
          END OF STATUS_LINE_TYPE.

  CLASS-METHODS: CLASS_CONSTRUCTOR.

  METHODS: CONSTRUCTOR,
           CREATE_TEAM,
           SELECTION,
           EXECUTION.

  PRIVATE SECTION.

  CLASS-DATA: TEAM_MEMBERS TYPE I,
              COUNTER TYPE I.

  DATA: ID TYPE I,
        STATUS_LINE TYPE STATUS_LINE_TYPE,
        STATUS_LIST TYPE SORTED TABLE OF STATUS_LINE_TYPE
                                    WITH UNIQUE KEY ID,

        BIKER_TAB TYPE BIKER_REF_TAB,
        BIKER_SELECTION LIKE BIKER_TAB,
        BIKER LIKE LINE OF BIKER_TAB.

  METHODS: WRITE_LIST.

ENDCLASS.

*---------------------------------------------------------------------

CLASS C_BIKER DEFINITION.

  PUBLIC SECTION.

  METHODS: CONSTRUCTOR IMPORTING TEAM_ID TYPE I MEMBERS TYPE I,
           SELECT_ACTION,
           STATUS_LINE EXPORTING LINE
                       TYPE C_TEAM=>STATUS_LINE_TYPE.

  PRIVATE SECTION.

  CLASS-DATA COUNTER TYPE I.

  DATA: ID TYPE I,
        BIKE TYPE REF TO C_BICYCLE,
        GEAR_STATUS  TYPE I VALUE 1,
        SPEED_STATUS TYPE I VALUE 0.

  METHODS BIKER_ACTION IMPORTING ACTION TYPE I.

ENDCLASS.

*---------------------------------------------------------------------
```

```
CLASS C_BICYCLE DEFINITION.

  PUBLIC SECTION.

  METHODS: DRIVE EXPORTING VELOCITY TYPE I,
           STOP  EXPORTING VELOCITY TYPE I,
           CHANGE_GEAR IMPORTING CHANGE TYPE I
                       RETURNING VALUE(GEAR) TYPE I
                       EXCEPTIONS GEAR_MIN GEAR_MAX.

  PRIVATE SECTION.

  DATA: SPEED TYPE I,
        GEAR  TYPE I VALUE 1.

  CONSTANTS: MAX_GEAR TYPE I VALUE 18,
             MIN_GEAR TYPE I VALUE 1.

ENDCLASS.

*********************************************************************
```

Note that none of the three classes has any public attributes. The states of the classes can only be changed by their methods. The class C_TEAM contains a static constructor CLASS_CONSTRUCTOR. C_TEAM and C_BIKER both contain instance constructors.

Implementations

The implementation parts of the classes contain the implementations of all of the methods declared in the corresponding declaration parts. The interfaces of the methods have already been defined in the declarations. In the implementations, the interface parameters behave like local data.

Methods of Class C_TEAM

The following methods are implemented in the section

```
CLASS C_TEAM IMPLEMENTATION.
```

...

```
ENDCLASS.
```

CLASS_CONSTRUCTOR

```
  METHOD CLASS_CONSTRUCTOR.
    TIT1 = 'Team members ?'.
    CALL SELECTION-SCREEN 100 STARTING AT 5 3.
    IF SY-SUBRC NE 0.
      LEAVE PROGRAM.
    ELSE.
      TEAM_MEMBERS = MEMBERS.
    ENDIF.
  ENDMETHOD.
```

The static constructor is executed before the class C_TEAM is used for the first time in a program. It calls the selection screen 100 and sets the static attribute TEAM_MEMBERS to the value entered by the program user. This attribute has the same value for all instances of the class C_TEAM.

CONSTRUCTOR

```
METHOD CONSTRUCTOR.
  COUNTER = COUNTER + 1.
  ID = COUNTER.
ENDMETHOD.
```

The instance constructor is executed directly after each instance of the class C_TEAM is created. It is used to count the number of instance of C_TEAM in the static attribute COUNTER, and assigns the corresponding number to the instance attribute ID of each instance of the class.

CREATE_TEAM

```
METHOD CREATE_TEAM.
  DO TEAM_MEMBERS TIMES.
    CREATE OBJECT BIKER EXPORTING TEAM_ID = ID
                                  MEMBERS = TEAM_MEMBERS.
    APPEND BIKER TO BIKER_TAB.
    CALL METHOD BIKER->STATUS_LINE IMPORTING LINE = STATUS_LINE.
    APPEND STATUS_LINE TO STATUS_LIST.
  ENDDO.
ENDMETHOD.
```

The public instance method CREATE_TEAM can be called by any user of the class containing a reference variable with a reference to an instance of the class. It is used to create instances of the class C_BIKER, using the **private** reference variable BIKER in the class C_TEAM. You must pass both input parameters for the instance constructor of class C_BIKER in the CREATE OBJECT statement. The references to the newly-created instances are inserted into the private internal table BIKER_TAB. After the method has been executed, each line of the internal table contains a reference to an instance of the class C_BIKER. These references are only visible within the class C_TEAM. External users cannot address the objects of class C_BIKER.

CREATE_TEAM also calls the method STATUS_LINE for each newly-created object, and uses the work area STATUS_LINE to append its output parameter LINE to the private internal table STATUS_LIST.

SELECTION

```
METHOD SELECTION.
  CLEAR BIKER_SELECTION.
  DO.
    READ LINE SY-INDEX.
    IF SY-SUBRC <> 0. EXIT. ENDIF.
    IF SY-LISEL+0(1) = 'X'.
      READ TABLE BIKER_TAB INTO BIKER INDEX SY-INDEX.
      APPEND BIKER TO BIKER_SELECTION.
    ENDIF.
  ENDDO.
  CALL METHOD WRITE_LIST.
ENDMETHOD.
```

The public instance method SELECTION can be called by any user of the class containing a reference variable with a reference to an instance of the class. It selects all of the lines in the current list in which the checkbox in the first column is selected. For these lines, the system copies the corresponding reference variables from the table BIKER_TAB into an additional

private internal table BIKER_SELECTION. SELECTION then calls the private method WRITE_LIST, which displays the list.

EXECUTION

```
METHOD EXECUTION.
  CHECK NOT BIKER_SELECTION IS INITIAL.
  LOOP AT BIKER_SELECTION INTO BIKER.
    CALL METHOD BIKER->SELECT_ACTION.
    CALL METHOD BIKER->STATUS_LINE IMPORTING LINE = STATUS_LINE.
    MODIFY TABLE STATUS_LIST FROM STATUS_LINE.
  ENDLOOP.
  CALL METHOD WRITE_LIST.
ENDMETHOD.
```

The public instance method EXECUTION can be called by any user of the class containing a reference variable with a reference to an instance of the class. The method calls the two methods SELECT_ACTION and STATUS_LINE for each instance of the class C_BIKER for which there is a reference in the table BIKER_SELECTION. The line of the table STATUS_LIST with the same key as the component ID in the work area STATUS_LINE is overwritten and displayed by the private method WRITE_LIST.

WRITE_LIST

```
METHOD WRITE_LIST.
  SET TITLEBAR 'TIT'.
  SY-LSIND = 0.
  SKIP TO LINE 1.
  POSITION 1.
  LOOP AT STATUS_LIST INTO STATUS_LINE.
    WRITE: / STATUS_LINE-FLAG AS CHECKBOX,
             STATUS_LINE-TEXT1,
             STATUS_LINE-ID,
             STATUS_LINE-TEXT2,
             STATUS_LINE-TEXT3,
             STATUS_LINE-GEAR,
             STATUS_LINE-TEXT4,
             STATUS_LINE-SPEED.
  ENDLOOP.
ENDMETHOD.
```

The private instance method WRITE_LIST can only be called from the methods of the class C_TEAM. It is used to display the private internal table STATUS_LIST on the basic list (SY-LSIND = 0) of the program.

Methods of Class C_BIKER

The following methods are implemented in the section

```
CLASS C_BIKER IMPLEMENTATION.
```

...

```
ENDCLASS.
```

Methods in ABAP Objects - Example

CONSTRUCTOR

```
METHOD CONSTRUCTOR.
  COUNTER = COUNTER + 1.
  ID = COUNTER - MEMBERS * ( TEAM_ID - 1).
  CREATE OBJECT BIKE.
ENDMETHOD.
```

The instance constructor is executed directly after each instance of the class C_BIKER is created. It is used to count the number of instance of C_BIKER in the static attribute COUNTER, and assigns the corresponding number to the instance attribute ID of each instance of the class. The constructor has two input parameters - TEAM_ID and MEMBERS - which you must pass in the CREATE OBJECT statement when you create an instance of C_BIKER.

The instance constructor also creates an instance of the class C_BICYCLE for each new instance of the class C_BIKER. The reference in the private reference variable BIKE of each instance of C_BIKER points to a corresponding instance of the class C_BICYCLE. No external user can address these instances of the class C_BICYCLE.

SELECT_ACTION

```
METHOD SELECT_ACTION.
  DATA ACTIVITY TYPE I.
  TIT2 = 'Select action for BIKE'.
  TIT2+24(3) = ID.
  CALL SELECTION-SCREEN 200 STARTING AT 5 15.
  CHECK NOT SY-SUBRC GT 0.
  IF GEARUP = 'X' OR GEARDOWN = 'X'.
    IF GEARUP = 'X'.
      ACTIVITY = 1.
    ELSEIF GEARDOWN = 'X'.
      ACTIVITY = -1.
    ENDIF.
  ELSEIF DRIVE = 'X'.
    ACTIVITY = 2.
  ELSEIF STOP = 'X'.
    ACTIVITY = 3.
  ENDIF.
  CALL METHOD BIKER_ACTION( ACTIVITY).
ENDMETHOD.
```

The public instance method SELECT_ACTION can be called by any user of the class containing a reference variable with a reference to an instance of the class. The method calls the selection screen 200 and analyzes the user input. After this, it calls the private method BIKER_ACTION of the same class. The method call uses the shortened form to pass the actual parameter ACTIVITY to the formal parameter ACTION.

BIKER_ACTION

```
METHOD BIKER_ACTION.
  CASE ACTION.
    WHEN -1 OR 1.
      CALL METHOD BIKE->CHANGE_GEAR
                        EXPORTING CHANGE = ACTION
                        RECEIVING GEAR = GEAR_STATUS
                        EXCEPTIONS GEAR_MAX = 1
```

```
                                    GEAR_MIN = 2.
      CASE SY-SUBRC.
        WHEN 1.
          MESSAGE I315(AT) WITH 'BIKE' ID
                               ' is already at maximal gear!'.
        WHEN 2.
          MESSAGE I315(AT) WITH 'BIKE' ID
                               ' is already at minimal gear!'.
      ENDCASE.
    WHEN 2.
      CALL METHOD BIKE->DRIVE IMPORTING VELOCITY = SPEED_STATUS.
    WHEN 3.
      CALL METHOD BIKE->STOP IMPORTING VELOCITY = SPEED_STATUS.
  ENDCASE.
ENDMETHOD.
```

The private instance method BIKER_ACTION can only be called from the methods of the class C_BIKER. The method calls other methods in the instance of the class C_BICYCLE to which the reference in the reference variable BIKE is pointing, depending on the value in the input parameter ACTION.

STATUS_LINE

```
METHOD STATUS_LINE.
  LINE-FLAG = SPACE.
  LINE-TEXT1 = 'Biker'.
  LINE-ID = ID.
  LINE-TEXT2 = 'Status:'.
  LINE-TEXT3 = 'Gear = '.
  LINE-GEAR  = GEAR_STATUS.
  LINE-TEXT4 = 'Speed = '.
  LINE-SPEED = SPEED_STATUS.
ENDMETHOD.
```

The public instance method STATUS_LINE can be called by any user of the class containing a reference variable with a reference to an instance of the class. It fills the structured output parameter LINE with the current attribute values of the corresponding instance.

Methods of Class C_BICYCLE

The following methods are implemented in the section

```
CLASS C_BICYCLE IMPLEMENTATION.
```

...

```
ENDCLASS.
```

DRIVE

```
METHOD DRIVE.
  SPEED = SPEED  + GEAR * 10.
  VELOCITY = SPEED.
ENDMETHOD.
```

The public instance method DRIVE can be called by any user of the class containing a reference variable with a reference to an instance of the class. The method changes the value of the private attribute SPEED and passes it to the caller using the output parameter VELOCITY.

STOP

```
METHOD STOP.
  SPEED = 0.
  VELOCITY = SPEED.
ENDMETHOD.
```

The public instance method STOP can be called by any user of the class containing a reference variable with a reference to an instance of the class. The method changes the value of the private attribute SPEED and passes it to the caller using the output parameter VELOCITY.

CHANGE_GEAR

```
METHOD CHANGE_GEAR.
  GEAR = ME->GEAR.
  GEAR = GEAR + CHANGE.
  IF GEAR GT MAX_GEAR.
    GEAR = MAX_GEAR.
    RAISE GEAR_MAX.
  ELSEIF GEAR LT MIN_GEAR.
    GEAR = MIN_GEAR.
    RAISE GEAR_MIN.
  ENDIF.
  ME->GEAR = GEAR.
ENDMETHOD.
```

The public instance method CHANGE_GEAR can be called by any user of the class containing a reference variable with a reference to an instance of the class. The method changes the value of the private attribute GEAR. Since the formal parameter with the same name obscures the attribute in the method, the attribute has to be addressed using the self-reference ME->GEAR.

Using the Classes in a Program

The following program shows how the above classes can be used in a program. The declarations of the selection screens and local classes, and the implementations of the methods must also be a part of the program.

```
REPORT OO_METHODS_DEMO NO STANDARD PAGE HEADING.

*---------------------------------------------------------------------*
* Declarations and Implementations
*---------------------------------------------------------------------*

...

*---------------------------------------------------------------------*
* Global Program Data
*---------------------------------------------------------------------*

TYPES TEAM TYPE REF TO C_TEAM.

DATA: TEAM_BLUE  TYPE TEAM,
      TEAM_GREEN TYPE TEAM,
      TEAM_RED   TYPE TEAM.

DATA  COLOR(5).
```

```
*********************************************************************
* Program events
*********************************************************************

START-OF-SELECTION.

  CREATE OBJECT: TEAM_BLUE,
                 TEAM_GREEN,
                 TEAM_RED.

  CALL METHOD: TEAM_BLUE->CREATE_TEAM,
               TEAM_GREEN->CREATE_TEAM,
               TEAM_RED->CREATE_TEAM.

  SET PF-STATUS 'TEAMLIST'.

  WRITE '                    Select a team!              ' COLOR = 2.

*---------------------------------------------------------------------

AT USER-COMMAND.
  CASE SY-UCOMM.
    WHEN 'TEAM_BLUE'.
      COLOR = 'BLUE '.
      FORMAT COLOR = 1 INTENSIFIED ON INVERSE ON.
      CALL METHOD TEAM_BLUE->SELECTION.
    WHEN 'TEAM_GREEN'.
      COLOR = 'GREEN'.
      FORMAT COLOR = 5 INTENSIFIED ON INVERSE ON.
      CALL METHOD TEAM_GREEN->SELECTION.
    WHEN 'TEAM_RED'.
      COLOR = 'RED '.
      FORMAT COLOR = 6 INTENSIFIED ON INVERSE ON.
      CALL METHOD TEAM_RED->SELECTION.
    WHEN 'EXECUTION'.
      CASE COLOR.
        WHEN 'BLUE '.
          FORMAT COLOR = 1 INTENSIFIED ON INVERSE ON.
          CALL METHOD TEAM_BLUE->SELECTION.
          CALL METHOD TEAM_BLUE->EXECUTION.
        WHEN 'GREEN'.
          FORMAT COLOR = 5 INTENSIFIED ON INVERSE ON.
          CALL METHOD TEAM_GREEN->SELECTION.
          CALL METHOD TEAM_GREEN->EXECUTION.
        WHEN 'RED '.
          FORMAT COLOR = 6 INTENSIFIED ON INVERSE ON.
          CALL METHOD TEAM_RED->SELECTION.
          CALL METHOD TEAM_RED->EXECUTION.
      ENDCASE.
  ENDCASE.
*********************************************************************
```

The program contains three class reference variables that refer to the class C_TEAM. It creates three objects from the class, to which the references in the reference variables then point. In each object, it calls the method CREATE_TEAM. The method CLASS_CONSTRUCTOR of class C_TEAM is executed before the first of the objects is created. The status TEAMLIST for the basic list allows the user to choose one of four functions:

Methods in ABAP Objects - Example

When the user chooses a function, the event AT USER-COMMAND is triggered and public methods are called in one of the three instances of C_TEAM, depending on the user's choice. The user can change the state of an object by selecting the corresponding line in the status list.

Inheritance

Inheritance allows you to derive a new class from an existing class. You do this using the INHERITING FROM addition in the

CLASS <subclass> DEFINITION INHERITING FROM <superclass>.

statement. The new class <subclass> inherits all of the components of the existing class <superclass>. The new class is called the subclass of the class from which it is derived. The original class is called the superclass of the new class.

If you do not add any new declarations to the subclass, it contains the same components as the superclass. However, only the **public** and **protected** components of the superclass are visible in the subclass. Although the private components of the superclass exist in the subclass, they are not visible. You can declare private components in a subclass that have the same names as private components of the superclass. Each class works with its own private components. Methods that a subclass inherits from a superclass use the private attributes of the superclass, and not any private components of the subclass with the same names.

If the superclass does not have a private visibility section, the subclass is an exact replica of the superclass. However, you can add new components to the subclass. This allows you to turn the subclass into a specialized version of the superclass. If a subclass is itself the superclass of further classes, you introduce a new level of specialization.

A class can have more than one direct subclass, but it may only have one direct superclass. This is called **single inheritance**. When subclasses inherit from superclasses and the superclass is itself the subclass of another class, all of the classes involved form an inheritance tree, whose degree of specialization increases with each new hierarchical level you add. Conversely, the classes become more generalized until you reach the root node of the inheritance tree. The root node of all inheritance trees in ABAP Objects is the predefined empty class OBJECT. This is the most generalized class possible, since it contains neither attributes nor methods. When you define a new class, you do not have to specify it explicitly as the superclass - the relationship is always implicitly defined. Within an inheritance tree, two adjacent nodes are the direct superclass or direct subclass of one another. Other related nodes are referred to as superclasses and subclasses. The component declarations in a subclass are distributed across all levels of the inheritance tree.

Redefining Methods

All subclasses contain the components of all classes between themselves and the root node in an inheritance tree. The visibility of a component cannot be changed. However, you can use the REDEFINITION addition in the METHODS statement to redefine an inherited public or protected instance method in a subclass and make its function more specialized. When you redefine a method, you cannot change its interface. The method retains the same name and interface, but has a new implementation.

The method declaration and implementation in the superclass is not affected when you redefine the method in a subclass. The implementation of the redefinition in the subclass obscures the original implementation in the superclass.

Any reference that points to an object of the subclass uses the redefined method, even if the reference was defined with reference to the superclass. This particularly applies to the self-reference ME->. If, for example, a superclass method M1 contains a call CALL METHOD [ME->]M2, and M2 is redefined in a subclass, calling M1 from an instance of the subclass will cause

the original method M2 to be called, and calling M1 from an instance of the subclass will cause the redefined method M2 to be called.

Within a redefine method, you can use the pseudoreference SUPER-> to access the obscured method. This enables you to use the existing function of the method in the superclass without having to recode it in the subclass.

Abstract and Final Methods and Classes

The **ABSTRACT** and **FINAL** additions to the METHODS and CLASS statements allow you to define abstract and final methods or classes.

An abstract method is defined in an abstract class and cannot be implemented in that class. Instead, it is implemented in a subclass of the class. Abstract classes cannot be instantiated.

A final method cannot be redefined in a subclass. Final classes cannot have subclasses. They conclude an inheritance tree.

References to Subclasses and Polymorphism

Reference variables defined with reference to a superclass or an interface defined with reference to it can also contain references to any of its subclasses. Since subclasses contain all of the components of all of their superclasses, and given that the interfaces of methods cannot be changed, a reference variable defined with reference to a superclass or an interface implemented by a superclass can contain references to instances of any of its subclasses. In particular, you can define the target variable with reference to the generic class OBJECT.

When you create an object using the CREATE OBJECT statement and a reference variable typed with reference to a subclass, you can use the TYPE addition to create an instance of a subclass, to which the reference in the reference variable will then point.

A static user can use a reference variable to address the components visible to it in the superclass to which the reference variable refers. However, it cannot address any specialization implemented in the subclass. If you use a dynamic method call, you can address all components of the class.

If you redefine an instance method in one or more subclasses, you can use a single reference variable to call different implementations of the method, depending on the position in the inheritance tree at which the referenced object occurs. This concept that different classes can have the same interface and therefore be addressed using reference variables with a single type is called polymorphism.

Namespace for Components

Subclasses contain all of the components of all of their superclasses within the inheritance tree. Of these components, only the public and protected ones are visible. All public and protected components within an inheritance tree belong to the same namespace, and consequently must have unique names. The names of private components, on the other hand, must only be unique within their class.

When you redefine methods, the new implementation of the method obscures the method of the superclass with the same name. However, the new definition replaces the previous method implementation, so the name is still unique. You can use the pseudoreference SUPER-> to access a method definition in a superclass that has been obscured by a redefinition in a subclass.

Inheritance and Static Attributes

Like all components, static attributes only exist once in each inheritance tree. A subclass can access the public and protected static attributes of all of its superclasses. Conversely, a superclass shares its public and protected static attributes with all of its subclasses. In terms of inheritance, static attributes are not assigned to a single class, but to a part of the inheritance tree. You can change them from outside the class using the class component selector with any class name, or within any class in which they are shared. They are visible in all classes in the inheritance tree.

When you address a static attribute that belongs to part of an inheritance tree, you always address the class in which the attribute is declared, irrespective of the class you specify in the class selector. This is particularly important when you call the static constructors of classes in inheritance. Static constructors are executed the first time you address a class. If you address a static attribute declared in a superclass using the class name of a subclass, only the static constructor of the superclass is executed.

Inheritance and Constructors

There are special rules governing constructors in inheritance.

Instance Constructors

Every class has an instance constructor called CONSTRUCTOR. This is an exception to the rule that states that component names within an inheritance tree must be unique. However, the instance constructors of the various classes in an inheritance tree are fully independent of one another. You cannot redefine the instance constructor of a superclass in a subclass, neither can you call one specifically using the statement CALL METHOD CONSTRUCTOR. Consequently, no naming conflicts can occur.

The instance constructor of a class is called by the system when you instantiate the class using CREATE OBJECT. Since a subclass contains all of the visible attributes of its superclasses, which can also be set by instance constructors, the instance constructor of a subclass has to ensure that the instance constructors of all of its superclasses are also called. To do this, the instance constructor of each subclass must contain a CALL METHOD SUPER->CONSTRUCTOR statement. The only exception to this rule are direct subclasses of the root node OBJECT.

In superclasses without an explicitly-defined instance constructor, the implicit instance constructor is called. This automatically ensures that the instance constructor of the immediate superclass is called.

When you call an instance constructor, you must supply values for all of its non-optional interface parameters. There are various ways of doing this:

- Using CREATE OBJECT

  If the class that you are instantiating has an instance constructor with an interface, you must pass values to it using `EXPORTING`.

  If the class that you are instantiating has an instance constructor without an interface, you do not pass any parameters.

  If the class you are instantiating does not have an explicit instance constructor, you must look in the inheritance tree for the next-highest superclass with an explicit instance constructor. If this has an interface, you must supply values using `EXPORTING`. Otherwise, you do not have to pass any values.

Inheritance

- Using CALL METHOD SUPER->CONSTRUCTOR

  If the direct superclass has an instance constructor with an interface, you must pass values to it using `EXPORTING`.

  If the direct superclass has an instance constructor without an interface, you do not pass any parameters.

  If the direct superclass does not have an explicit instance constructor, you must look in the inheritance tree for the next-highest superclass with an explicit instance constructor. If this has an interface, you must supply values using `EXPORTING`. Otherwise, you do not have to pass any values.

In both CREATE OBJECT and CALL METHOD SUPER->CONSTRUCTOR, you must look at the next-available explicit instance constructor and, if it has an interface, pass values to it. The same applies to exception handling for instance constructors. When you work with inheritance, you need an precise knowledge of the entire inheritance tree. When you instantiate a class at the bottom of the inheritance tree, you may need to pass parameters to the constructor of a class that is much nearer the root node.

The instance constructor of a subclass is divided into two parts by the CALL METHOD SUPER->CONSTRUCTOR statement. In the statements before the call, the constructor behaves like a static method, that is, it cannot access the instance attributes of its class. You cannot address instance attributes until after the call. Use the statements before the call to determine the actual parameters for the interface of the instance constructor of the superclass. You can only use static attributes or local data to do this.

When you instantiate a subclass, the instance constructors are called hierarchically. The first nesting level in which you can address instance attributes is the highest-level superclass. When you return to the constructors of the lower-level classes, you can also successively address their instance attributes.

In a constructor method, the methods of the subclasses of the class are not visible. If an instance constructor calls an instance method of the same class using the implicit self-reference ME->, the method is called as it is implemented in the class of the instance constructor, and not in any redefined form that may occur in the subclass you want to instantiate. This is an exception to the rule that states that when you call instance methods, the system always calls the method as it is implemented in the class to whose instance the reference is pointing.

Static Constructors

Every class has a static constructor called CLASS_CONSTRUCTOR. As far as the namespace within an inheritance tree, the same applies to static constructors as to instance constructors.

The first time you address a subclass in a program, its static constructor is executed. However, before it can be executed, the static constructors of all of its superclasses must already have been executed. A static constructor may only be called once per program. Therefore, when you first address a subclass, the system looks for the next-highest superclass whose static constructor has not yet been executed. It executes the static constructor of that class, followed by those of all classes between that class and the subclass you addressed.

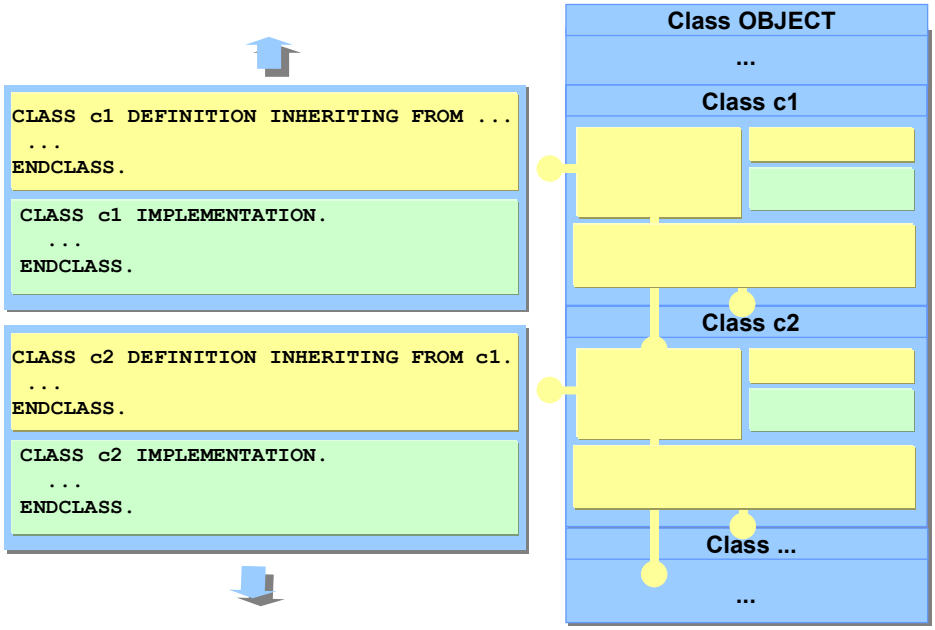
See also:

Overview Graphics [Page 1332]

Inheritance: Introductory Example [Page 1335]

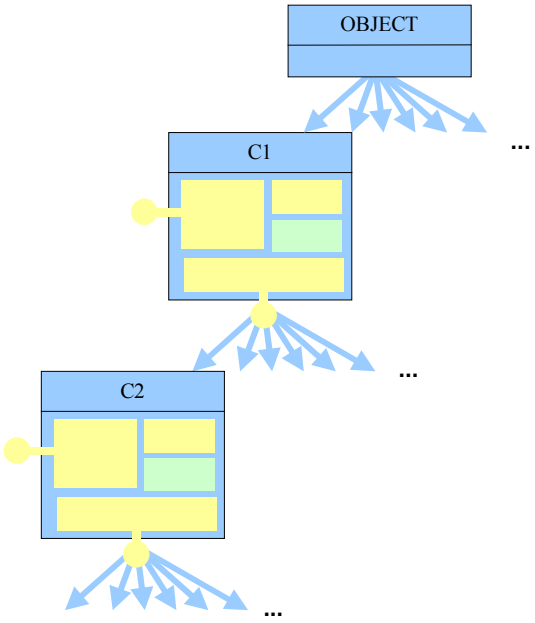
Inheritance: Overview Graphic

Inheritance: Overview



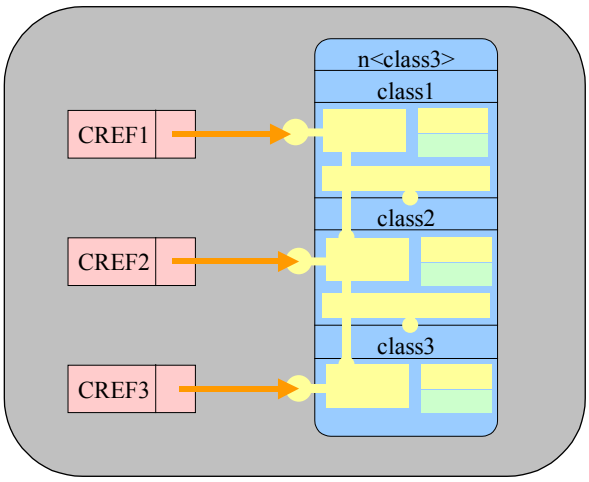
The left-hand part of the graphic shows how you can derive a subclass c2 from a superclass c1 using the INHERTING FROM addition in the CLASS statement. The right-hand part of the graphic shows the distribution of the subclass in the inheritance tree, which stretches back to the default empty class OBJECT. A subclass contains all of the components declared above it in the inheritance tree, and can address all of them that are declared public or protected.

Single Inheritance



This graphic illustrates single inheritance. A class may only have one direct superclass, but it can have more than one direct subclass. The empty class OBJECT is the root node of every inheritance tree in ABAP Objects.

Inheritance and Reference Variables



This graphic shows how reference variables defined with reference to a superclass can point to objects of subclasses. The object on the right is an instance of the class class3. The class reference variables CREF1, CREF2, and CREF3 are typed with reference to class1, class2, and

Inheritance: Overview Graphic

class3. All three can point to the object. However, CREF1 can only address the public components of class1. CREF2 can address the public components of class1 and class2. CREF3 can address the public components of all of the classes.

If you redefine a method of a superclass in a subclass, you can use a reference variable defined with reference to the superclass to address objects with different method implementations. When you address the superclass, the method has the original implementation, but when you address the subclass, the method has the new implementation. Using a single reference variable to call identically-named methods that behave differently is called polymorphism.

Inheritance: Introductory Example

The following simple example shows the principle of inheritance within ABAP Objects. It is based on the Simple Introduction to Classes [Page 1306]. A new class counter_ten inherits from the existing class counter.

```
REPORT demo_inheritance.

CLASS counter DEFINITION.
  PUBLIC SECTION.
    METHODS: set IMPORTING value(set_value) TYPE i,
             increment,
             get EXPORTING value(get_value) TYPE i.
  PROTECTED SECTION.
    DATA count TYPE i.
ENDCLASS.

CLASS counter IMPLEMENTATION.
  METHOD set.
    count = set_value.
  ENDMETHOD.
  METHOD increment.
    ADD 1 TO count.
  ENDMETHOD.
  METHOD get.
    get_value = count.
  ENDMETHOD.
ENDCLASS.

CLASS counter_ten DEFINITION INHERITING FROM counter.
  PUBLIC SECTION.
    METHODS increment REDEFINITION.
    DATA count_ten.
ENDCLASS.

CLASS counter_ten IMPLEMENTATION.
  METHOD increment.
    DATA modulo TYPE I.
    CALL METHOD super->increment.
    write / count.
    modulo = count mod 10.
    IF modulo = 0.
      count_ten = count_ten + 1.
      write count_ten.
    ENDIF.
  ENDMETHOD.
ENDCLASS.

DATA: count TYPE REF TO counter,
      number TYPE i VALUE 5.

START-OF-SELECTION.

  CREATE OBJECT count TYPE counter_ten.
```

```
CALL METHOD count->set EXPORTING set_value = number.

DO 20 TIMES.
  CALL METHOD count->increment.
ENDDO.
```

The class COUNTER_TEN is derived from COUNTER. It redefines the method INCREMENT. To do this, you must change the visibility of the COUNT attribute from PRIVATE to PROTECTED. The redefined method calls the obscured method of the superclass using the pseudoreference SUPER->. The redefined method is a specialization of the inherited method.

The example instantiates the subclass. The reference variable pointing to it has the type of the superclass. When the INCREMENT method is called using the superclass reference, the system executes the redefined method from the subclass.

Interfaces

Einführendes Beispiel zu Interfaces [Page 1341]

Classes, their instances (objects), and access to objects using reference variables form the basics of ABAP Objects. These means already allow you to model typical business applications, such as customers, orders, order items, invoices, and so on, using objects, and to implement solutions using ABAP Objects.

However, it is often necessary for similar classes to provide similar functions that are coded differently in each class but which should provide a uniform point of contact for the user. For example, you might have two similar classes, savings account and check account, both of which have a method for calculating end of year charges. The interfaces and names of the methods are the same, but the actual implementation is different. The user of the classes and their instances must also be able to run the end of year method for all accounts, without having to worry about the actual type of each individual account.

ABAP Objects makes this possible by using interfaces. Interfaces are independent structures that you can implement in a class to extend the scope of that class. The class-specific scope of a class is defined by its components and visibility sections. For example, the public components of a class define its public scope, since all of its attributes and method parameters can be addressed by all users. The protected components of a class define its scope with regard to its subclasses. (However, inheritance is not supported in Release 4.5B).

Interfaces extend the scope of a class by adding their own components to its public section. This allows users to address different classes via a universal point of contact. Interfaces, along with inheritance, provide one of the pillars of polymorphism, since they allow a single method within an interface to behave differently in different classes.

Defining Interfaces

Like classes, you can define interfaces either globally in the R/3 Repository or locally in an ABAP program. For information about how to define local interfaces, refer to the Class Builder [Ext.] section of the ABAP Workbench Tools documentation. The definition of a local interface <intf> is enclosed in the statements:

INTERFACE <intf>.

...

ENDINTERFACE.

The definition contains the declaration for all components (attributes, methods, events) of the interface. You can define the same components in an interface as in a class. The components of interfaces do not have to be assigned individually to a visibility section, since they automatically belong to the public section of the class in which the interface is implemented. Interfaces do not have an implementation part, since their methods are implemented in the class that implements the interface.

Implementing Interfaces

Unlike classes, interfaces do not have instances. Instead, interfaces are implemented by classes. To implement an interface in a class, use the statement

INTERFACES <intf>.

in the declaration part of the class. This statement may only appear in the public section of the class.

When you implement an interface in a class, the components of the interface are added to the other components in the public section. A component <icomp> of an interface <intf> can be addressed as though it were a member of the class under the name <intf~icomp>.

The class must implement the methods of all interfaces implemented in it. The implementation part of the class must contain a method implementation for each interface method <imeth>:

METHOD <intf~imeth>.

...

ENDMETHOD.

Interfaces can be implemented by different classes. Each of these classes is extended by the same set of components. However, the methods of the interface can be implemented differently in each class.

Interfaces allow you to use different classes in a uniform way using interface references (polymorphism). For example, interfaces that are implemented in different classes extend the public scope of each class by the same set of components. If a class does not have any class-specific public components, the interfaces define the entire public face of the class.

Interface References

Reference variables allow you to access objects (refer to Working with Objects [Page 1307]). Instead of creating reference variables with reference to a class, you can also define them with reference to an interface. This kind of reference variable can contain references to objects of classes that implement the corresponding interface.

To define an interface reference, use the addition TYPE REF TO <intf> in the TYPES or DATA statement. <intf> must be an interface that has been declared to the program before the actual reference declaration occurs. A reference variable with the type interface reference is called a interface reference variable, or interface reference for short.

An interface reference <iref> allows a user to use the form <iref>-><icomp> to address all visible interface components <icomp> of the object to which the object reference is pointing. It allows the user to access all of the components of the object that were added to its definition by the implementation of the interface.

Addressing Objects Using Interface References

To create an object of the class <class>, you must first have declared a reference variable <cref> with reference to the class. If the class <class> implements an interface <intf>, you can use the following assignment between the class reference variable <cref> and an interface reference <iref> to make the interface reference in <iref> point to the same object as the class reference in <cref>:

<iref> = <cref>

If the interface <intf> contains an instance attribute <attr> and an instance method <meth>, you can address the interface components as follows:

Using the **class reference variable** <cref>:

- To access an attribute <attr>: <cref>-><intf~attr>
- To call a method <meth>: `CALL METHOD <cref>-><intf~meth>`

Using the **interface reference variable** <iref>:

- To access an attribute <attr>: `<iref>-><attr>`
- To call a method <meth>: `CALL METHOD <iref>-><meth>`

As far as the static components of interfaces are concerned, you can only use the interface name to access constants:

Addressing a constant <const>: `<intf>=><const>`

For all other static components of an interface, you can only use object references or the class <class> that implements the interface:

Addressing a static attribute <attr>: `<class>=><intf~attr>`

Calling a static method <meth>: `CALL METHOD <class>=><intf~meth>`

Assignment Using Interface References - Casting

Like class references, you can assign interface references to different reference variables. You can also make assignments between class reference variables and interface reference variables. When you use the MOVE statement or the assignment operator (=) to assign reference variables, the system must be able to recognize in the syntax check whether an assignment is possible.

Suppose we have a class reference <cref> and interface references <iref>, <iref1>, and <iref2>. The following assignments with interface references can be checked statically:

- <iref1> = <iref2>

  Both interface references must refer to the same interface, or the interface of <iref1> must contain the interface <iref2> as a component.

- <iref> = <cref>

  The class of the class reference <cref> must implement the interface of the interface reference <iref>.

- <cref> = <iref>

  The class of <cref> must be the predefined empty class OBJECT.

In all other cases, you would have to work with the statement MOVE ...? TO or the **casting operator** (?=). The casting operator replaces the assignment operator (=). In the MOVE... ? TO statement, or when you use the casting operator, there is no static type check. Instead, the system checks at **runtime** whether the object reference in the source variable points to an object to which the object reference in the target variable can also point. If the assignment is possible, the system makes it, otherwise, the catchable runtime error MOVE_CAST_ERROR occurs.

You must always use casting for assigning an interface reference to a class reference if <cref> does not refer to the predefined empty class OBJECT:

<cref> ?= <iref>

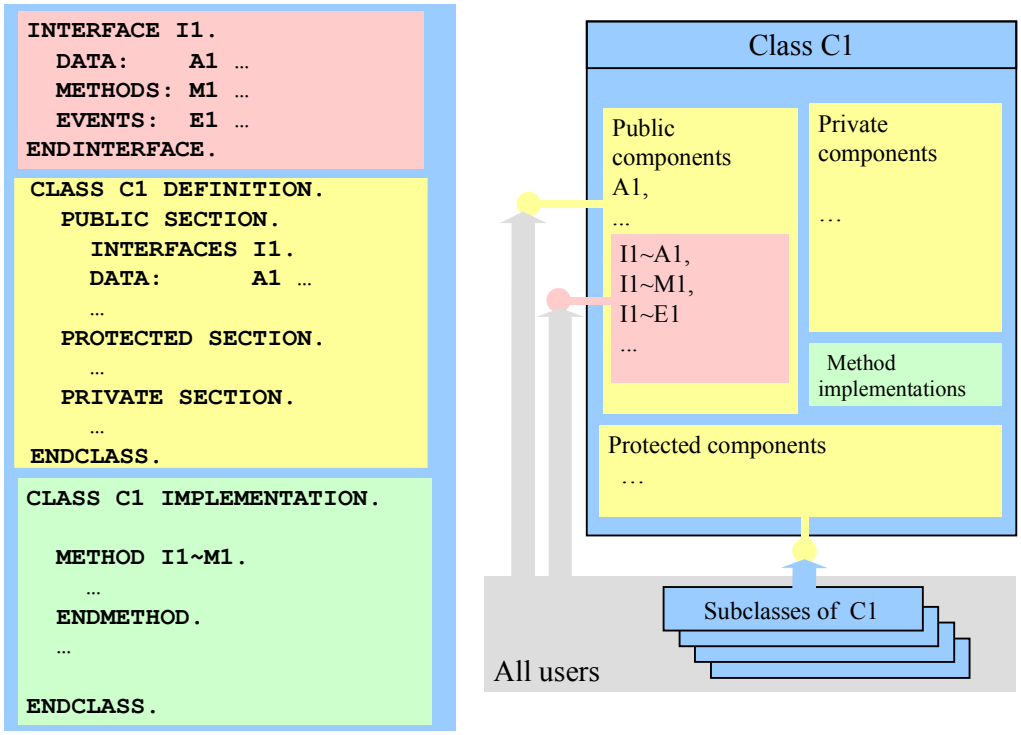
For the casting to be successful, the object to which <iref> points must be an object of the same class as the type of the class variable <cref>.

See also:

Overview Graphics [Page 1340]

Overview Graphics

Interfaces



[Ext.] [Ext.]

The left-hand side of the diagram shows the definition of a local interface I1 and the declaration and implementation parts of a local class C1 that implements the interface I1 in its **public section**. The interface method I1~M1 is implemented in the class. You cannot implement interfaces in the other visibility sections.

The right-hand side illustrates the structure of the class with the components in their respective visibility areas, and the implementation of the methods. The interface components extend the public scope of the class. All users can access the public components specific to the class and those of the interface.

Interfaces - Introductory Example

The following simple example shows how you can use an interface to implement two counters that are different, but can be addressed in the same way. See also the example in the Classes section.

```
INTERFACE I_COUNTER.
  METHODS: SET_COUNTER IMPORTING VALUE(SET_VALUE) TYPE I,
           INCREMENT_COUNTER,
           GET_COUNTER EXPORTING VALUE(GET_VALUE) TYPE I.
ENDINTERFACE.

CLASS C_COUNTER1 DEFINITION.
  PUBLIC SECTION.
    INTERFACES I_COUNTER.
  PRIVATE SECTION.
    DATA COUNT TYPE I.
ENDCLASS.

CLASS C_COUNTER1 IMPLEMENTATION.
  METHOD I_COUNTER~SET_COUNTER.
    COUNT = SET_VALUE.
  ENDMETHOD.
  METHOD I_COUNTER~INCREMENT_COUNTER.
    ADD 1 TO COUNT.
  ENDMETHOD.
  METHOD I_COUNTER~GET_COUNTER.
    GET_VALUE = COUNT.
  ENDMETHOD.
ENDCLASS.

CLASS C_COUNTER2 DEFINITION.
  PUBLIC SECTION.
    INTERFACES I_COUNTER.
  PRIVATE SECTION.
    DATA COUNT TYPE I.
ENDCLASS.

CLASS C_COUNTER2 IMPLEMENTATION.
  METHOD I_COUNTER~SET_COUNTER.
    COUNT = ( SET_VALUE / 10) * 10.
  ENDMETHOD.
  METHOD I_COUNTER~INCREMENT_COUNTER.
    IF COUNT GE 100.
      MESSAGE I042(00).
      COUNT = 0.
    ELSE.
      ADD 10 TO COUNT.
    ENDIF.
  ENDMETHOD.
  METHOD I_COUNTER~GET_COUNTER.
    GET_VALUE = COUNT.
```

```
  ENDMETHOD.
ENDCLASS.
```

The interface I_COUNTER contains three methods SET_COUNTER, INCREMENT_COUNTER, and GET_COUNTER. The classes C_COUNTER1 and C_COUNTER2 implement the interface in the public section. Both classes must implement the three interface methods in their implementation part. C_COUNTER1 is a class for counters that can have any starting value and are then increased by one. C_COUNTER2 is a class for counters that can only be increased in steps of 10. Both classes have an identical outward face. It is fully defined by the interface in both cases.

The following sections explain how a user can use an interface reference to address the objects of both classes:

 [Ext.]

Triggering and Handling Events

Übersichtsgrafiken zu Ereignissen [Page 1346]

Einführendes Beispiel zu Ereignissen [Page 1349]

Komplexes Beispiel zu Ereignissen [Page 1351]

In ABAP Objects, triggering and handling an event means that certain methods act as **triggers** and trigger events, to which other methods - the **handlers** - react. This means that the handler methods are executed when the event occurs.

This section contains explains how to work with events in ABAP Objects. For precise details of the relevant ABAP statements, refer to the corresponding keyword documentation in the ABAP Editor.

Triggering Events

To trigger an event, a class must

- Declare the event in its declaration part
- Trigger the event in one of its methods

Declaring Events

You declare events in the declaration part of a class or in an interface. To declare instance events, use the following statement:

EVENTS <evt> EXPORTING... VALUE(<e_i>) TYPE type [OPTIONAL]..

To declare static events, use the following statement:

CLASS-EVENTS <evt>...

Both statements have the same syntax.

When you declare an event, you can use the EXPORTING addition to specify parameters that are passed to the event handler. The parameters are always passed by value. Instance events always contain the implicit parameter SENDER, which has the type of a reference to the type or the interface in which the event is declared.

Triggering Events

An instance event in a class can be triggered by any method in the class. Static events can be triggered by any static method. To trigger an event in a method, use the following statement:

RAISE EVENT <evt> EXPORTING... <e_i> = <f_i>...

For each formal parameter <e_i> that is not defined as optional, you must pass a corresponding actual parameter <f_i> in the EXPORTING addition. The self-reference ME is automatically passed to the implicit parameter SENDER.

Handling Events

Events are handled using special methods. To handle an event, a method must

- be defined as an event handler method for that event
- be registered at runtime for the event.

Declaring Event Handler Methods

Any class can contain event handler methods for events from other classes. You can, of course, also define event handler methods in the same class as the event itself. To declare an event handler method, use the following statement:

METHODS <meth> FOR EVENT <evt> OF <cif> IMPORTING.. <e_i>..

for an instance method. For a static method, use CLASS-METHODS instead of METHODS. <evt> is an event declared in the class or interface <cif>.

The interface of an event handler method may only contain formal parameters defined in the declaration of the event <evt>. The attributes of the parameter are also adopted by the event. The event handler method does not have to use all of the parameters passed in the RAISE EVENT statement. If you want the implicit parameter SENDER to be used as well, you must list it in the interface. This parameter allows an instance event handler to access the trigger, for example, to allow it to return results.

If you declare an event handler method in a class, it means that the instances of the class or the class itself are, in principle, able to handle an event <evt> triggered in a method.

Registering Event Handler Methods

To allow an event handler method to react to an event, you must determine at runtime the trigger to which it is to react. You can do this with the following statement:

SET HANDLER... <h_i>... [FOR]...

It links a list of handler methods with corresponding trigger methods. There are four different types of event.

It can be

- An instance event declared in a class
- An instance event declared in an interface
- A static event declared in a class
- A static event declared in an interface

The syntax and effect of the SET HANDLER depends on which of the four cases listed above applies.

For an instance event, you must use the FOR addition to specify the instance for which you want to register the handler. You can either specify a single instance as the trigger, using a reference variable <ref>:

SET HANDLER... <h_i>...FOR <ref>.

or you can register the handler for all instances that can trigger the event:

SET HANDLER... <h_i>...FOR ALL INSTANCES.

The registration then applies even to triggering instances that have not yet been created when you register the handler.

You cannot use the FOR addition for static events:

SET HANDLER... <h_i>...

The registration applies automatically to the whole class, or to all of the classes that implement the interface containing the static event. In the case of interfaces, the registration also applies to classes that are not loaded until after the handler has been registered.

Timing of Event Handling

After the RAISE EVENT statement, all registered handler methods are executed before the next statement is processed (synchronous event handling). If a handler method itself triggers events, its handler methods are executed before the original handler method continues. To avoid the possibility of endless recursion, events may currently only be nested 64 deep.

Handler methods are executed in the order in which they were registered. Since event handlers are registered dynamically, you should not assume that they will be processed in a particular order. Instead, you should program as though all event handlers will be executed simultaneously.

Overview Graphic

Suppose we have two classes, C1 and C2:

Event trigger

```
CLASS C1 DEFINITION.

 PUBLIC SECTION.
   EVENTS E1
     EXPORTING VALUE(P1)
                   TYPE I.
   METHODS M1.

   PRIVATE SECTION.
     DATA A1 TYPE I.

ENDCLASS.
```

```
CLASS C1 IMPLEMENTATION.

  METHOD M1.
    A1 = ...
    RAISE EVENT E1
      EXPORTING P1 = A1.
  ENDMETHOD.

ENDCLASS.
```

Event handler

```
CLASS C2 DEFINITION.

 PUBLIC SECTION.
   METHODS: M2
     FOR EVENT E1 OF C1
             IMPORTING P1.

   PRIVATE SECTION.
     DATA A2 TYPE I.

ENDCLASS.
```

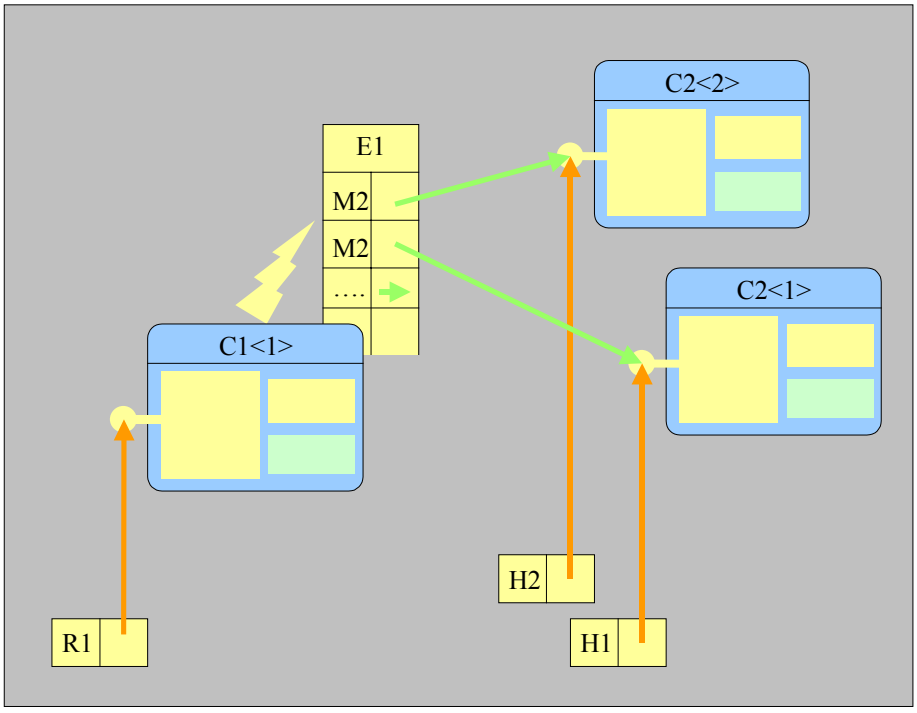
```
CLASS C2 IMPLEMENTATION.

  METHOD M2.
    A2 = P1.
    ...

  ENDMETHOD.

ENDCLASS.
```

The class C1 contains an event E1, which is triggered by the method M1. Class C2 contains a method M2, which can handle event E1 of class C1.

The following diagram illustrates handler registration:

Registering handlers

```
DATA: R1 TYPE REF TO C1,
      H1 TYPE REF TO C2,
      H2 TYPE REF TO C2.
CREATE OBJECT: R1,
               H1,
               H2.

SET HANDLER H1->M2
            H2->M2 FOR R1.
CALL METHOD R1->M1.
```



The program creates an instance of the class C1 and two instances of the class C2. The values of the reference variables R1, H1, and H2 point to these instances.

The SET HANDLER statement creates a handler table, invisible to the user, for each event for which a handler method has been registered.

The handler table contains the names of the handler methods and **references** to the registered instances. The entries in the table are administered dynamically by the SET HANDLER statement. A reference to an instance in a handler table is like a reference in a reference

Overview Graphic

variable. In other words, it counts as a use of the instance, and therefore directly affects its lifetime. In the above diagram, this means that the instances C2<1> and C2<2> are not deleted by the garbage collection, even if H1 and H2 are initialized, so long as their registration is not deleted from the handler table.

For static events, the system creates an instance-independent handler table for the relevant class.

When an event is triggered, the system looks in the corresponding event table and executes the methods in the appropriate instances (or in the corresponding class for a static handler method).

Events: Introductory Example

The following simple example shows the principle of events within ABAP Objects. It is based on the Simple Introduction to Classes [Page 1306]. An event critical_value is declared and triggered in class counter.

```
REPORT demo_class_counter_event.

CLASS counter DEFINITION.
  PUBLIC SECTION.
    METHODS increment_counter.
    EVENTS  critical_value EXPORTING value(excess) TYPE i.
  PRIVATE SECTION.
    DATA: count     TYPE i,
          threshold TYPE i VALUE 10.
ENDCLASS.

CLASS counter IMPLEMENTATION.
  METHOD increment_counter.
    DATA diff TYPE i.
    ADD 1 TO count.
    IF count > threshold.
      diff = count - threshold.
      RAISE EVENT critical_value EXPORTING excess = diff.
    ENDIF.
  ENDMETHOD.
ENDCLASS.

CLASS handler DEFINITION.
  PUBLIC SECTION.
    METHODS handle_excess
            FOR EVENT critical_value OF counter
            IMPORTING excess.
ENDCLASS.

CLASS handler IMPLEMENTATION.
  METHOD handle_excess.
    WRITE: / 'Excess is', excess.
  ENDMETHOD.
ENDCLASS.

DATA: r1 TYPE REF TO counter,
      h1 TYPE REF TO handler.

START-OF-SELECTION.

  CREATE OBJECT: r1, h1.

  SET HANDLER h1->handle_excess FOR ALL INSTANCES.

  DO 20 TIMES.
    CALL METHOD r1->increment_counter.
  ENDDO.
```

Events: Introductory Example

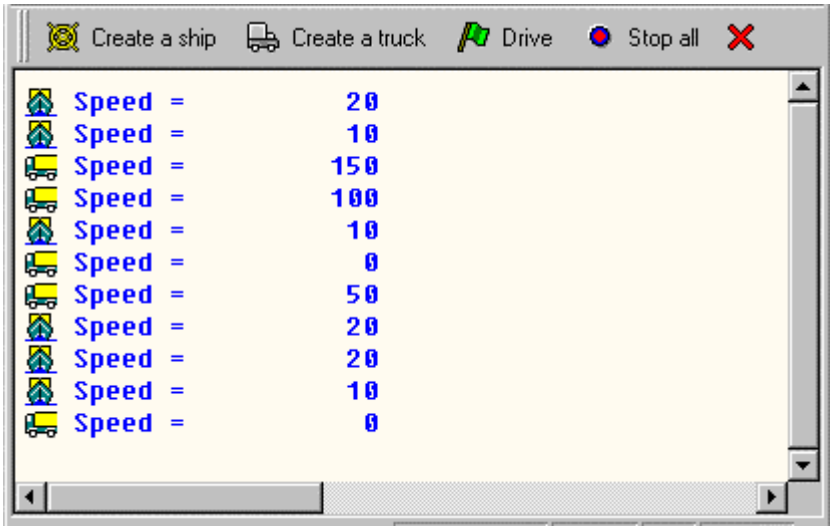
The class COUNTER implements a counter. It triggers the event CRITICAL_VALUE when a threshold value is exceeded, and displays the difference. HANDLER can handle the exception in COUNTER. At runtime, the handler is registered for all reference variables that point to the object.

Events in ABAP Objects - Example

The following example shows how to declare, call, and handle events in ABAP Objects.

Overview

This object works with the interactive list displayed below. Each user interaction triggers an event in ABAP Objects. The list and its data is created in the class C_LIST. There is a class STATUS for processing user actions. It triggers an event BUTTON_CLICKED in the AT USER-COMMAND event. The event is handled in the class C_LIST. It contains an object of the class C_SHIP or C_TRUCK for each line of the list. Both of these classes implement the interface I_VEHICLE. Whenever the speed of one of these objects changes, the event SPEED_CHANGE is triggered. The class C_LIST reacts to this and updates the list.



Constraints

The ABAP statements used for list processing are not yet fully available in ABAP Objects. However, to produce a simple test output, you can use the following statements:

- WRITE [AT] /<offset>(<length>) <f>
- ULINE
- SKIP
- NEW-LINE

Note: The behavior of formatting and interactive list functions in their current state are not guaranteed. Incompatible changes could occur in a future release.

Declarations

Events in ABAP Objects - Example

This example is implemented using **local** interfaces and classes. Below are the declarations of the interfaces and classes:

```
*********************************************************************
* Interface and Class declarations
*********************************************************************
INTERFACE I_VEHICLE.
  DATA     MAX_SPEED TYPE I.
  EVENTS SPEED_CHANGE EXPORTING VALUE(NEW_SPEED) TYPE I.
  METHODS: DRIVE,
           STOP.
ENDINTERFACE.
*---------------------------------------------------------------------
CLASS C_SHIP DEFINITION.
  PUBLIC SECTION.
  METHODS CONSTRUCTOR.
  INTERFACES I_VEHICLE.
  PRIVATE SECTION.
  ALIASES MAX FOR I_VEHICLE~MAX_SPEED.
  DATA SHIP_SPEED TYPE I.
ENDCLASS.
*---------------------------------------------------------------------
CLASS C_TRUCK DEFINITION.
  PUBLIC SECTION.
  METHODS CONSTRUCTOR.
  INTERFACES I_VEHICLE.
  PRIVATE SECTION.
  ALIASES MAX FOR I_VEHICLE~MAX_SPEED.
  DATA TRUCK_SPEED TYPE I.
ENDCLASS.
*---------------------------------------------------------------------
CLASS STATUS DEFINITION.
  PUBLIC SECTION.
  CLASS-EVENTS BUTTON_CLICKED EXPORTING VALUE(FCODE) LIKE SY-UCOMM.
  CLASS-METHODS: CLASS_CONSTRUCTOR,
                 USER_ACTION.
ENDCLASS.
*---------------------------------------------------------------------
```

```
CLASS C_LIST DEFINITION.

  PUBLIC SECTION.

  METHODS: FCODE_HANDLER FOR EVENT BUTTON_CLICKED OF STATUS
                         IMPORTING FCODE,
           LIST_CHANGE   FOR EVENT SPEED_CHANGE OF I_VEHICLE
                         IMPORTING NEW_SPEED,
           LIST_OUTPUT.

  PRIVATE SECTION.

  DATA: ID TYPE I,
        REF_SHIP  TYPE REF TO C_SHIP,
        REF_TRUCK TYPE REF TO C_TRUCK,

        BEGIN OF LINE,
          ID TYPE I,
          FLAG,
          IREF  TYPE REF TO I_VEHICLE,
          SPEED TYPE I,
        END OF LINE,
        LIST LIKE SORTED TABLE OF LINE WITH UNIQUE KEY ID.

ENDCLASS.
*********************************************************************
```

The following events are declared in the example:

- The instance event SPEED_CHANGE in the interface I_VEHICLE
- The static event BUTTON_CLICKED in the class STATUS.

The class C_LIST contains event handler methods for both events.

Note that the class STATUS does not have any attributes, and therefore only works with static methods and events.

Implementations

Below are the implementations of the methods of the above classes:

```
*********************************************************************
* Implementations
*********************************************************************

CLASS C_SHIP IMPLEMENTATION.

  METHOD CONSTRUCTOR.
    MAX = 30.
  ENDMETHOD.

  METHOD I_VEHICLE~DRIVE.
    CHECK SHIP_SPEED < MAX.
    SHIP_SPEED = SHIP_SPEED + 10.
    RAISE EVENT I_VEHICLE~SPEED_CHANGE
                EXPORTING NEW_SPEED = SHIP_SPEED.
  ENDMETHOD.
```

Events in ABAP Objects - Example

```
  METHOD I_VEHICLE~STOP.
    CHECK SHIP_SPEED > 0.
    SHIP_SPEED = 0.
    RAISE EVENT I_VEHICLE~SPEED_CHANGE
                EXPORTING NEW_SPEED = SHIP_SPEED.
  ENDMETHOD.

ENDCLASS.

*---------------------------------------------------------------------

CLASS C_TRUCK IMPLEMENTATION.

  METHOD CONSTRUCTOR.
    MAX = 150.
  ENDMETHOD.

  METHOD I_VEHICLE~DRIVE.
    CHECK TRUCK_SPEED < MAX.
    TRUCK_SPEED = TRUCK_SPEED + 50.
    RAISE EVENT I_VEHICLE~SPEED_CHANGE
                EXPORTING NEW_SPEED = TRUCK_SPEED.
  ENDMETHOD.

  METHOD I_VEHICLE~STOP.
    CHECK TRUCK_SPEED > 0.
    TRUCK_SPEED = 0.
    RAISE EVENT I_VEHICLE~SPEED_CHANGE
                EXPORTING NEW_SPEED = TRUCK_SPEED.
  ENDMETHOD.

ENDCLASS.

*---------------------------------------------------------------------

CLASS STATUS IMPLEMENTATION.

  METHOD CLASS_CONSTRUCTOR.
    SET PF-STATUS 'VEHICLE'.
    WRITE 'Click a button!'.
  ENDMETHOD.

  METHOD USER_ACTION.
    RAISE EVENT BUTTON_CLICKED EXPORTING FCODE = SY-UCOMM.
  ENDMETHOD.

ENDCLASS.

*---------------------------------------------------------------------

CLASS C_LIST IMPLEMENTATION.

  METHOD FCODE_HANDLER.
    CLEAR LINE.
    CASE FCODE.
      WHEN 'CREA_SHIP'.
        ID = ID + 1.
        CREATE OBJECT REF_SHIP.
        LINE-ID = ID.
        LINE-FLAG = 'C'.
```

```
        LINE-IREF = REF_SHIP.
        APPEND LINE TO LIST.
      WHEN 'CREA_TRUCK'.
        ID = ID + 1.
        CREATE OBJECT REF_TRUCK.
        LINE-ID = ID.
        LINE-FLAG = 'T'.
        LINE-IREF = REF_TRUCK.
        APPEND LINE TO LIST.
      WHEN 'DRIVE'.
        CHECK SY-LILLI > 0.
        READ TABLE LIST INDEX SY-LILLI INTO LINE.
        CALL METHOD LINE-IREF->DRIVE.
      WHEN 'STOP'.
        LOOP AT LIST INTO LINE.
          CALL METHOD LINE-IREF->STOP.
        ENDLOOP.
      WHEN 'CANCEL'.
        LEAVE PROGRAM.
    ENDCASE.
    CALL METHOD LIST_OUTPUT.
  ENDMETHOD.

  METHOD LIST_CHANGE.
    LINE-SPEED = NEW_SPEED.
    MODIFY TABLE LIST FROM LINE.
  ENDMETHOD.

  METHOD LIST_OUTPUT.
    SY-LSIND = 0.
    SET TITLEBAR 'TIT'.
    LOOP AT LIST INTO LINE.
      IF LINE-FLAG = 'C'.
        WRITE / ICON_WS_SHIP AS ICON.
      ELSEIF LINE-FLAG = 'T'.
        WRITE / ICON_WS_TRUCK AS ICON.
      ENDIF.
      WRITE: 'Speed = ', LINE-SPEED.
    ENDLOOP.
  ENDMETHOD.

ENDCLASS.

*****************************************************************
```

The static method USER_ACTION of the class STATUS triggers the static event BUTTON_CLICKED. The instance methods I_VEHICLE~DRIVE and I_VEHICLE~STOP trigger the instance event I_VEHICLE~SPEED_CHANGE in the classes C_SHIP and C_TRUCK.

Using the Classes in a Program

The following program uses the above classes:

```
REPORT OO_EVENTS_DEMO NO STANDARD PAGE HEADING.
```

Events in ABAP Objects - Example

```
*********************************************************************
* Global data of program
*********************************************************************

DATA LIST TYPE REF TO C_LIST.

*********************************************************************
* Program events
*********************************************************************

START-OF-SELECTION.

  CREATE OBJECT LIST.

  SET HANDLER: LIST->FCODE_HANDLER,
               LIST->LIST_CHANGE FOR ALL INSTANCES.

*---------------------------------------------------------------------

AT USER-COMMAND.

  CALL METHOD STATUS=>USER_ACTION.

*********************************************************************
```

The program creates an object of the class C_LIST and registers the event handler method FCODE_HANDLER of the object for the class event BUTTON_CLICKED, and the event handler method LIST_CHANGE for the event SPEED_CHANGE of all instances that implement the interface I_VEHICLE.

Class Pools

This section discusses the structure and special features of class pools.

Global Classes and Interfaces

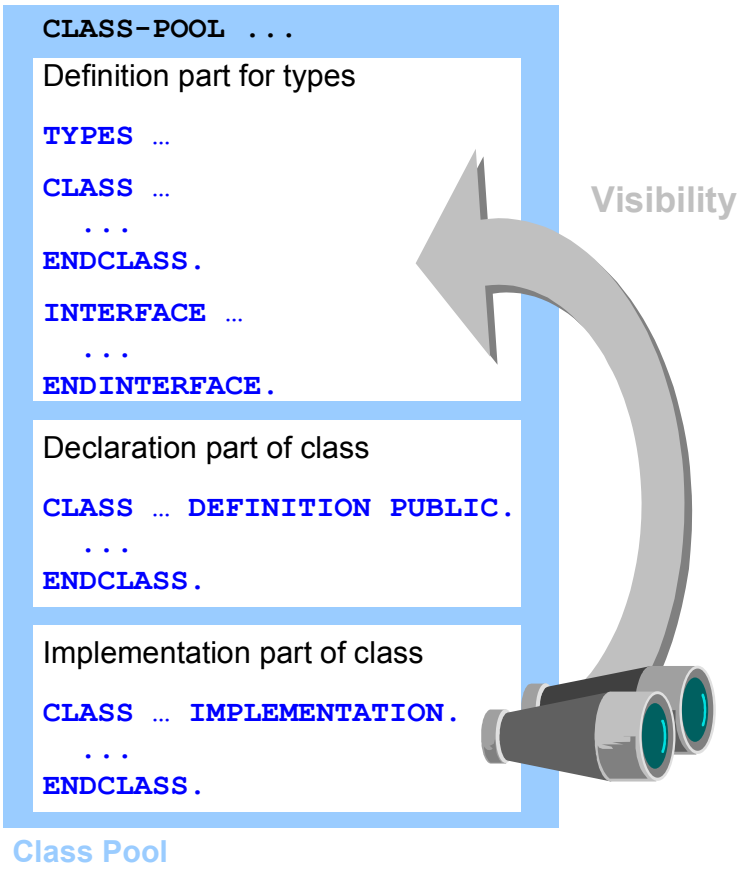
Classes and interfaces are both object types. You can define them either globally in the R/3 Repository or locally in an ABAP program. If you define classes and interfaces globally, they are stored in special ABAP programs called class pools (type K) or interface pools (type J), which serve as containers for the respective object types. Each class or interface pool contains the definition of a single class or interface. The programs are automatically generated by the Class Builder when you create a class or interface.

A class pool is comparable to a module pool or function group. It contains both declarative and executable ABAP statements, but cannot be started on its own. Instead, the system can only execute the statements in the class pool on request, that is, when the CREATE OBJECT statement occurs to create instances of the class.

Interface pools do not contain any executable statements. Instead, they are used as containers for interface definitions. When you implement an interface in a class, the interface definition is implicitly included in the class definition.

Structure of a Class Pool

Class pools are structured as follows:



Class pools contain a definition part for type declarations, and the declaration and implementation parts of the class.

Differences From Other ABAP Programs

Class pools are different from other ABAP programs for the following reasons:

- ABAP programs such as executable programs, module pools, or function modules, usually have a declaration part in which the global data for the program is defined. This data is visible in all of the processing blocks in the program. Class pools, on the other hand, have a definition part, in which you can define data and object types, but **no** data objects or field symbols. The types that you define in a class pool are **only** visible in the implementation part of the global class.
- The only processing blocks that you can use are the declaration part and implementation part of the global class. The implementation part may only implement the methods declared in the global class. You **cannot** use any of the other ABAP processing blocks (dialog modules, event blocks, subroutines, function modules).
- The processing blocks of class pools are not controlled by the ABAP runtime environment. No events occur, and you cannot call any dialog modules or procedures. Class pools serve exclusively for class programming. You can only access the data and functions of a class using its **interface**.

- Since events and dialog modules are not permitted in classes, you cannot process screens in classes. You cannot program lists and selection screens in classes, since they cannot react to the appropriate events. It is intended to make screens available in classes. Instead of dialog modules, it will be possible to call methods of the class from the screen flow logic.

Local Classes in Class Pools

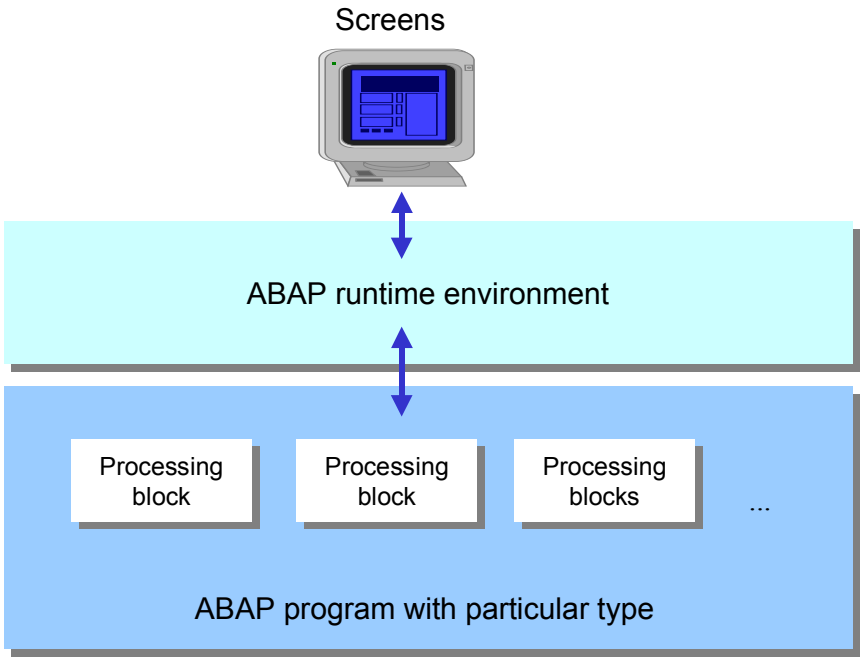
The classes and interfaces that you define in the definition part of a class pool are not visible externally. Within the class pool, they have a similar function to local classes and interfaces in other ABAP programs. Local classes can only be instantiated in the methods of the global class. Since subroutines are not allowed in class pools, local classes are the only possible modularization unit in global classes. Local classes have roughly the same function for global classes as subroutines in function groups, but with the significant exception that they are not visible externally.

Appendix

Programs, Screens, and Processing Blocks

This section contains a summary of possible ABAP programs, their screens, and their processing blocks.

Overview



ABAP programs consist of processing blocks, and can contain screens as components. Both processing blocks and screens are controlled by the ABAP runtime environment.

ABAP Programs

ABAP has the following program types:

- **Executable Program**

  Type 1; introduced with the REPORT statement; can be started by entering the program name or using a transaction code; can be called using SUBMIT; can have its own screens.

- **Module Pool**

  Type M; introduced with the PROGRAM statement; can be started using a transaction code; can be called using CALL TRANSACTION or LEAVE TO TRANSACTION; have their own screens.

- **Function Group**

  Type F; introduced with the FUNCTION-POOL statement; non-executable; container for function modules; can have its own screens.

- **Class Definition**

Type K; introduced with the CLASS-POOL statement; non-executable; container for classes; cannot (currently) have its own screens.

- **Interface Definition**

  Type J; introduced with the CLASS-POOL statement; non-executable; container for interfaces; cannot have its own screens.

- **Subroutine Pool**

  Type S; introduced with the PROGRAM statement; non-executable; container for subroutines; cannot have its own screens.

- **Type Groups**

  Type T; introduced with the TYPE-POOL statement; non-executable; container for type definitions; cannot have its own screens.

- **Include Program**

  Type I; no introductory statement; non-executable; container for source code modules.

Non-executable programs with types F, K, J, S, and T are loaded into memory as required. Include programs are not separately-compiled units; their source code is inserted into the programs in which the corresponding include statement occurs.

Screens

ABAP programs with types 1, M, or F can contain and process the following types of screen:

- **Screens**

  Defined using the Screen Painter; can be combined into screen sequences; called using CALL SCREEN or a transaction code; processed in dialog modules of the corresponding ABAP program.

- **Selection Screen**

  Defined within an ABAP program; called by the runtime environment or using the CALL SELECTION-SCREEN statement; processed in event blocks of the corresponding ABAP program.

- **Lists**

  Defined within an ABAP program; called by the runtime environment; processed in event blocks of the corresponding ABAP program.

Class definitions (type K programs) do not yet support screens. Subroutine pools (type S programs) cannot contain their own screens.

Processing Blocks

All ABAP programs are made up of processing blocks. You cannot nest processing blocks. When a program is executed, its processing blocks are called. All of the statements in an ABAP program, apart from its global data declarations, belong to a processing block.

ABAP contains the following processing blocks:

Dialog Module

Defined between the MODULE...ENDMODULE statements in type 1, M, and F programs; has no local data area and no parameter interface; called using the MODULE statement in screen flow logic; processes screens.

Event Block

Defined by one of the event key words; no local data area and no parameter interface; reacts to events in the ABAP runtime environment. (Exceptions: AT SELECTION-SCREEN and GET are implemented internally using subroutines, and have a local data area).

We differentiate between:

- **Reporting events**

  INITIALIZATION

  START-OF-SELECTION

  GET

  END-OF-SELECTION

  Called by the ABAP runtime environment while a type 1 program is running; contain application logic for report programs.

- **Selection screen events**

  AT SELECTION-SCREEN OUTPUT

  AT SELECTION-SCREEN ON VALUE REQUEST

  AT SELECTION-SCREEN ON HELP REQUEST

  AT SELECTION-SCREEN ON <f>

  AT SELECTION-SCREEN ON BLOCK

  AT SELECTION-SCREEN ON RADIOBUTTON GROUP

  AT SELECTION SCREEN

  AT SELECTION SCREEN ON END OF <f>

  Called by the ABAP runtime environment following a user action on a selection screen in a type 1, M, or F program; process selection screens.

- **List events**

  TOP-OF-PAGE

  END-OF-PAGE

  AT LINE-SELECTION

  AT PF<nn>

  AT USER-COMMAND

  Called by the ABAP runtime environment while a list is being created or after a user action on a list in a type 1, M, or F program.

Procedures

ABAP contains the following procedures. They have a local data area and a parameter interface:

- **Subroutines**

  Defined by FORM...ENDFORM in any program except for type K; called using the PERFORM statement in any ABAP program.

- **Function modules**

  Defined by FUNCTION...ENDFUNCTION in type F programs; called using CALL FUNCTION from any ABAP program.

- **Methods**

  Defined by METHOD...ENDMETHOD in global classes in programs with type K, or in local classes in any ABAP program; called using CALL METHOD from any ABAP program for global classes, and, for local classes, from the program in which the class is defined.

Introductory Statements for Programs

Each ABAP program type has a statement that introduces programs of that type:

Program type	Introductory statement
1	REPORT
M	PROGRAM
F	FUNCTION-POOL
K	CLASS-POOL
J	CLASS-POOL
S	PROGRAM
T	TYPE-POOL
I	-

Include programs (type I) are not compilation units. Instead, they are purely modularization units that are only ever used in the context of the programs to which they belong. For this reason, include programs do not have a special introductory statement.

The following sections describe the function of introductory statements:

REPORT and PROGRAM

The REPORT and PROGRAM statements currently have the same function. They allow you to specify the message class of the program and the formatting options for its default list. Whether a program is executable or can only be started using a transaction code depends exclusively on the program type and not on the statement that introduces it. However, executable programs should always begin with a REPORT statement, and module pools always with a PROGRAM statement. Subroutine pools (type S programs) should also always begin with a PROGRAM statement.

FUNCTION-POOL

The introductory statement FUNCTION-POOL declares a program in which you can define function modules. At runtime, function pool programs are loaded in to a new program group [Page 494] with their own user dialogs and their own shared data areas in the internal session of the calling program. For this reason, function groups (type F programs) must always begin with a FUNCTION-POOL statement. This is usually generated by the Function Builder. Type 1, M, or S programs should not begin with a FUNCTION-POOL statement, since they would then not share common data areas with the caller. However, in exceptional cases, you can introduce a type 1 or type M program with FUNCTION-POOL to ensure that externally-called subroutines can process their own screens. As in the REPORT and PROGRAM statements, you can specify the message class and standard list formatting options of the program in the FUNCTION-POOL statement.

CLASS-POOL

The introductory statement CLASS-POOL can only be used for class or interface definitions (type K or J programs). A program introduced with the CLASS-POOL statement can only contain global type definitions and definitions of classes and interfaces. The CLASS-POOL statement is

generated automatically where required by the Class Builder - you should not insert it into programs manually.

TYPE-POOL

The introductory statement TYPE-POOL can only be used for type groups (type T programs). A program introduced with the TYPE-POOL statement can only contain global type definitions and constants declarations. The CLASS-POOL statement is generated automatically where required by the ABAP Dictionary - you should not insert it into programs manually.

Overview of ABAP Calls

ABAP programs can contain calls to various other program units. The units can be classified according to the context in which they run when called, and according to the type of unit.

Call contexts:

- Internal Calls [Page 1369]
- External Procedure Calls [Page 1371]
- External Program Calls [Page 1373]

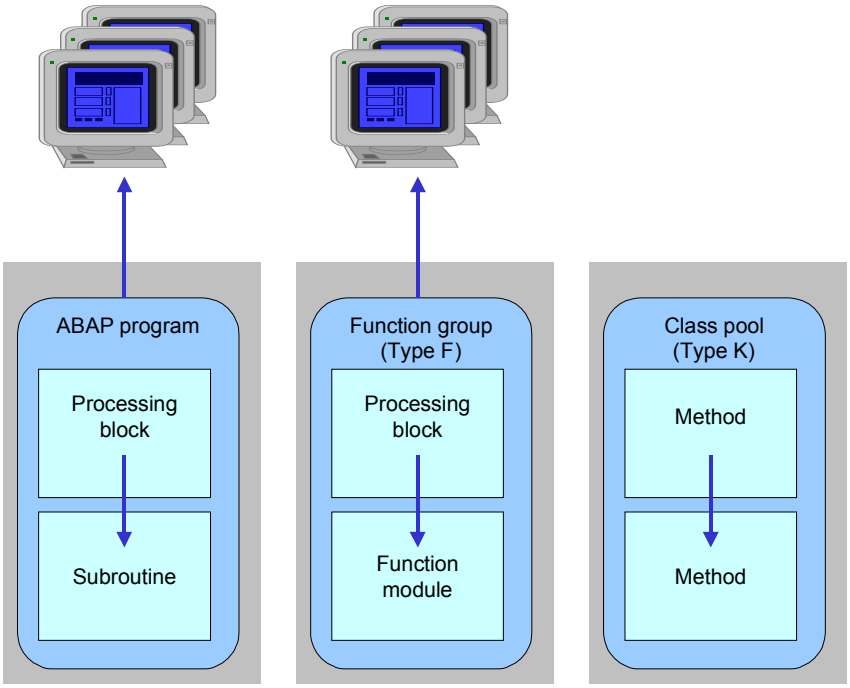
Callable units:

- ABAP Programs [Page 1376]
- Procedures [Page 1378]
- Screens and Screen Sequences [Page 1380]

Call Contexts

Internal Calls

In an internal call, the called unit is part of the calling program. The source code of the called unit is either part of the source code of the calling program or attached to it as an include program. This means that, at runtime, **no** extra programs have to be loaded into the internal session of the calling program. You can use internal calls for procedures and subscreens.



Internal sessions

Procedures

All procedures [Page 1378] - subroutines, function modules, and methods - can be called internally in the program in which they are defined.

- Using an internal call is the **recommended** and most frequently used method of calling a subroutine.
- Function modules can only be called internally if a function module calls another function module from the same group. Otherwise, function modules are called externally.
- Methods are called internally when a class calls one of its own methods, or when you use a method of a local class in an ABAP program.

Screens

All of the screens [Page 1380] belonging to a program can be called internally. When you call a screen, a screen sequence begins, in which the next screen is defined by the next screen specified in the current screen.

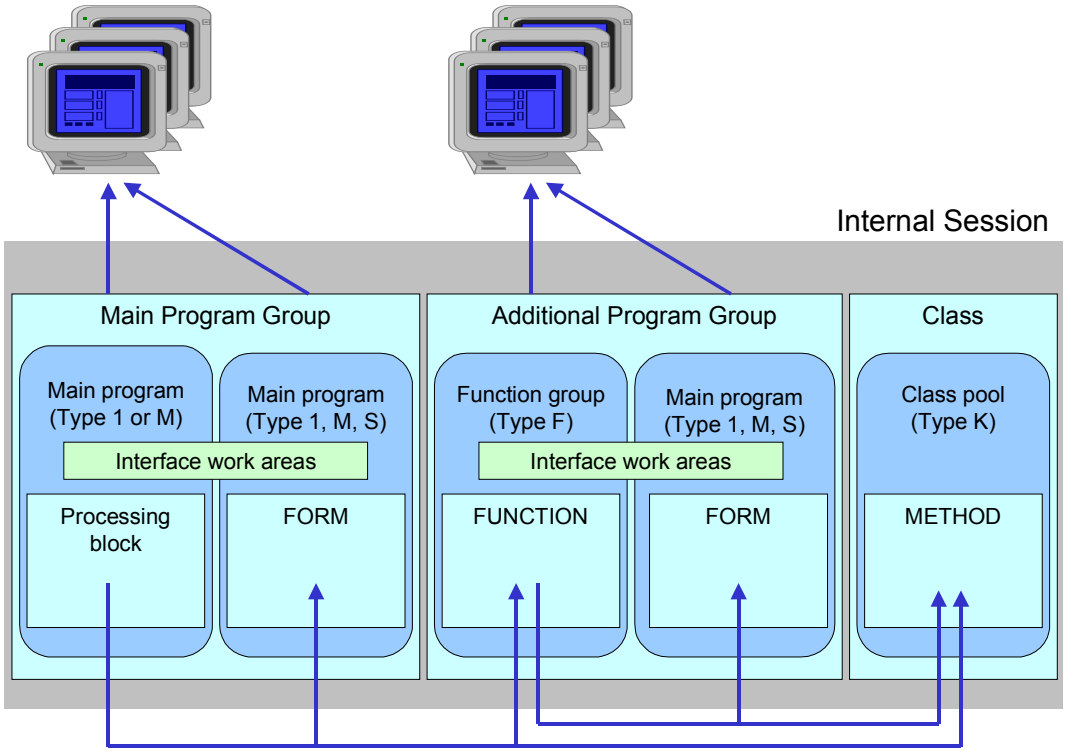
Internal Calls

The special screen types selection screen and list are only ever called internally. You should only call subscreens internally.

External Procedure Calls

In an external procedure call, the called unit is **not** part of the calling program. This means that, at runtime, an extra program has to be loaded **into the internal session of the calling program**. You can use external calls for procedures and subscreens.

The additional loaded program is an instance with its own global data area. When you call an external subroutine, the calling program and the loaded program form a single program group, unless the loaded program is a function group. Function groups always form their own program group. Furthermore, when you call methods externally, they and their class form their own program group. Within a program group, interface work areas [Page 130] and the screens of the calling program are shared. Classes cannot contain interface work areas.



Procedures

All procedures [Page 1378], that is, subroutines, function modules, and methods, can be called externally from any ABAP program.

- You are recommended not to use external calls for subroutines, in particular if the subroutines share interface work areas [Page 130] with the calling program and call their own screens. The program group into which the main program of a subroutine is loaded depends on the sequence of the calls. The sequence is often not statically defined, but changes depending on user actions or the contents of fields. For this reason, it may not be clear which interface work areas [Page 130] and screens will be used by an externally-called subroutine.

External Procedure Calls

- Function modules are intended for external procedure calls.
- Accessing an externally-visible method of a global class counts as an external procedure call.

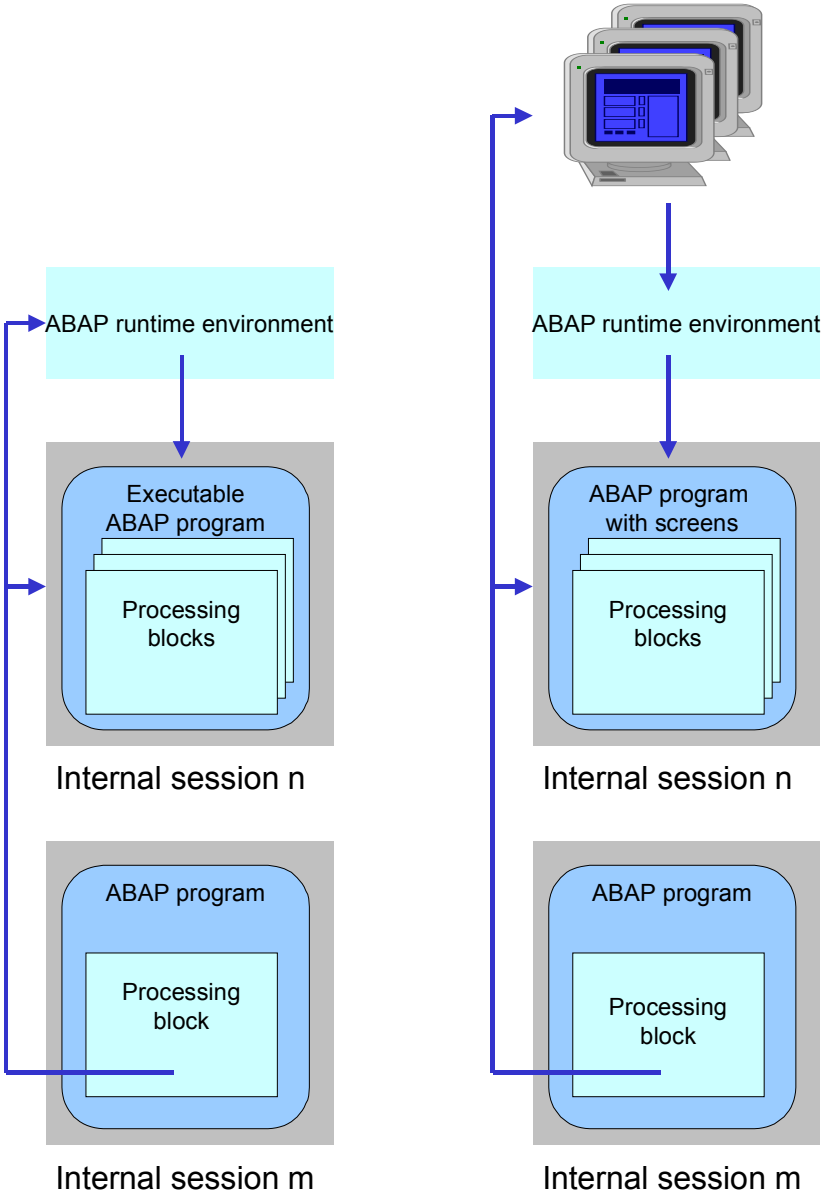
Subscreens

You can call a subscreen screen [Page 1380] externally by specifying an ABAP program other than the current program in the CALL SUBSCREEN statement. This program is treated in the same way as in an external subroutine call. In other words, it is loaded into the program group of the calling program, or, if it is a function group, as its own program group in the same internal session as the calling ABAP program.

For this reason, you should not use external subscreens that do not belong to a function group, since the same problems can occur with interface work areas [Page 130] and screen calls as can occur with external subroutines.

External Program Calls

In an external program call, the unit that you call is an independent ABAP program. At runtime, the called program is loaded into **its own internal session** in the current external session. Any program that can have its own screens can be called in an external program call. The most usual external program calls are for executable programs and transactions assigned to a module pool.



Executable Programs

When you call an executable program, the program is loaded, and the ABAP runtime environment calls the processors that control its flow.

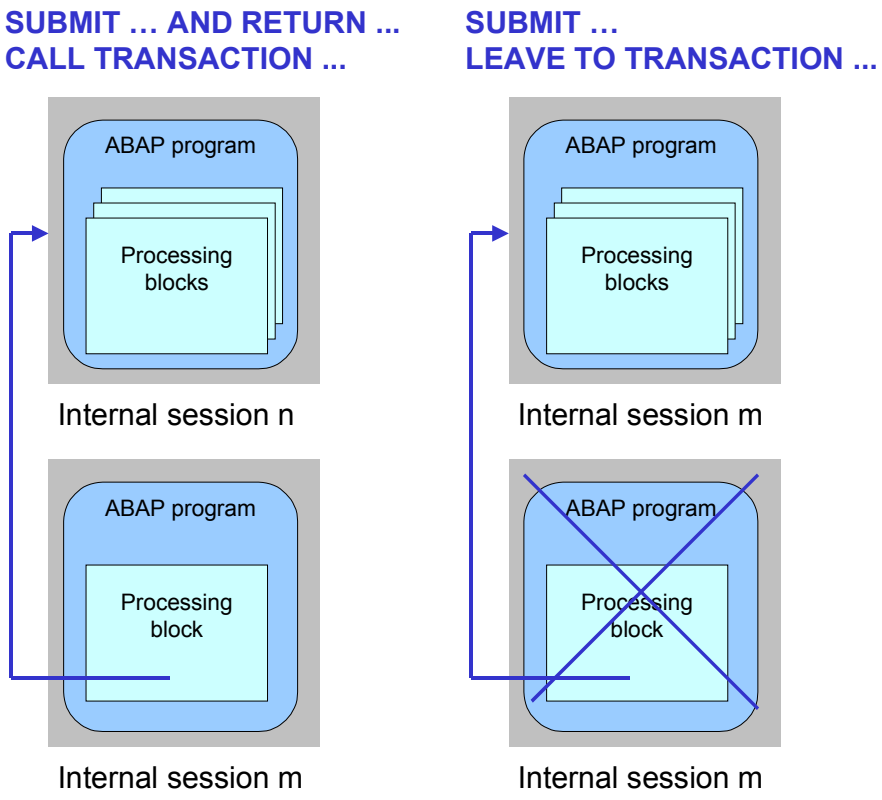
Transactions

When you call a transaction, the program linked to the transaction code is loaded and its initial screen is processed. The initial screen calls dialog modules in the called program, and then branches to its next screen.

Callable Units

ABAP Programs

Any ABAP programs with type 1, M, or F that have their own screens can be called externally in their **own internal session** and with their **own SAP LUW**.



Executable Programs

You can call executable programs directly using the statement

SUBMIT <prog> ... [AND RETURN].

If you omit the AND RETURN addition, the system terminates the calling program and calls the new executable program <prog>. The internal session of the calling program is replaced by the internal session of the new program. When the new program has finished, control returns to the point from which the calling program was started.

If you use the AND RETURN addition, the executable program <prog> is started in a new session. The internal session of the calling program is returned, and control returns to the calling program after the new program has finished.

All Programs with Screens

You can assign a transaction code to any screen of a program. Normally, you assign a transaction code to a single screen in a module pool (type M program). This then allows you to start the program using one of the statements

LEAVE TO TRANSACTION <tcode> ...

or

CALL TRANSACTION <tcode> ...

The program starts by processing the screen that you specified when you defined the transaction code. This is called the initial screen.

If you use LEAVE TO TRANSACTION, the calling program terminates, and its internal session is replaced by the internal session of the new program. When the new program has finished, control returns to the point from which the calling program was started.

If you use CALL TRANSACTION, the calling program is started in a new internal session, and the internal session of the calling program is retained. When the new program has finished, control returns to the point from which the calling program was started.

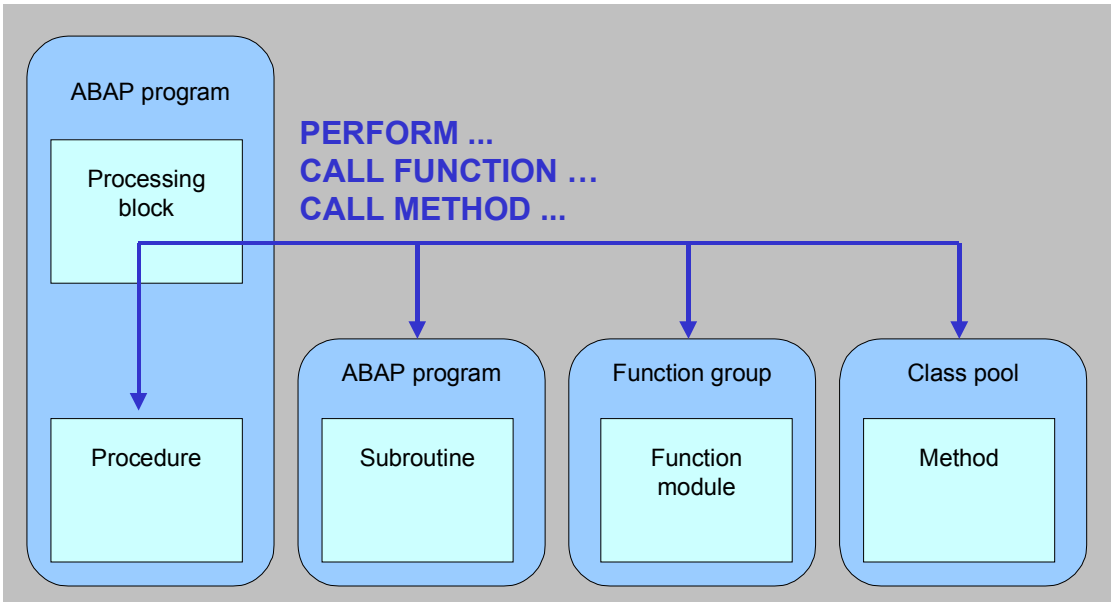
Leaving a Called Program

To leave a program that you have called, use the ABAP statement

LEAVE PROGRAM.

Procedures

Called procedures (subroutines, function modules, and methods) always run in the **same internal session** as the calling program.



Internal session

Subroutines

You call a subroutine using the

PERFORM <subr>[(<prog>)] ...

statement. If you omit the (<prog>) addition, the subroutine is called internally, that is, it belongs to the calling program and must not be loaded.

If you use the (<prog>) addition, the call is external, that is, the subroutine belongs to the program <prog>. When you call the subroutine, the entire program <prog> is loaded into the internal session of the calling program (if it has not been already). The loaded program belongs to the program group of the calling program. However, if the subroutine belongs to a function group, a new additional program group is created. The program group to which the external subroutine belongs determines the interface work areas and screens that it will use. However, this assignment can vary dynamically, so it is best to avoid using external subroutines.

Function Modules

You call function modules using the

CALL FUNCTION <func> ...

statement. Function module calls are external unless a function module is called by another procedure within the same function group. When you call a function module, its entire function group is loaded into the internal session of the calling program (unless it has already been

loaded). Within the internal session, the function group forms an additional program group with its own interface work areas and screens. This means that function modules provide better encapsulation of data and screens than external subroutines.

Methods

You call a method using the

CALL METHOD [<ref>->|<class>=>]<meth> ...

statement. If you omit the <ref> or <class> part of the statement, the call is local within the same class. The class is not reloaded.

If you use <ref> or <class>, the method call applies to the method of the class specified through the object reference <ref> or the class name <class>. When you call the method, the entire class pool is loaded into the internal session of the calling program (unless it has already been loaded). The class pool forms an additional program group in the internal session, which does not share data and screens with the caller, and from which you cannot call external subroutines.

Leaving a Called Procedure

You can leave a procedure using the

EXIT.

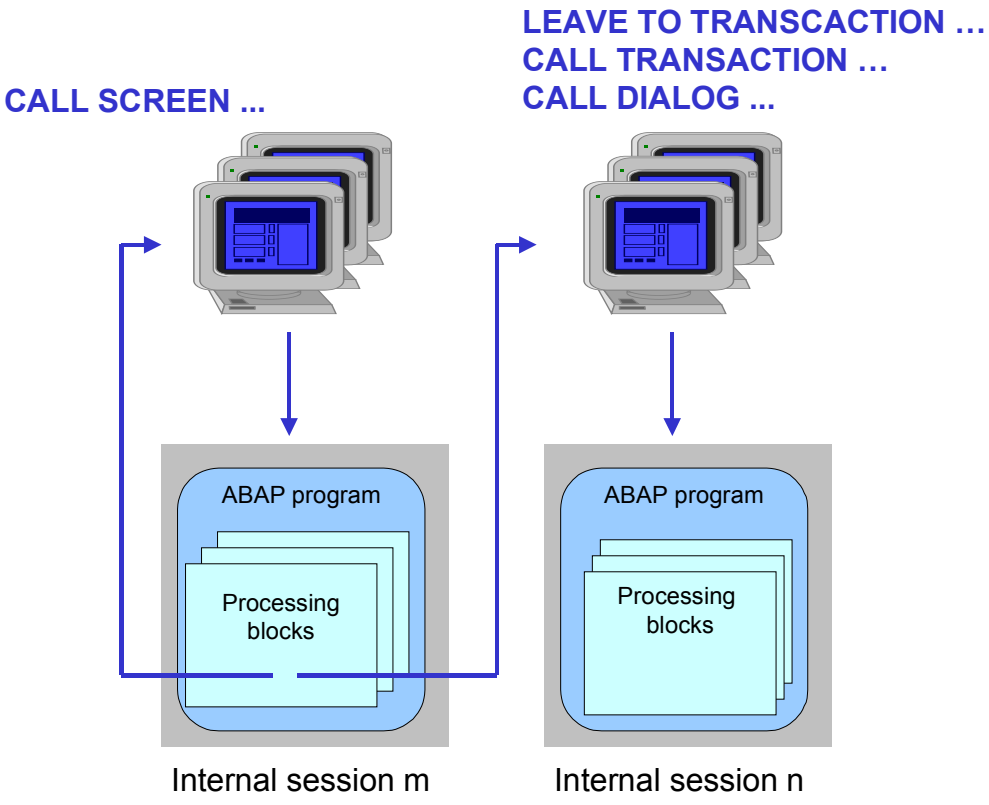
or

CHECK <logexp>.

statement.

Screens and Screen Sequences

Screens and their flow logic, which together form dynamic programs, are instances that control the flow of an ABAP program by calling a series of dialog modules. Screens can be combined into sequences, where the next screen in the sequence can be defined either statically or dynamically. The simplest screen sequence is a single screen. The sequence starts when you call its first screen. You can do this in a variety of ways.



Calling Screens Internally from the Same ABAP Program

In any ABAP program that can have its own screens (type 1, M, or F), you can use the

CALL SCREEN <dynnr>.

statement to call a screen and its subsequent sequence within that program. The flow logic of each screen calls dialog modules in the program that called the screen.

When the screen sequence ends, control returns to the statement after the original CALL SCREEN statement.

Calling Screens as a Transaction

A transaction (or transaction code) links a screen to a main program (usually a module pool).

You can call a transaction from any ABAP program using the

CALL TRANSACTION <tcod> ...

or

LEAVE TO TRANSACTION <tcod> ...

statement. The transaction starts with the initial screen that you specified when you defined the transaction code. The program of the transaction is started in a new internal session, and **has its own SAP LUW**. The screens in the screen sequence call the various dialog modules of the main program.

When the screen sequence is finished, control returns to the program that contained the CALL TRANSACTION statement. If you start a transaction using LEAVE TO TRANSACTION, control returns to the point from which the calling program was started.

Calling Screens as a Dialog Module

A dialog module can be linked to a screen of any main program, usually a module pool.

You can call a dialog module from any ABAP program using the

CALL DIALOG <diag> ...

statement. The dialog module starts with the initial screen that you specified when you defined it. The program of the dialog module is started in a new internal session, and **has its own SAP LUW**. The screens in the screen sequence call the various dialog modules of the main program.

When the screen sequence ends, control returns to the statement after the dialog module call.

Dialog modules are obsolete, and should no longer be used. Instead, you can encapsulate screen sequences in function groups and call them from an appropriately-programmed function module.

Leaving a Screen Sequence

A screen sequence terminates when a screen ends and the defined next screen has the number 0.

You can leave a **single** screen within a sequence using the

LEAVE SCREEN.

or

LEAVE TO SCREEN <dynnr>.

statement. These statements exit the current screen and call the defined next screen. If the next screen is screen 0, the entire screen sequence concludes.

Special Single Screens

There are three special types of screen:

Selection Screen

A selection screen is a special screen, created using ABAP statements. You can only call them using the

CALL SELECTION-SCREEN <dynnr> ...

statement. The selection screen is processed (reaction to user input in the selection screen events) in the calling program.

List

Each screen in a screen sequence has a corresponding list system of twenty levels. You can start this list system using the

LEAVE TO LIST-PROCESSING [AND RETURN TO SCREEN <dynnr>].

statement. This statement calls a system program that contains the standard container screen used for lists. This replaces the current screen. On this screen, you can display a basic list and up to 19 detail lists. List processing (reacting to user actions in list events) takes place in the calling program.

You can leave the list system using the

LEAVE LIST-PROCESSING.

statement.

In an executable program, the list system is automatically called after the last reporting event.

Subscreens

In the PBO event of the **flow logic** of a screen, you can call a subscreen using the following statement:

CALL SUBSCREEN <area> INCLUDING [<prog>] <dynnr>.

The screen of a subscreen that you call is placed in the subscreen area <area> on the main screen.

If you do not specify a program <prog>, the system uses a screen from the current ABAP program. If you do specify a program <prog>, the system uses a screen from the program <prog> for the subscreen. This program is treated in the same way as an external subroutine call. In other words, it is loaded into the program group of the calling program, or, if it is a function group, as its own program group in the same internal session as the calling ABAP program.

ABAP Statement Overview

The following is an alphabetical classification of the most important generally-released ABAP statements.

A B C D E F G H I L M N O P R S T U W

A

ADD for Single Fields

Adds two single fields.

Syntax

```
ADD <n> TO <m>.
```

The contents of <n> are added to the contents of <m>. The result is placed in <m>. Equivalent of <m> = <m> + <n>.

ADD for Sequences of Fields

Adds sequences of fields.

Syntax

```
ADD <n1> THEN <n2> UNTIL <nz> GIVING <m>.
ADD <n1> THEN <n2> UNTIL <nz> ACCORDING TO <sel> GIVING <m>.
ADD <n1> THEN <n2> UNTIL <nz> TO <m>.
ADD <n1> FROM <m1> TO <mz> GIVING <m>.
```

If <n1>, <n2>,..., <nz> is a sequence of fields with a uniform gap between each, the same type, and the same length, the fields are added together and the result placed in <m>. The variants allow you to restrict the fields to a partial sequence, to include <m> in the sum, or to perform the operation for a sequence of consecutive fields.

ADD-CORRESPONDING

Adds components of structures.

Syntax

```
ADD-CORRESPONDING <struc1> TO <struc2>.
```

Adds together all of the components of structures <struc1> and <struc2> that have identical names, and places the results in the corresponding components of <struc2>.

ALIASES

Defines class-specific alias names for an interface component in ABAP Objects.

Syntax

```
ALIASES <alias> FOR <intf~comp>.
```

Defines <alias> within a class as a synonym for the interface component <intf~comp>.

APPEND

Appends one or more lines to the end of an index table.

Syntax

```
APPEND <line>|LINES OF <jtab> TO <itab>.
```

Appends one line <line> or several lines of an internal table <jtab> to the index table <itab>.

ASSIGN

Assigns a field to a field symbol.

Syntax

```
ASSIGN <f> TO <FS>.
```

Assigns the data object <f> to the field symbol <FS>, after which, <FS> points to the data object. The pointed brackets are part of the syntax of the field symbol.

AT for Event Blocks

Event keywords for defining event blocks for screen events.

Syntax

```
AT SELECTION-SCREEN...

AT LINE-SELECTION.

AT USER-COMMAND.

AT PFn.
```

User actions on selection screens or lists trigger events in the ABAP runtime environment. The event keywords define event blocks, which are called when the corresponding event occurs.

AT for Control Levels

Control level change when you process extracts and internal tables in a loop.

Syntax

```
AT NEW <f>.
```

```
AT END OF <f>.

AT FIRST.

AT LAST.

AT <fg>.
```

These statements are used in control level processing with extract datasets or internal tables. Each introduces a statement block that you must conclude with the ENDAT statement. The statements between AT and ENDAT are executed whenever the corresponding control level change occurs.

AUTHORITY-CHECK

Checks the authorization of a user.

Syntax

```
AUTHORITY-CHECK OBJECT <object> ID <name1> FIELD <f1>
                                ID <name2> FIELD <f2>
                               ...
                                ID <name10> FIELD <f10>.
```

The statement checks whether the user has all of the authorizations defined in the authorization object <object>. <name1>,..., <name10> are the authorization fields in the object, and <f1>,... ,<f10> are data objects in the program. The value of each data object is checked against the corresponding authorization field.

B

BACK

Relative positioning for output in a list.

Syntax

```
BACK.
```

Positions the list output either in the first column of the first line after the page header on the current page, or in the first column of the first line of a line block if you have previously used the RESERVE statement.

BREAK-POINT

Starts the ABAP Debugger.

Syntax

```
BREAK-POINT.
```

Interrupts program execution and starts the Debugger. This allows you to test your programs by halting them at any point.

C

CALL CUSTOMER-FUNCTION

Calls a customer function module.

Syntax

```
CALL CUSTOMER-FUNCTION <func>...
```

Similar to CALL FUNCTION. The function module that it calls must be programmed and activated by a customer using the modification concept.

CALL FUNCTION

Calls a function module.

Syntax

CALL FUNCTION <func> [EXPORTING ... f_i = a_i...]
[IMPORTING ... f_i = a_i...]
[CHANGING ... f_i = a_i...]
[TABLES ... f_i = a_i...]
[EXCEPTIONS... e_i = r_i...]
[DESTINATION <dest>]
[IN UPDATE TASK]
[STARTING NEW TASK]
[IN BACKGOUND TASK].

Starts a function module, either in the same system or in an external system, depending on the type of call that you use. You can also use update function modules in transactions, and call function modules asynchronously. The remaining additions are used for the parameter interface of the function module, in which you can specify actual parameters and determine how to handle exceptions.

CALL DIALOG

Calls a dialog module.

Syntax

CALL DIALOG <dialog> [AND SKIP FIRST SCREEN]
[EXPORTING... f_i = a_i...]
[IMPORTING... f_i = a_i...]
[USING itab].

Calls the dialog module <dial>. A dialog module is an ABAP program containing a chain of screens. It does not have to be called using a transaction code, and runs in the same SAP LUW as the program that called it. The additions allow you to skip the first screen in the chain and pass actual parameters to and from the parameter interface of the dialog module.

CALL METHOD

Calls a method in ABAP Objects.

Syntax

CALL METHOD <meth> [EXPORTING ... <i_i> =.<f_i>...]
[IMPORTING ... <e_i> =.<g_i>...]
[CHANGING ... <c_i> =.<f_i>...]
[RECEIVING r = h]
[EXCEPTIONS... <e_i> = r_i...]

Calls a static method or instance method <meth>. The additions allow you to pass parameters to and from the method and handle its exceptions.

CALL METHOD OF

Calls a method in OLE2 Automation.

Syntax

```
CALL METHOD OF <obj> <m>.
```

Calls the method <m> of the OLE2 Automation object <obj>.

CALL SCREEN

Calls a sequence of screens.

Syntax

```
CALL SCREEN <scr>
            [STARTING AT <X1> <Y1>]
            [ENDING AT <X2> <Y2>].
```

Calls the sequence of screens beginning with screen number <scr>. All of the screens in the chain belong to the same ABAP program. The chain ends when a screen has the next screen number 0. The additions allow you to call a single screen as a modal dialog box.

CALL SELECTION-SCREEN

Calls a selection screen.

Syntax

```
CALL SELECTION-SCREEN <scr>
                      [STARTING AT <x1> <y1>]
                      [ENDING AT <x2> <y2>].
```

Calls a user-defined selection screen in a program. Selection screens are processed in the AT SELECTION-SCREEN events. The additions allow you to call a selection screen as a modal dialog box.

CALL TRANSACTION

Calls a transaction.

Syntax

```
CALL TRANSACTION <tcod>
                 [AND SKIP FIRST SCREEN]
                 [USING <itab>].
```

Calls the transaction <tcod> while retaining the data in the calling program. At the end of the transaction, control returns to the point from which the transaction was called. The additions allow you to skip the first screen of the transaction or pass an internal table for batch input to it.

CASE

Conditional branching.

Syntax

```
CASE <f>.
```

Opens a CASE control structure that must conclude with the ENDCASE statement. This allows you to branch to various statement blocks (introduced with the WHEN statement), depending on the contents of the data object <f>.

CATCH

Catches runtime errors.

Syntax

```
CATCH SYSTEM-EXCEPTIONS <except1> = <rc1>... <exceptn> = <rcn>.
```

Introduces a CATCH area, which concludes with an ENDCATCH statement. If a catchable runtime error <$except_i$> occurs within this block, the current block terminates immediately, and the program jumps directly to the corresponding ENDCATCH statement, filling SY-SUBRC with <rc_i>.

CHECK

Conditional termination of a loop pass or a processing block.

Syntax

```
CHECK <logexp>.
```

If the logical expression <logexp> is true, the program continues at the next statement. If, however, <logexp> is false, the current loop pass terminates and the next begins. If the program is not currently processing a loop, the current processing block terminates. There are special forms of the CHECK statement for use with selection tables and in GET event blocks.

CLASS - Declaration

Declares a class in ABAP Objects.

Syntax

```
CLASS <class> DEFINITION [PUBLIC]
              [INHERITING FROM <superclass>]
              [DEFERRED]
              [LOAD].
```

This statement introduces the declaration part of a class <class>. The declaration part concludes with ENDCLASS, and contains the declarations of all components in the class. The PUBLIC addition is generated by the Class Builder, and defines a global class in the class library. The INHERITING FROM addition allows you to derive the class <class> from a superclass <superclass>. The DEFERRED addition declares the class before it is actually defined. The LOAD addition loads a class explicitly from the class library.

CLASS – Implementation

Implementation of a class in ABAP Objects.

Syntax

```
CLASS <class> IMPLEMENTATION.
```

Introduces the implementation part of a class <class>. This concludes with the ENDCLASS statement, and contains the implementations of all of the methods in the class.

CLASS-DATA

Declares static attributes of a class or interface.

Syntax

```
CLASS-DATA <a>...
```

Like DATA. However, the attribute <a> is declared as a static attribute. Static attributes are independent of instances of the class. Only one copy of the attribute exists in the class, and this is shared by all instances.

CLASS-METHODS

Declares static methods of a class or interface.

Syntax

```
CLASS-METHODS <meth>...
```

Like METHODS. However, the method <meth> is declared as a static method. A static method can access static attributes, and may only trigger static events.

CLASS-EVENTS

Declares static events of a class or interface.

Syntax

```
CLASS-EVENTS <evt>...
```

Like EVENTS. However, the event <evt> is declared as a static event. Static events are the only events that may be triggered in a static method.

CLEAR

Resets a variable to its initial value.

Syntax

```
CLEAR <f>.
```

Resets the variable <f>, which may be of any data type, to the initial value defined for that type.

CLOSE DATASET

Closes a file.

Syntax

```
CLOSE DATASET <dsn>.
```

Closes a file <dsn> on the application server previously opened with the OPEN DATASET statement.

CLOSE CURSOR

Closes a database cursor.

Syntax

```
CLOSE CURSOR <c>.
```

Closes a cursor opened using the OPEN CURSOR statement.

COLLECT

Inserts lines into an internal table in summarized form.

Syntax

```
COLLECT <line> INTO <itab>.
```

The statement first checks whether the internal table contains an entry with the same key. If not, it acts like INSERT. If there is already a table entry with the same key, COLLECT does not insert a new line. Instead, it adds the values from the numeric fields of the work area <line> to the values in the corresponding fields of the existing table entry.

COMMIT

Concludes an SAP LUW.

Syntax

```
COMMIT WORK [AND WAIT].
```

All database updates are written firmly to the database, and all locks are released. Triggers the database update. The AND WAIT addition allows you to pause the program until the update is complete. If you omit it, the database is updated asynchronously.

COMMUNICATION

Allows communication between programs.

Syntax

```
COMMUNICATION INIT DESTINATION <dest> ID <id> [Additions].
COMMUNICATION ALLOCATE ID <id> [Additions].
COMMUNICATION ACCEPT ID <id> [Additions].
COMMUNICATION SEND ID <id> BUFFER <f> [Additions].
COMMUNICATION RECEIVE ID <id> [Additions].
COMMUNICATION DEALLOCATE ID <id> [Additions].
```

These statements allow you to initialize, start, and accept program-to-program communication, send and receive data between partner programs, and then terminate the connection.

COMPUTE

Performs numeric operations.

Syntax

```
COMPUTE <n> = <expression>.
```

The result of the mathematical expression in <expression> is assigned to the result field <n>. The COMPUTE keyword is optional.

CONCATENATE

Combines a series of strings into a single string.

Syntax

```
CONCATENATE <c1>... <cn> INTO <c> [ SEPARATED BY <s> ].
```

The strings <c1> to <cn> are concatenated, and the result placed in <c>. The SEPARATED BY addition allows you to specify a string <s> to be placed between the strings.

CONDENSE

Removes spaces from a string.

Syntax

```
CONDENSE <c> [NO-GAPS].
```

Removes all leading spaces, and replaces other series of blanks with a single space in the character field <c>. If you use the NO-GAPS addition, all of the spaces are removed.

CONSTANTS

Declares constant data objects.

Syntax

```
CONSTANTS <c>... VALUE [<val> | IS INITIAL]...
```

The syntax is similar to DATA, except that the VALUE addition is required, and that internal tables and deep structures cannot be declared as constants. The starting value that you assign in the VALUE addition cannot be changed during the program.

CONTINUE

Ends a loop pass.

Syntax

```
CONTINUE.
```

Only possible within loops. This statement terminates the current loop pass and starts the next.

CONTEXTS

Declares a context.

Syntax

CONTEXTS <c>.

Generates an implicit data type CONTEXT_<c>, which you can use to create context instances.

CONTROLS

Defines a control.

Syntax

```
CONTROLS <ctrl> TYPE <ctrl_type>.
```

Defines an ABAP runtime object <ctrl>. This displays data in a particular format on a screen, depending on the type <ctrl_type>. Currently, <ctrl_type> may be a table control or tabstrip control.

CONVERT for Dates

Converts a data into an inverted date form.

Syntax

```
CONVERT DATE <d1> INTO INVERTED-DATE <d2>.
```

```
CONVERT INVERTED-DATE <d1> INTO DATE <d2>.
```

If <d1> and <d2> are date fields in the internal form YYYYMMDD, the nines complement of <d1> is placed in field <d2> and vice versa. In inverted date format, the most recent date has the smaller numerical value.

CONVERT for Timestamps

Converts a timestamp into the correct date and time for the current time zone.

Syntax

```
CONVERT TIME STAMP <tst> TIME ZONE <tz> INTO DATE <d> TIME <t>.
```

```
CONVERT DATE <d> TIME <t> INTO TIME STAMP <tst> TIME ZONE <tz>.
```

As long as <tst> has type P(8) or P(11) with 7 decimal placed, and <tz> has type C(6), the time stamp <tst> will be converted to the correct date <d> and time <t> for the time zone <tz>.

CONVERT for Text

Converts a text into a format that can be sorted alphabetically.

Syntax

```
CONVERT TEXT <text> INTO SORTABLE CODE <x>.
```

<text> must have type C and <x> must have type X. The string is then converted so that the relative order of the characters allows them to be sorted alphabetically in the current text environment.

CREATE OBJECT in ABAP Objects

Creates an object in ABAP Objects.

Syntax

```
CREATE OBJECT <cref>.
```

<cref> must be a reference variable, defined with reference to a class. CREATE OBJECT then creates an object of that class, to which the reference in <cref> then points.

CREATE OBJECT in OLE2 Automation

Creates an external object in OLE2 Automation.

Syntax

```
CREATE OBJECT <obj> <class>.
```

If <class> is a class assigned to an automation server, an initial object <obj> of this class is created.

D

DATA with Reference to Declared Data Types

Declares variables with a previously-declared data type.

Syntax

DATA <f>... [TYPE <type>|LIKE <obj>]... [VALUE <val>].

Declares a variable <f> with the fully-defined data type <type> or the same data type as another data object <obj>. The data type <type> can be D, F, I, T, a type defined locally in the program using the TYPES statement, or a type from the ABAP Dictionary. The data object <obj> is a data object or line of an internal table that has already been defined. The VALUE addition allows you to specify a starting value.

DATA with Reference to Generic Data Types

Declares variables by completing the description of a generic type.

Syntax

DATA <f>[(<length>)] TYPE <type> [DECIMALS <d>]... [VALUE <val>].

DATA <f> TYPE <itab>.

The data type <type> can be C, N, P, or X. In the <length> option, you specify the length of the field. If you do not specify the length, the default value for the data type is used. If <type> is P, you can use the DECIMALS addition to specify a number of decimal places <d>. If you do not specify a number of decimal places, it is set to none. If you do not specify a type, the system uses the default type C.

Syntax

DATA <f> TYPE <itab>.

The data type <itab> is a standard internal table with generic key. The default key is automatically used in the DATA statement.

DATA, Creating an Associated Data Type

Declares variables with data types that only exist as an attribute of the variable.

Syntax

DATA <f> TYPE REF TO <class>|<interface>.

Declares the variable <f> as a reference variable for the class <class> or the interface <interface>.

Syntax

DATA: BEGIN OF <structure>,
...
<f_i>...,
...
END OF <structure>.

Combines the variables <f_i> to form the structure <structure>. You can address the individual components of a structure by placing a hyphen between the structure name and the component name: <structure>-<f_i>.

Syntax

DATA <f> TYPE|LIKE <tabkind> OF <linetype> WITH <key>.

Declares the variable <f> as an internal table with the table type <tabkind>, line type <linekind>, and key <key>.

DATA for Shared Data Areas

Declares shared data areas in a program.

Syntax

```
DATA: BEGIN OF COMMON PART <c>,
        <fi>...
      END OF COMMON PART.
```

The variables <f_i> are assigned to a data area <c>, which can be defined in more than one program. These data areas use the same memory addresses for all programs that are loaded into the same internal session.

DEFINE

Defines a macro.

Syntax

```
DEFINE <macro>.
```

Introduces the definition of the macro <macro>. Each macro must consist of complete ABAP statement and be concluded with the END-OF-DEFINITION statement.

DELETE for Files

Deletes files on the application server.

Syntax

```
DELETE DATASET <dsn>.
```

Deletes the file <dsn> from the file system of the application server.

DELETE for Database Table Entries

Deletes entries from database tables.

Syntax

```
DELETE FROM <dbtab> WHERE <cond>.
```

Deletes all of the lines from the database table <dbtab> that satisfy the WHERE condition.

Syntax

DELETE <dbtab> FROM <wa>.

DELETE <dbtab> FROM TABLE <itab>.

Deletes the lines with the same primary key as the work area <wa>, or all of the lines from the database table with the same primary key as one of the lines in the internal table <itab>. The work area <wa> or the lines of the internal table <itab> must be at least as long as the primary key of the database table and have the same alignment.

DELETE for Cluster Database Tables

Deletes data clusters from cluster database tables.

Syntax

DELETE FROM DATABASE <dbtab>(<ar>) ID <key>.

Deletes the entire data cluster from the area <ar> with the name <key> from the cluster database table <dbtab>.

DELETE for the Cross-Transaction Application Buffer

Deletes data clusters from the cross-transaction application buffer.

Syntax

DELETE FROM SHARED BUFFER <dbtab>(<ar>) ID <key>.

Deletes the data cluster for the area <ar> with the name <key> stored in the cross-transaction application buffer for the table <dbtab>.

DELETE for Lines from an Internal Table

Deletes lines from internal tables of any type.

Syntax

DELETE TABLE <itab> FROM <wa>.

DELETE TABLE <itab> WITH TABLE KEY $<k_1>$ = $<f_1>$... $<k_n>$ = $<f_n>$.

Deletion using the table key: All lines with the same key are deleted. The key values are taken either from a compatible work area <wa> or specified explicitly.

Syntax

DELETE <itab> WHERE <cond>.

Deletion using a condition: Deletes all table entries that satisfy the logical expression <cond>. The logical condition may consist of more than one expression. However, the first operand in each expression must be a component of the line structure.

Syntax

DELETE ADJACENT DUPLICATE ENTRIES FROM <itab> [COMPARING...].

Deletes adjacent duplicate entries, either by comparing the key fields or the comparison fields specified explicitly in the COMPARING addition.

DELETE for Lines from Index Tables

Deletes lines from index tables.

Syntax

DELETE <itab> [INDEX <idx>].

If you use the INDEX option, deletes the line with the index <idx> from the table <itab>. If you do not use the INDEX option, the statement can only be used within a LOOP … ENDLOOP construction. In this case, it deletes the current line.

Syntax

DELETE <itab> [FROM <n_1>] [TO <n_2>] [WHERE <cond>].

Deletes all rows from <itab> with index between <n_1 > and <n $_2$ > an which satisfy the WHERE condition. If you do not use the FROM addition, the system deletes lines starting at the beginning of the table. If you do not use the TO addition, the system deletes lines to the end of the table. The logical expression <cond> can consist of more than one expression. However, the first operand in each expression must be a component of the line structure of the internal table.

DEMAND

Retrieves values from a context instance.

Syntax

DEMAND <val_1> = <f $_1$>... <val $_n$> = <f $_n$> FROM CONTEXT <inst>
[MESSAGES INTO <itab>].

Fills the fields <f_n> with the values <val_n> of the context instance <inst>. The MESSAGES addition allows you to control how messages from the context are handled in the program.

DESCRIBE DISTANCE

Determines the distance between two fields.

Syntax

```
DESCRIBE DISTANCE BETWEEN <f1> AND <f2> INTO <f3>.
```

Writes the distance in bytes between fields <f1> and <f2> to <f3>, always including the length of the field that occurs first in memory.

DESCRIBE FIELD

Describes the attributes of a field.

Syntax

```
DESCRIBE FIELD <f> [LENGTH <l>] [TYPE <t> [COMPONENTS <n>]]
                   [OUTPUT-LENGTH <o>] [DECIMALS <d>]
                   [EDIT MASK <m>] [HELP-ID <h>].
```

The attributes of the data object <f> named in the additions to the statement are placed in the corresponding variables. You can use any number of additions in a single statement.

DESCRIBE LIST

Describes the attributes of a list.

Syntax

```
DESCRIBE LIST NUMBER OF LINES <lin> [INDEX <idx>].

DESCRIBE LIST NUMBER OF PAGES <n> [INDEX <idx>].

DESCRIBE LIST LINE <lin> PAGE <pag> [INDEX <idx>].

DESCRIBE LIST PAGE <pag> [INDEX <idx>]...
```

Depending on the variant of the statement that you use, writes the number of lines, number of pages, a line of a list on a given page, or various attributes of a page to variables.

DESCRIBE TABLE

Describes the attributes of an internal table.

Syntax

```
DESCRIBE TABLE [LINES <l>] [OCCURS<n>] [KIND <k>].
```

Depending on the additions you use, writes the number of lines occupied, the value specified for the INITIAL SIZE of the table, or the table type into a corresponding variable.

DIVIDE

Divides one field by another.

Syntax

```
DIVIDE <n> BY <m>.
```

Divides the content of <n> by <m>, and places the result in <n>. The equivalent of n = n / m.

DIVIDE-CORRESPONDING

Divides matching components of structures.

Syntax

```
DIVIDE-CORRESPONDING <struc1> BY <struc2>.
```

Divides all matching components of the structures <struc1> and <struc2> and places the results into the corresponding components of <struc1>.

DO

Introduces a loop.

Syntax

DO [<n> TIMES] [VARYING <f> FROM <f1> NEXT <f2>].

Introduces a statement block that must conclude with ENDDO. If you omit the TIMES addition, the statement block is repeated until a termination statement such as CHECK or EXIT occurs. The TIMES addition restricts the number of loop passes to <n>. The VARYING addition allows you to process a sequence of fields the same distance apart in memory.

E

EDITOR-CALL

Loads an ABAP program or internal table into a text editor.

Syntax

```
EDITOR-CALL FOR <itab>...
```

```
EDITOR-CALL FOR REPORT <prog>...
```

Loads the internal table <itab> or the program <prog> into a text editor, where you can edit it using standard editor functions.

ELSE

Introduces a statement block in an IF control structure.

Syntax

```
ELSE.
```

If the logical expression in an IF statement is false, ELSE introduces the statement block to be executed instead.

ELSEIF

Introduces a statement block in an IF control structure.

Syntax

```
ELSEIF <logexp>.
```

If the logical expression in the preceding IF is false and <logexp> is true, ELSEIF introduces the statement block that will be executed.

END-OF-DEFINITION

Concludes a macro definition.

Syntax

```
END-OF-DEFINITION.
```

This statement concludes a macro definition introduced with DEFINITION.

END-OF-PAGE

Event keyword for defining an event block for a list event.

Syntax

```
END-OF-PAGE.
```

Whenever the page footer is reached while a list is being created, the runtime environment triggers the END-OF-PAGE event, and the corresponding event block is executed.

END-OF-SELECTION

Event keyword for defining an event block for a reporting event.

Syntax

```
END-OF-SELECTION.
```

Once a logical database has read all of the required lines and passed them to the executable program, the runtime environment triggers the END-OF-SELECTION event, and the corresponding event block is executed.

ENDAT

Concludes a statement block in control level processing.

Syntax

```
ENDAT.
```

The statement concludes a control level processing block introduced with the AT statement.

ENDCASE

Concludes a CASE control structure.

Syntax

```
ENDCASE.
```

This statement concludes a control structure introduced with the CASE statement.

ENDCATCH

Concludes a CATCH area.

Syntax

ENDCATCH.

The statement concludes an exception handling area introduced with CATCH.

ENDCLASS

Concludes a class definition.

Syntax

ENDCLASS.

This statement concludes a class declaration or implementation introduced with CLASS.

ENDDO

Concludes a DO loop.

Syntax

```
ENDDO.
```

This statement concludes a loop introduced with DO.

ENDEXEC

Concludes a Native SQL statement.

Syntax

```
ENDEXEC.
```

This statement concludes a Native SQL statement introduced with EXEC SQL.

ENDFORM

Concludes a subroutine.

Syntax

```
ENDFORM.
```

This statement concludes a subroutine definition introduced with FORM.

ENDFUNCTION

Concludes a function module.

Syntax

```
ENDFUNCTION.
```

This statement concludes a function module introduced with FUNCTION.

ENDIF

Concludes an IF control structure.

Syntax

```
ENDIF.
```

This statement concludes a control structure introduced using IF.

ENDINTERFACE

Concludes an interface definition.

Syntax

```
ENDINTERFACE.
```

This statement concludes an interface definition introduced with INTERFACE.

ENDLOOP

Concludes a loop.

Syntax

```
ENDLOOP.
```

This statement concludes a loop introduced with LOOP.

ENDMETHOD

Concludes a method.

Syntax

```
ENDMETHOD.
```

This statement concludes a method implementation introduced with METHOD.

ENDMODULE

Concludes a dialog module.

Syntax

```
ENDMODULE.
```

This statement concludes a dialog module introduced with MODULE.

ENDON

Concludes a conditional statement block.

Syntax

```
ENDON.
```

This statement concludes a conditional statement block introduced with ON CHANGE.

ENDPROVIDE

Concludes a PROVIDE loop.

Syntax

```
ENDPROVIDE.
```

This statement concludes a loop introduced with PROVIDE.

ENDSELECT

Concludes a SELECT loop.

Syntax

```
ENDSELECT.
```

This statement concludes a loop introduced with SELECT.

ENDWHILE

Concludes a WHILE loop.

Syntax

```
ENDWHILE.
```

This statement concludes a loop introduced with WHILE.

EVENTS

Defines events in classes or interfaces.

Syntax

EVENTS <evt> EXPORTING.. VALUE(<e_i>) TYPE type [OPTIONAL]...

The event <evt> can be declared in the declaration part of a class or within an interface definition, and may have EXPORTING parameters that are passed to the event handler. The parameters are always passed by value.

EXEC SQL

Introduces a Native SQL statement.

Syntax

```
EXEC SQL [PERFORMING <form>].
```

Between EXEC SQL and the ENDEXEC statement, you can include a database-specific Native SQL statement. The PERFORMING addition allows you to pass a multiple-line selection line by line to a subroutine.

EXIT

Terminates a loop or processing block.

Syntax

```
EXIT.
```

Within a loop: The entire loop is terminated, and processing continues with the first statement following the loop.

Outside a loop: Terminates the current processing block.

In a reporting event: Jumps directly to the output list.

EXIT FROM STEP-LOOP

Ends a step loop.

Syntax

```
EXIT FROM STEP-LOOP.
```

Terminates step loop processing. A step loop is a way of displaying a table on a screen.

EXIT FROM SQL

Terminates Native SQL processing.

Syntax

```
EXIT FROM SQL.
```

This statement may occur within a subroutine called using the PERFORMING addition in the EXEC SQL statement. The entire subroutine is processed, but no more subsequent lines of the selection are processed.

EXPORT

Exports a data cluster.

Syntax

EXPORT... <f_i> [FROM <g_i>]... | (<itab>)
TO MEMORY
| DATABASE <dbtab>(<ar>) ID(<key>)
| SHARED BUFFER <dbtab>(<ar>) ID(<key>).

The data objects <f_i> or <g_i>, or the data objects in the internal table <itab> are stored as a data cluster in the cross-program ABAP memory of the current internal session, in a cluster database table <dbtab>, or in the cross-transaction application buffer of the table <dbtab>.

EXTRACT

Creates an extract dataset and adds lines to it.

Syntax

```
EXTRACT <fg>.
```

The first EXTRACT statement in a program creates an extract dataset and adds the first entry to it. Each subsequent EXTRACT statement adds a new entry. Each extract entry contains the fields of the field group <fg> and, at the beginning, the fields of the field group HEADER as a sort key.

F

FETCH

Uses a cursor to read entries from a database table.

Syntax

```
FETCH NEXT CURSOR <c> INTO <target>.
```

If the cursor <c> is linked with a selection in a database table, FETCH writes the next line of the selection into the flat target area <target>.

FIELD-GROUPS

Declares a field group for an extract dataset.

Syntax

```
FIELD-GROUPS <fg>.
```

Declares the field group <fg>. Field groups define the line structure of an extract dataset. You can define a special field group called HEADER as the sort key. When you fill the extract dataset, the HEADER field group precedes each entry.

FIELD-SYMBOLS

Declares a field symbol.

Syntax

FIELD-SYMBOLS <FS> [<type>|STRUCTURE <s> DEFAULT <wa>].

Field symbols are placeholders or symbolic names for fields. The pointed brackets in the name of a field symbol are part of its syntax. The <type> addition allows you to specify a type. The STRUCTURE addition imposes a structure on the data object assigned to the field symbol.

FORM

Defines a subroutine.

Syntax

FORM <subr> [USING ... [VALUE(]<p_i>[)] [TYPE <t>|LIKE <f>]...]
[CHANGING... [VALUE(]<p_i>[)] [TYPE <t>|LIKE <f>]...].

Introduces a subroutine <form>. The USING and CHANGING additions define the parameter interface. The subroutine definition is concluded with the ENDFORM statement.

FORMAT

Sets formatting options for list output.

Syntax

FORMAT... <$option_i$> [ON|OFF]...

The formatting options <$option_i$> (color, for example) apply to all subsequent list output until they are disabled with the OFF option.

FREE

Releases memory space.

Syntax

```
FREE <itab>.

FREE MEMORY ID(<key>).

FREE OBJECT <obj>.
```

This statement deletes an internal table, a data cluster in ABAP memory, or an external object in OLE2 Automation, depending on the form of the statement used. It also releases the memory occupied by the object.

FUNCTION

Defines a function module.

Syntax

```
FUNCTION <func>.
```

Introduces the function module <func>. This statement does not have to be entered in the ABAP Editor, but is automatically generated by the Function Builder in the ABAP Workbench. The function module definition is concluded with the ENDFUNCTION statement.

FUNCTION-POOL

Introduces a function group.

Syntax

```
FUNCTION-POOL.
```

The first statement in a function group. This statement does not have to be entered by hand, but is generated automatically by the Function Builder in the ABAP Workbench. A function group is an ABAP program that contains function modules.

G

GET

Event keyword that defines event blocks for reporting events.

Syntax

GET <node> [FIELDS $<f_1>$ $<f_2>$...].

Only occurs in executable programs. When the logical database has passed a line of the node <node> to the program, the runtime environment triggers the GET event, and the corresponding event block is executed. The FIELDS addition allows you to specify explicitly the columns of the node that the logical database should retrieve.

GET BIT

Reads an individual bit.

Syntax

```
GET BIT <n> OF <f> INTO <g>.
```

Reads the bit at position <n> of the hexadecimal field <f> into the field <b>.

GET CURSOR

Determines the cursor position on a screen or in an interactive list event.

Syntax

```
GET CURSOR FIELD <f> [OFFSET <off>]
                     [LINE <lin>]
                     [VALUE <val>]
                     [LENGTH <len>].

GET CURSOR LINE <lin> [OFFSET <off>]
                      [VALUE <val>]
                      [LENGTH <len>].
```

At a user action on a list or screen, the statement writes the position, value, and displayed length of a field or line into the corresponding variables.

GET LOCALE LANGUAGE

Finds out the current text environment.

Syntax

```
GET LOCALE LANGUAGE <lg> COUNTY <c> MODIFIER <m>.
```

Returns the current language, country ID and any modifier into the corresponding variables.

GET PARAMETER

Finds out the value of a SPA/GPA parameter.

Syntax

```
GET PARAMETER ID <pid> FIELD <f>.
```

Places the value of the SPA/GPA parameter <pid> from the user-specific SAP memory into the variable <f>.

GET PF-STATUS

Finds out the current GUI status.

Syntax

```
GET PF-STATUS <f> [PROGRAM <prog>] [EXCLUDING <itab>].
```

Returns the name of the current GUI status (the same as SY-PFKEY) into the variable <f>. The PROGRAM addition writes the name of the ABAP program to which the status belongs into the variable <prog>. The EXCLUDING addition returns a list of all currently inactive function codes into the internal table <itab>.

GET PROPERTY

Finds out a property of an OLE2 Automation object.

Syntax

```
GET PROPERTY OF <obj> <p> = <f>.
```

Returns the property <p> of an external OLE2 Automation object to the variable <f>.

GET RUN TIME FIELD

Measures the runtime in microseconds.

Syntax

```
GET RUN TIME FIELD <f>.
```

The first time the statement is executed, the variable <f> is set to zero. In each further call, the runtime since the first call is written to <f>.

GET TIME

Synchronizes the time.

Syntax

```
GET TIME [FIELD <f>].
```

Refreshes the system fields SY-UZEIT, SY-DATUM, SY-TIMLO, SY-DATLO, and SY-ZONLO. If you use the FIELD addition, the variable <f> is filled with the current time.

GET TIME STAMP FIELD

Returns a time stamp.

Syntax

```
GET TIME STAMP FIELD <f>.
```

Returns the short or long form of the current date and time, depending on whether the variable <f> has the type P(8) or P(11). The long form returns the time correct to seven decimal places.

H

HIDE

Stores information about list lines.

Syntax

```
HIDE <f>.
```

During list creation, this statement stores the contents of the field <f> and the current line number in the internal HIDE area. When the cursor is positioned on a line in an interactive list event, the stored value is returned to the field <f>.

I

IF

Conditional branching.

Introduces a new branch.

Syntax

```
IF <logexp>.
```

Opens an IF control structure that must be concluded with ENDIF. The system evaluates the logical expression <logexp>, and processes different statement blocks depending on the result.

IMPORT

Imports a data cluster.

Syntax

IMPORT... <f_i> [TO <g_i>]... | (<itab>)
FROM MEMORY
| DATABASE <dbtab>(<ar>) ID(<key>)
| SHARED BUFFER <dbtab>(<ar>) ID(<key>).

The data objects <f_i> or the objects listed in the table <itab> are written from data clusters in the cross-program ABAP memory of the current internal session, a cluster database table <dbtab>, or the cross-transaction application buffer of the table <dbtab> into the variables <f_i> or <g_i>.

IMPORT DIRECTORY

Creates the directory of a data cluster from a cluster database.

Syntax

IMPORT DIRECTORY INTO <itab>
FROM DATABASE <dbtab>(<ar>)
Id <key>.

The statement creates a directory of the data objects in a data cluster of the cluster database <dbtab> and writes it to the internal table <itab>.

In the third variant, the table <itab> contains a directory of the objects stored using EXPORT TO DATABASE.

INCLUDE

Inserts an include program in another program.

Syntax

```
INCLUDE <incl>.
```

This has the same effect as inserting the source code of the include program <incl> at the same position in the program as the INCLUDE statement. Includes are not loaded dynamically at

runtime, but are automatically expanded when the program is loaded. An include must have the program type I.

INCLUDE STRUCTURE

Includes a structure within another.

Syntax

```
INCLUDE STRUCTURE <s>|TYPE <t>.
```

Adopts the structure of an ABAP Dictionary structure <s> or a structured data type <t> as part of a new structure declared using DATA BEGIN OF …

INITIALIZATION

Event keyword that defines an event block for a reporting event.

Syntax

```
INITIALIZATION.
```

Only occurs in executable programs. The ABAP runtime environment triggers the INITIALIZATION event before the selection screen is processed, at which point the corresponding event block is processed.

INSERT for Database Tables

Inserts lines into a database table.

Syntax

INSERT <dbtab> FROM <wa>.

INSERT <dbtab> FROM TABLE <itab> [ACCEPTING DUPLICATE KEYS].

Inserts one line from the work area <wa> or several lines from the internal table <itab> into the database table <dbtab>. The addition ACCEPTING DUPLICATE KEYS prevents a runtime error from occurring if two entries have the same primary key. Instead, it merely discards the duplicate.

INSERT for Field Groups

Defines the structure of field groups for extract datasets.

Syntax

INSERT <f_1>... <f_n> INTO <fg>.

Includes the fields <f_i> in the field group <fg>, thus defining a line structure for an extract dataset.

INSERT for any Internal Table

Inserts lines in an internal table of any type.

Syntax

INSERT <line>|LINES OF <jtab> [FROM <n_1>] [TO <n_2>]
INTO TABLE <itab>.

Inserts a line <line> or a set of lines from the internal table <jtab> into the internal table <itab>. If <jtab> is an index table, you can use the FROM and TO additions to restrict the lines inserted.

INSERT for Index Tables

Inserts lines in index tables.

Syntax

INSERT <line>|LINES OF <jtab> [FROM <n_1>] [TO <n_2>]
INTO <itab> [INDEX <idx>].

Inserts a line <line> or a set of lines from an internal table <jtab> into the internal table <itab> before the line with the index <idx>. If <jtab> is an index table, you can restrict the lines to be inserted using the FROM and TO additions. If you omit the INDEX addition, you can only use the statement within a LOOP construction. In this case, the new line is inserted before the current line.

INSERT for Programs

Inserts ABAP programs into the program library.

Syntax

INSERT REPORT <prog> FROM <itab>.

The lines of the internal table <itab> are added to the program library as the program <prog>.

INTERFACE

Defines an interface in ABAP Objects.

Syntax

```
INTERFACE <ifac> [DEFERRED]
                 [LOAD].
```

Introduces the definition of the interface <interface>. The definition concludes with ENDINTERFACE, and contains the declaration of all of the components in the interface. You can use the DEFERRED addition to declare the interface before you actually define it. The LOAD addition loads the interface definition explicitly from the class library.

INTERFACES

Implements interfaces in a class.

Syntax

```
INTERFACES <ifac>.
```

Used in a class declaration: This statement adds the components of the interface to the existing class definition.

Used in an interface definition: Forms a compound interface.

L

LEAVE for Screens

Leaves a screen.

Syntax

```
LEAVE SCREEN.
```

Terminates the current screen and calls the next screen. The next screen can either be defined statically in the screen attributes or set dynamically using the SET SCREEN statement.

Syntax

```
LEAVE TO SCREEN <scr>.
```

Terminates the current screen and calls the dynamically-defined next screen <scr>.

LEAVE for Lists During Screen Processing

Switches between screen and list processing.

Syntax

```
LEAVE TO LIST-PROCESSING [AND RETURN TO SCREEN <scr>].
```

This statement allows you to create and display a list while processing a series of screens. The addition allows you to specify the next screen (to which you return after the list has been displayed). If you do not use the addition, screen processing resumes with the PBO of the current screen.

Syntax

```
LEAVE LIST-PROCESSING.
```

Allows you to switch back explicitly from list processing to screen processing.

LEAVE for Programs

Terminates an ABAP program.

Syntax

```
LEAVE [PROGRAM].
```

Terminates the current program and returns to the point from which it was called.

Syntax

```
LEAVE TO TRANSACTION <tcod> [AND SKIP FIRST SCREEN].
```

Terminates the current program and starts a new transaction <tcod>. The addition allows you to skip the initial screen of the transaction.

LOCAL

Protects global data against changes.

Syntax

```
LOCAL <f>.
```

Only occurs in subroutines. When the subroutine starts, the value of <f> is stored temporarily, and restored to the variable <f> at the end of the subroutine.

LOOP Through Extracts

Starts a loop through an extract dataset.

Syntax

```
LOOP.
```

Loops through an extract dataset. The loop is concluded with ENDLOOP. When the LOOP statement is executed, the system finishes creating the extract dataset, and loops through all of its entries. One entry is read in each loop pass. The values of the extracted data are placed in the output fields of the field group within the loop.

LOOP Through Internal Tables

Starts a loop through an internal table.

Syntax

LOOP AT <itab> INTO <wa> WHERE <logexp>.

LOOP AT <itab> ASSIGNING <FS> WHERE <logexp>.

LOOP AT <itab> TRANSPORTING NO FIELDS WHERE <logexp>.

Loops through an internal table. The loop is concluded with ENDLOOP. If the logical expression <logexp> is true, the current line contents are either placed in the work area <wa>, assigned to the field symbol <FS>, or not assigned at all. The first operand in each part of <logexp> must be a component of the internal table. The pointed brackets in the field symbol name are part of its syntax.

With index tables, you can use the additions FROM <n> and TO <n> to restrict the lines that are read by specifying an index range.

LOOP Through Screen Fields

Starts a loop through the special table SCREEN.

Syntax

```
LOOP AT SCREEN...
```

Similar to a loop through an internal table. The system table SCREEN contains the names and attributes of all of the fields on the current screen.

M

MESSAGE

Outputs a message.

Syntax

```
MESSAGE <xnnn> [WITH <f1>... <f4>] [RAISING <except>].

MESSAGE ID <mid> TYPE <x> NUMBER <nnn>.

MESSAGE <xnnn>(<mid>).
```

Outputs the message <nnn> of message class <mid> as message type <x>. The message type determines how the message is displayed, and how the program reacts. The WITH addition allows you to fill placeholders in the message text. The RAISING addition in function modules and methods allows you to terminate the procedure and trigger the exception <exception>.

METHOD

Introduces the implementation of a method in a class.

Syntax

```
METHOD <meth>.
```

Only occurs in the implementation part of classes. This statement begins a statement block that must be concluded with ENDMETHOD. You do not have to specify any interface parameters, since these are defined in the method declaration.

METHODS

Declares methods in classes and interfaces.

Syntax

METHODS <meth>
IMPORTING... [VALUE(]<i_i>[)] TYPE <t> [OPTIONAL]...
EXPORTING... [VALUE(]<e_i>[)] TYPE <t> [OPTIONAL]...
CHANGING ... [VALUE(]<c_i>[)] TYPE <t> [OPTIONAL]...
RETURNING VALUE(<r>)
EXCEPTIONS ... <e_i>...

Declares a method <meth> in the declaration part of a class or in an interface definition. The additions define the parameter interface and exceptions of the method. The function of the method must be implemented using the METHOD statement.

MODIFY for Database Tables

Inserts or changes lines in database tables.

Syntax

MODIFY <dbtab> FROM <wa>.

MODIFY <dbtab> FROM TABLE <itab>.

Works like INSERT for database tables if there is not yet a line in the table with the same primary key. Works like UPDATE if a line already exists with the same primary key.

MODIFY for All Internal Tables

Changes the contents of lines in any type of internal table.

Syntax

MODIFY TABLE <itab> FROM <wa> [TRANSPORTING <f_1> <f_2>...].

Copies the work area <wa> into the line of the internal table with the same table key as <wa>. You can use the TRANSPORTING addition to specify the exact components that you want to change.

MODIFY <itab> FROM <wa> TRANSPORTING <f_1> <f_2>... WHERE <logexp>.

Copies the work area <wa> into the lines of the internal table for which the logical expression is true. The first operand in each comparison of the logical expression must be a component of the line structure.

MODIFY for Index Tables

Changes the contents of lines in index tables.

Syntax

MODIFY <itab> FROM <wa> [INDEX <idx>] [TRANSPORTING <f_1> <f_2>...].

Copies the work area <wa> into the line of the internal table with index <idx>. If you omit the INDEX addition, you can only use the statement within a LOOP. This changes the current line.

MODIFY for Lists

Changes a line of a list.

Syntax

```
MODIFY LINE <n> [INDEX <idx>] [OF CURRENT PAGE|OF PAGE <p>]
    |CURRENT LINE
    LINE FORMAT <option1> <option2>...
    FIELD VALUE <f1> [FROM <g1>] <f2> [FROM <g2>]...
    FIELD FORMAT <f1> <options1> <f2> <options2>
```

Changes either line <n> on the current or specified list (or page), or the last line to be chosen. The exact nature of the change is specified in the additions.

MODIFY SCREEN

Changes the table SCREEN.

Syntax

```
MODIFY SCREEN...
```

Like changing an internal table. This statement allows you to change the attributes of fields on the current screen.

MODULE

Introduces a dialog module.

Syntax

```
MODULE <mod> OUTPUT |[INPUT].
```

Introduces the dialog module <mod>. The OUTPUT and INPUT additions designate the module as a PBO or PAI module respectively. Each module must conclude with the ENDMODULE statement.

MOVE

Assigns values.

Syntax

```
MOVE <f1> TO <f2>.
```

Assigns the contents of the data object <f1> to the variable <f2>, with automatic type conversion if necessary. Equivalent to <f2> = <f1>.

MOVE-CORRESPONDING

Assigns values between identically-named components of structures.

Syntax

```
MOVE-CORRESPONDING <struc1> TO <struc2>.
```

The contents of the components of the structure <struc1> are assigned to the identically-named components of the structure <struc2>.

MULTIPLY

Multiplies two individual fields.

Syntax

```
MULTIPLY <n> BY <m>.
```

The contents of <n> are multiplied by the contents of <m>, and the result is placed in <m>. Equivalent is M = m * n.

MULTIPLY-CORRESPONDING

Multiplies components of structures.

Syntax

```
MULTIPLY-CORRESPONDING <struc1> BY <struc2>.
```

Multiplies all of the identically-named components of <struc1> and <struc2> and places the results in the components in <struc1>.

N

NEW-LINE

Inserts a line break in a list.

Syntax

```
NEW-LINE [NO-SCROLLING|SCROLLING].
```

Positions the list output on a new line. The NO-SCROLLING addition locks the line against horizontal scrolling. To lift the lock, use the SCROLLING addition.

NEW-PAGE

Inserts a page break in a list.

Syntax

```
NEW-PAGE [NO-TITLE|WITH-TITLE]
         [NO-HEADING|WITH-HEADING]
         [LINE-COUNT]
         [LINE-SIZE]
         [PRINT ON|OFF].
```

Generates a new page and positions the list output after the page header. The additions control how the page header is displayed, the length and width of the page, and the print output.

NODES

Declares an interface work area.

Syntax

```
NODES <node>.
```

Declares a variable with the same data type and the same name as a data type from the ABAP Dictionary. Structures in main programs and subroutines declared with nodes use a common data area. This statement is used in conjunction with logical databases.

O

ON CHANGE

Introduces a branch.

Syntax

```
ON CHANGE OF <f> [OR <f1> OR <f2>...].
```

Opens an ON control structure, concluded with ENDON. The statement block is executed whenever the contents of the field <f> or one of the other fields <fi> has changed since the statement was last executed.

OPEN CURSOR

Opens a database cursor.

Syntax

```
OPEN CURSOR [WITH HOLD] <c> FOR SELECT      <result>
                        FROM      <source>
                        [WHERE    <condition>]
                        [GROUP BY <fields>]
                        [HAVING   <cond>]
                        [ORDER BY <fields>].
```

Opens a cursor <c> with type CURSOR for a SELECT statement. All of the clauses of the SELECT statement apart from the INTO clause can be used. The INTO clause is set in the FETCH statement. If you use the WITH HOLD addition, the cursor is not closed when a database commit occurs.

OPEN DATASET

Opens a file.

Syntax

```
OPEN DATASET <dsn> [FOR INPUT|OUTPUT|APPENDING]
                   [IN BINARY|TEXT MODE]
                   [AT POSITION <pos>]
                   [MESSAGE <mess>]
                   [FILTER <filt>].
```

Opens a file <dsn> on the application server. The additions determine whether the file is for reading or writing, whether the contents are to be interpreted in binary or character form, the position in the file, the location to which an operating system can be written, and allow you to execute an operating system command.

OVERLAY

Overlays a character string with another.

Syntax

```
OVERLAY <c1> WITH <c2> [ONLY <str>].
```

All of the characters in field <c1> that occur in <str> are overlaid with the contents of <c2>. <c2> remains unchanged. If you omit the ONLY <str> addition, all positions in <c1> containing spaces are overwritten.

P

PACK

Converts type C variables into type P.

Syntax

```
PACK <f> TO <g>.
```

Packs the string <f> and places it in the field <g>. This can be reversed with the UNPACK statement.

PARAMETERS

Declares parameters for a selection screen.

Syntax

```
PARAMETERS <p>[(<length>)] [TYPE <type>|LIKE <obj>] [DECIMALS <d>]
               [DEFAULT <f>]
               [MEMORY ID <pid>]
               [LOWER CASE]
               [OBLIGATORY]
               [VALUE CHECK]
               [AS CHECKBOX]
               [RADIOBUTTON GROUP <radi>]
               [NO-DISPLAY]
               [MODIF ID <key>].
```

Declares a variable <p>, as in the DATA statement. However, the PARAMETERS statement also creates an input field for <p> on the relevant selection screen. You can use the additions to define default values, accept lowercase input, define the field as required, check values, define a checkbox or radio button, prevent the field from being displayed on the selection screen, or modify the field.

PERFORM

Calls a subroutine.

Syntax

PERFORM <subr>
| <subr>(<prog>) [IF FOUND]
|(<fsubr>)[IN PROGRAM (<fprog>)][IF FOUND]
[USING ... <p_i>...]
[CHANGING... <p_i>...]
[ON COMMIT].

Calls an internal or external subroutine <subr> or the subroutine whose name occurs in the <fsubr> field. The external program is <prog> or the name contained in <fprog>. The IF FOUND addition prevents a runtime error from occurring if the subroutine does not exist. You must use the USING and CHANGING additions to supply values to the interface parameters of the subroutine. The ON COMMIT addition delays the execution of the subroutine until the next COMMIT WORK statement.

POSITION

Absolute positioning of the output on a list.

Syntax

```
POSITION <col>.
```

Positions the list output in column <col>.

PRINT-CONTROL for Print Format

Specifies the print format.

Syntax

PRINT-CONTROL <formats> [LINE <lin>] [POSITION <col>].

Sets the print format starting either at the current list position or at line <lin> and column <col>.

PRINT-CONTROL for Index Lines

Creates index lines in the spool file.

Syntax

```
PRINT-CONTROL INDEX-LINE <f>.
```

Writes the contents of the field <f> into an index line at the end of the current print line. The index line is not printed. In optical archiving, the spool system separates the lists into a data file and a description file containing the index lines.

PRIVATE

Defines the private section of a class.

Syntax

```
PRIVATE SECTION.
```

Introduces the declaration of all of the components of a class that are only visible in the class itself.

PROGRAM

Introduces a program.

Syntax

```
PROGRAM <prog>...
```

The first statement in some ABAP programs. Equivalent of REPORT.

PROTECTED

Defines the protected section of a class.

Syntax

```
PROTECTED SECTION.
```

Introduces the declaration of all of the components of a class that are only visible in the class and its subclasses.

PROVIDE

Loops through internal tables at given intervals.

Syntax

```
PROVIDE <f1>... <fn> FROM <itab1>
        <g1>... <gm> FROM <itab2>
         ...         FROM <itabn>
         ... BETWEEN <f> AND <g>.
```

The contents of the specified fields of the internal tables <itab1> … <itabn> are placed in their header lines. Then, the processing block between PROVIDE and ENDPROVIDE is executed for each interval.

PUBLIC

Defines the public section of a class.

Syntax

```
PUBLIC SECTION.
```

Introduces the declaration of all components of a class that are visible in the class, its subclasses, and for all users.

PUT

Triggers a GET event.

Syntax

```
PUT <node>.
```

Only occurs in logical databases. Branches the program flow according to the structure of the logical database.

R

RAISE for Exceptions

Triggers an exception.

Syntax

```
RAISE <except>.
```

Only occurs in function modules and methods. Terminates the procedure and triggers the exception <except>.

RAISE for Events

Triggers an event in ABAP Objects.

Syntax

```
RAISE EVENT <evt>.
```

Only occurs in methods. The event <evt> is triggered, and this calls all of the handler methods registered for it.

RANGES

Declares a RANGES table.

Syntax

```
RANGES <rangetab> FOR <f>.
```

Declares a RANGES table for the field <f>. RANGES tables have the same data type as a selection table, but they do not have input fields on a selection screen.

READ for Files

Reads a file.

Syntax

READ DATASET <dsn> INTO <f> [LENGTH <len>].

Reads the contents of the file <dsn> on the application server to the variable <f>. The number of bytes transferred can be written to <len>.

READ for any Internal Table

Reads a line from any internal table.

Syntax

```
READ TABLE <itab>  FROM <wa>
              |WITH TABLE KEY <k1> = <f1>... <kn> = <fn>
              |WITH KEY = <f>
              |WITH KEY <k1> = <f1>... <kn> = <fn>
    INTO <wa> [COMPARING <f1> <f2>... |ALL FIELDS]
           [TRANSPORTING <f1> <f2>... |ALL FIELDS|NO FIELDS]
   |ASSIGNING <FS>.
```

This statement reads either the line of the internal table with the same key as specified in the work area <wa>, the line with the key specified in the TABLE KEY addition, the line that corresponds fully to <f>, or the one corresponding to the freely-defined key in the KEY addition. The contents of the line are either written to the work area <wa>, or the line is assigned to the field symbol <FS>. If you assign the line to a work area, you can compare field contents and specify the fields that you want to transport.

READ for Index Tables

Reads a line of an index table.

Syntax

```
READ TABLE <itab> INDEX <idx> INTO <wa>... | ASSIGNING <FS>.
```

Reads the line with the index <idx>. The result is available as described above.

READ for Lists

Reads the contents of a line from a list.

Syntax

```
READ LINE  <n> [INDEX <idx>] [OF CURRENT PAGE|OF PAGE <p>]
       |CURRENT LINE
       [FIELD VALUE <f1> [INTO <g1>]... <fn> [INTO <gn>]].
```

Reads either the line <n> on the current or specified list or page, or the last line to have been selected by the user. The addition specifies the fields that you want to read, and the target fields into which they should be placed. The entire line is always placed in the system field SY-LISEL, and the HIDE area is filled for the line.

READ for Programs

Reads ABAP programs from the program library.

Syntax

READ REPORT <prog> INTO <itab>.

Copies the lines of the program <prog> into the internal table <itab>.

RECEIVE

Receives results from an asynchronous function module call.

Syntax

```
RECEIVE RESULTS FROM FUNCTION <func> [KEEPING TASK]
                      [IMPORTING ... f_i = a_i... ]
                      [TABLES    ... f_i = a_i... ]
                      [EXCEPTIONS... e_i = r_i... ]
```

Occurs in special subroutines to receive IMPORTING and TABLES parameters from function modules called using the STARTING NEW TASK addition.

REFRESH

Initializes an internal table.

Syntax

```
REFRESH <itab>.
```

Resets the internal table <itab> to its initial value, that is, deletes all of its lines.

REFRESH CONTROL

Initializes a control.

Syntax

```
REFRESH CONTROL <ctrl> FROM SCREEN <scr>.
```

The control <ctrl> defined in the CONTROLS statement is reset with the initial values specified for screen <scr>.

REJECT

Terminates a GET processing block.

Syntax

```
REJECT [<dbtab>].
```

Stops processing the current line of the node of the logical database. If you specify <dbtab>, the logical database reads the next line of the node <dbtab>, otherwise the next line of the current node.

REPLACE

Replaces strings in fields with another string.

Syntax

```
REPLACE <str1> WITH <str2> INTO <c> [LENGTH <l>].
```

This statement searches the first occurrence of the first <l> characters of the search pattern <str1> in field <c> and replaces them with the string <str2>.

REPORT

Introduces a program.

Syntax

```
REPORT <rep> [MESSAGE-ID <mid>]
             [NO STANDARD PAGE HEADING]
             [LINE-SIZE <col>]
             [LINE-COUNT <n>(<m>)]
             [DEFINING DATABASE <ldb>].
```

This is the first statement within certain ABAP programs. <rep> can be any name you choose. The addition MESSAGE-ID specifies a message class to be used in the program. The DEFINING DATABASE addition defines the program as the database program of the logical database <ldb>. The other options are formatting specifications for the default list of the program.

RESERVE

Conditional page break in a list.

Syntax

```
RESERVE <n> LINES.
```

Executes a page break on the current page if less than <n> lines are free between the current line and the page footer.

ROLLBACK

Undoes the changes in a SAP LUW.

Syntax

```
ROLLBACK WORK.
```

Undoes all changes within a database LUW to the beginning of the LUW. The registered update modules are not executed, and the record entry is deleted from table VBLOG.

S

SCROLL

Scrolls in a list.

Syntax

SCROLL LIST FORWARD|BACKWARD [INDEX <idx>].

SCROLL LIST TO FIRST PAGE|LAST PAGE|PAGE <pag>
[INDEX <idx>] [LINE <lin>].

SCROLL LIST LEFT|RIGHT [BY <n> PLACES] [INDEX <idx>].

SCROLL LIST TO COLUMN <col> [INDEX <idx>].

Positions the current list or the list level <idx> in accordance with the additions specified. You can scroll by window, page, columns, or to the left- or right-hand edge of the list.

SEARCH

Searches for a string.

Syntax

```
SEARCH <f>|<itab> FOR <g> [ABBREVIATED]
                          [STARTING AT <n1>]
                          [ENDING AT <n2>]
                          [AND MARK].
```

Searches the field <f> or the table <itab> for the string in field <g>. The result is placed in the system field SY-FDPOS. The additions allow you to hide intermediate characters, search from and to a particular position, and convert the found string into uppercase.

SELECT

Reads data from the database.

Syntax

SELECT <result>
 INTO <target>
 FROM <source>
 [WHERE <condition>]
 [GROUP BY <fields>]
 [HAVING <cond>]
 [ORDER BY <fields>].

The SELECT statement consists of a series of clauses, each of which fulfils a certain task:

SELECT clause

Defines the structure of the selection.

ABAP Statement Overview

Syntax

```
SELECT [SINGLE]|[DISTINCT]
    * | <si> [AS <ai>]... <agg>( [DISTINCT] <sj>) [AS <aj>]...
```

The selection can be a single line SINGLE or a series of lines. You can eliminate duplicate lines using the DISTINCT addition. To select the entire line, use *, otherwise, you can specify individual columns <s_i>. For individual columns, you can use aggregate functions <agg>, and assign alternative column names <a_i>.

INTO clause

Defines the target area into which the selection from the SELECT clause is to be placed.

Syntax

```
...  INTO [CORRESPONDING FIELDS OF] <wa>
  | INTO|APPENDING [CORRESPONDING FIELDS OF] TABLE <itab>
                                [PACKAGE SIZE <n>]
  | INTO (<f1>, <f2>,...)
```

The target area can be a flat work area <wa>, an internal table <itab>, or a list of fields <f_i>. If you use the CORRESPONDING FIELDS addition, data is only selected if there is an identically-named field in the target area. If you use APPENDING instead of INTO, the data is appended to an internal table instead of overwriting the existing contents. PACKAGE SIZE allows you to overwrite or extend the internal table in a series of packages. The data type of the target area must be appropriate for the selection in the SELECT clause.

FROM clause

Specifies the database tables from which the data in the selection in the SELECT clause is to be read.

Syntax

```
... FROM [<tab> [INNER]|LEFT [OUTER] JOIN] <dbtab> [AS <alias>]
                        [ON <cond>]
      [CLIENT SPECIFIED]
      [BYPASSING BUFFER]
      [UP TO <n> ROWS]
```

You can read a single table <dbtab> or more than one table, using inner and outer joins to link tables with conditions <cond>, where <tab> is a single table or itself a join condition. You can specify individual database tables either statically or dynamically, and you can replace their names with an alternative <alias>. You can bypass automatic client handling with the CLIENT SPECIFIED addition, and SAP buffering with BYPASSING BUFFER. You can also restrict the number of lines read from the table using the UP TO <n> ROWS addition.

WHERE clause

Restricts the number of lines selected.

Syntax

```
... [FOR ALL ENTRIES IN <itab>] WHERE <cond>
```

The condition <cond> may contain one or more comparisons, tests for belonging to intervals, value list checks, subqueries, selection table queries or null value checks, all linked with AND, OR, and NOT. If you use the FOR ALL ENTRIES addition, the condition <cond> is checked for each line of the internal table <itab> as long as <cond> contains a field of the internal table as an

operand. For each line of the internal table, the lines from the database table meeting the condition are selected. The result set is the union of the individual selections resulting from each line.

GROUP BY clause

Groups lines in the selection

Syntax

... GROUP BY <s_1> <s_2>

Groups lines with the same contents in the specified columns. Uses aggregate functions for all other columns in each group. All columns listed in the SELECT clause that do not appear in the GROUP BY addition must be specified in aggregate expressions.

HAVING clause

Restricts the number of line groups.

Syntax

... HAVING <cond>

Like the WHERE clause, but can only be used in conjunction with a GROUP BY clause. Applies conditions to aggregate expressions to reduce the number of groups selected.

ORDER BY clause

Sorts the lines in the selection.

Syntax

... ORDER BY PRIMARY KEY |... <s_i> [ASCENDING|DESCENDING]...

Sorts the selection in ascending or descending order according to the primary key or the contents of the fields listed.

SELECT-OPTIONS

Declares selection criteria for a selection screen.

Syntax

```
SELECT-OPTIONS <sel> FOR <f>
               [DEFAULT <g> [to <h>] [OPTION <op>] SIGN <s>]
               [MEMORY ID <pid>]
               [LOWER CASE]
               [OBLIGATORY]
               [NO-DISPLAY]
               [MODIF ID <key>]
               [NO-EXTENSION]
               [NO INTERVALS]
               [NO DATABASE SELECTION].
```

Declares a selection table <sel> for the field <f>, and also places input fields on the corresponding selection screen. The additions allow you to set a default value, accept input in lowercase, define a required field, suppress or modify the display on the selection screen, restrict

the selection table to a line or a selection to a single field, or prevent input from being passed to a logical database.

SELECTION-SCREEN for Selection Screen Formatting

Formats a selection screen.

Syntax

```
SELECTION-SCREEN SKIP [<n>].

SELECTION-SCREEN ULINE [[/]<pos(len)>] [MODIF ID <key>].

SELECTION-SCREEN COMMENT [/]<pos(len)> <comm> [FOR FIELD <f>]
                                [MODIF ID <key>].

SELECTION-SCREEN BEGIN OF LINE.
 ...
SELECTION-SCREEN END OF LINE.

SELECTION-SCREEN BEGIN OF BLOCK <block>
                        [WITH FRAME [TITLE <title>]]
                        [NO INTERVALS].
 ...
SELECTION-SCREEN END OF BLOCK <block>.

SELECTION-SCREEN FUNCTION KEY <i>.

SELECTION SCREEN PUSHBUTTON [/]<pos(len)> <push>
                    USER-COMMAND <ucom> [MODIF ID <key>].
```

Allows you to insert blank lines, lines and comments, group input fields together in lines and blocks, and create pushbuttons.

SELECTION-SCREEN for Defining Selection Screens

Defines selection screens.

Syntax

```
SELECTION-SCREEN BEGIN OF <numb> [TITLE <tit>] [AS WINDOW].
 ...
SELECTION-SCREEN END OF <numb>.
```

Defines a selection screen with the screen number <numb>. All PARAMETERS, SELECT-OPTIONS, and SELECTION-SCREEN statements within the SELECTION-SCREEN BEGIN OF … END OF block belong to the selection screen <numb>. The TITLE addition allows you to define a title for the selection screen. The AS WINDOW addition allows you to define the selection screen as a modal dialog box.

SELECTION-SCREEN for Selection Screen Versions

Defines selection screen versions.

Syntax

```
SELECTION-SCREEN BEGIN OF VERSION <dynnr>
 ...
  SELECTION-SCREEN EXCLUDE <f>.
 ...
SELECTION-SCREEN BEGIN OF VERSION <dynnr>.
```

Only in logical databases. The statement hides fields that otherwise appear on the standard selection screen.

SELECTION-SCREEN for Logical Databases

Provides special functions in conjunction with logical databases.

Syntax

```
SELECTION-SCREEN DYNAMIC SELECTIONS | FIELD SELECTION
                FOR NODE|TABLE <node>.
```

Can only be used in logical databases. Declares a node as accepting dynamic selections or field selections.

SET BIT

Sets an individual bit.

Syntax

```
SET BIT <n> OF <f> [TO <b>].
```

Sets t he bit at position <n> of hexadecimal field <f> to 1 or the value of the field <b>. <b> must be 0 or 1.

SET BLANK LINES

Allows blank lines in lists.

Syntax

```
SET BLANK LINES ON|OFF.
```

Prevents blank lines created in WRITE statements from being suppressed in list output.

SET COUNTRY

Sets the output format

Syntax

```
SET COUNTRY <c>.
```

Sets the output formats for numeric and date fields for the country with the ID <c>.

SET CURSOR

Sets the cursor on the screen.

Syntax

SET CURSOR FIELD <f> [OFFSET <off>]
[LINE <lin>].

SET CURSOR LINE <lin> [OFFSET <off>].

SET CURSOR <col> <line>.

Sets the cursor either to a particular position in a field, line, or column of a line.

SET EXTENDED CHECK

Affects the extended program check.

Syntax

SET EXTENDED CHECK ON|OFF.

Switches the extended program check (SLIN) on or off, suppressing the corresponding messages.

SET HANDLER

Registers event handlers in ABAP Objects.

Syntax

SET HANDLER... <h_i>... [FOR <ref>|FOR ALL INSTANCES].

If you do not use the FOR addition, the handler is set for all static events. Use the FOR addition to register handlers for instance events.

SET HOLD DATA

Sets a screen attribute.

Syntax

SET HOLD DATA ON|OFF.

Sets the screen attribute “Hold data” from the program.

SET LANGUAGE

Sets the display language.

Syntax

SET LANGUAGE <lg>.

All text symbols are refreshed with the contents of the text pool in language <lg>.

SET LEFT SCROLL BOUNDARY

Sets the left-hand boundary for horizontal scrolling.

Syntax

SET LEFT SCROLL-BOUNDARY [COLUMN <col>].

Sets the current output position or the position <col> as the left-hand edge of the scrollable area on the current list page.

SET LOCALE LANGUAGE

Sets the text environment.

Syntax

SET LOCALE LANGUAGE <lg> [COUNTRY <c>] [MODIFIER <m>].

Sets the text environment for alphabetical sorting according to the language <lg>, country <c>, and any further modifier <m>.

SET MARGIN

Sets the margin of a print page.

SET MARGIN <x> [<y>].

Sends the current list page to the spool system with a margin of <x> columns from the left-hand edge and <y> rows from the top edge of the page.

SET PARAMETER

Sets a SPA/GPA parameter.

Syntax

```
SET PARAMETER ID <pid> FIELD <f>.
```

Copies the value of the field <f> into the SPA/GPA parameter <pid> in the user-specific SAP memory.

SET PF-STATUS

Sets the GUI status.

Syntax

SET PF-STATUS <stat> [EXCLUDING <f>|<itab>]
[IMMEDIATELY] [OF PROGRAM <prog>].

Sets the GUI status <stat> for the subsequent screens. The EXCLUDING addition allows you to deactivate functions dynamically. The IMMEDIATELY addition sets the GUI status of the list

currently displayed. The OF PROGRAM addition allows you to use a GUI status from another program.

SET PROPERTY

Sets a property of an OLE2 Automation object.

Syntax

```
GET PROPERTY OF <obj> <p> = <f>.
```

Sets the property <p> of an external OLE2 Automation object to <f>.

SET RUN TIME ANALYZER

Controls the runtime analysis.

Syntax

SET RUN TIME ANALYZER ON|OFF.

The runtime analysis only measures the runtime of the statements in the block between SET RUN TIME ANALYZER ON and OFF.

SET RUN TIME CLOCK

Sets the clock accuracy for runtime analysis.

Syntax

SET RUN TIME CLOCK RESOLUTION HIGH|LOW.

Sets the accuracy of the runtime to low accuracy with long measurement interval or high accuracy with shorter measurement interval.

SET SCREEN

Sets the next screen.

Syntax

SET SCREEN <scr>.

Temporarily overwrites the statically-defined next screen with <scr>. <scr> is processed after the current screen.

SET TITLEBAR

Sets the screen title.

Syntax

SET TITLEBAR <t> [OF PROGRAM <prog>] [WITH <g1 >... <g9>].

Sets the title <t> for the subsequent screens. The OF PROGRAM addition allows you to use a title from another program. The WITH addition fills any placeholders in the title.

SET UPDATE TASK LOCAL

Switches on local update.

Syntax

SET UPDATE TASK LOCAL.

Updates are processed in the current work process.

SET USER-COMMAND

Triggers a list event.

Syntax

SET USER-COMMAND <fc>.

Triggers a list event with the function code <fc> and calls the corresponding event block.

SHIFT

Shifts strings.

Syntax

SHIFT <c> [BY <n> PLACES] [LEFT|RIGHT|CIRCULAR].

Shifts the field <c> by one or <n> places. The additions allow you to specify the direction, and how the empty spaces are dealt with.

SKIP for Blank Lines

Creates blank lines on the output list.

Syntax

```
SKIP [<n>].
```

Creates <n> blank lines after the current line in a list. If you omit <n>, inserts one line.

SKIP for Positioning

Absolute positioning for output on a list.

Syntax

```
SKIP TO LINE <lin>.
```

Positions the list output in line <lin>.

SORT for Extracts

Sorts an extract dataset.

Syntax

SORT [ASCENDING|DESCENDING] [AS TEXT] [STABLE]
... BY <f_i> [ASCENDING|DESCENDING] [AS TEXT]...

Ends creation of the extract dataset in the program and sorts it. If you omit the BY addition, the extract is sorted by the key specified in the HEADER field group. The BY addition allows you to specify a different sort key. The other additions specify whether you want to sort in ascending or descending order, and whether strings should be sorted alphabetically.

SORT for Internal Tables

Sorts internal tables.

Syntax

SORT <itab> [ASCENDING|DESCENDING] [AS TEXT] [STABLE]
... BY <f_i> [ASCENDING|DESCENDING] [AS TEXT]...

Sorts the internal table <itab>. If you omit the BY addition, the table is sorted by its key. The BY addition allows you to specify a different sort key. The remaining additions allow you to specify whether you want to sort in ascending or descending order, and whether strings should be sorted alphabetically.

SPLIT

Splits a character string.

Syntax

```
SPLIT <c> AT <del> INTO <c1>... <cn> INTO TABLE <itab>.
```

Searches the field <c> for the character <del> and places the partial fields before and after <del> into the target fields <c1> … <cn>, or into a new line of the internal table <itab>.

START-OF-SELECTION

Event keyword that defines an event block for a reporting event.

Syntax

```
START-OF-SELECTION.
```

After the selection screen has been processed, the runtime environment triggers the START-OF-SELECTION event and processes the corresponding event block.

STATICS

Defines static variables.

Syntax

```
STATICS <f>...
```

Like DATA. Retains the value of a local variable beyond the runtime of the procedure in which it occurs.

STOP

Exits a reporting event.

Syntax

```
STOP.
```

Can only occur in event blocks for reporting events. Leaves the block and jumps to END-OF-SELECTION.

SUBMIT

Calls an executable program.

Syntax

SUBMIT <rep> [AND RETURN] [VIA SELECTION-SCREEN]
[USING SELECTION-SET <var>]
[WITH <sel> <criterion>]
[WITH FREE SELECTIONS <freesel>]
[WITH SELECTION-TABLE <rspar>]
[LINE-SIZE <width>]
[LINE-COUNT <length>].

Calls the program <rep>. If you omit the AND RETURN addition, the current program is terminated, otherwise, the data from the current program is retained, and processing returns to the calling program when <rep> has finished running. The other additions control the selection screen and set attributes of the default list in the called program.

SUBTRACT for Single Fields

Subtracts two single fields.

Syntax

```
SUBTRACT <n> FROM <m>.
```

The contents of <n> are subtracted from the contents of <m> and the result placed in <m>. Equivalent of m = m - n.

SUBTRACT-CORRESPONDING

Subtracts components of structures.

Syntax

```
SUBTRACT-CORRESPONDING <struc1> FROM <struc2>.
```

Subtracts the contents of the components of <struc1> from identically-named components in <struc2> and places the results in the components of <struc2>.

SUM

Calculates sums of groups.

Syntax

SUM.

Can only be used in loops through internal tables. Calculates the sums of the numeric fields in all lines of the current control level and writes the results to the corresponding fields in the work area.

SUPPLY

Fills context instances with values.

Syntax

SUPPLY <key_1> = <f_1>... <key_n> = <f_n> TO CONTEXT <inst>.

Fills the key fields <key_n> of the context instance <inst> with the values <f_n>.

SUPPRESS DIALOG

Prevents the current screen from being displayed.

Syntax

```
SUPPRESS DIALOG.
```

Can only occur in a PBO dialog module. The screen is not displayed, but its flow logic is still processed.

T

TABLES

Declares an interface work area.

Syntax

```
TABLES <dbtab>.
```

Declares a structure with the same data type and name as a database table, a view, or a structure from the ABAP Dictionary. Structures declared using TABLES in main programs and subroutines use a common data area.

TOP-OF-PAGE

Event keyword for defining an event block for a list event.

Syntax

```
TOP-OF-PAGE [DURING LINE-SELECTION].
```

Whenever a new page begins while a standard list is being created, the runtime environment triggers the TOP-OF-PAGE event and the corresponding event block is executed. The addition DURING LINE-SELECTION has the same function, but for detail lists.

TRANSFER

Writes data to a file.

Syntax

TRANSFER <f> TO [LENGTH <len>].

Writes the field <f> to the file <dsn> on the application server. The LENGTH addition specifies the length <len> of the data you want to transfer.

TRANSLATE

Converts characters in strings.

Syntax

TRANSLATE <c> TO UPPER|LOWER CASE
|USING <r>.

The characters of the string <c> are converted into upper- or lowercase, or according to a substitution rule specified in <r>.

TYPE-POOL

Introduces a type group.

Syntax

```
TYPE-POOL <tpool>.
```

The first statement in a type group. You do not have to enter this statement in the ABAP Editor - instead, it is automatically inserted in the type group by the ABAP Dictionary. A type group is an ABAP program containing type definitions and constant declarations that can then be used in several different programs.

TYPE-POOLS

Declares the types and constants of a type group to a program.

Syntax

```
TYPE-POOLS <tpool>.
```

After this statement, you can use all of the data types and constants defined in the type group <tpool> in your program.

TYPES for Simple Field Types

Defines a simple field type.

Syntax

TYPES <t>[(<length>)] [TYPE <type>|LIKE <obj>] [DECIMALS <dec>].

Defines the internal data type <t> in the program with length <len>, reference to the ABAP Dictionary type <type> or a data object <obj>, and, where appropriate, with <dec> decimal places.

Syntax

TYPES <t> TYPE REF TO <class>|<interface>.

Defines the internal data type <t> in the program with reference to the class <class> or the interface <interface>.

TYPES for Aggregate Types

Defines aggregated types.

Syntax

TYPES: BEGIN OF <structure>,
...
<t_i>...,
...
END OF <structure>.

Combines the data types <t_i> to form the structure <structure>. You can address the individual components of a structure in a program using a hyphen between the structure name and the component name as follows: <structure>-<t_i>.

Syntax

TYPES <t> TYPE|LIKE <tabkind> OF <linetype> [WITH <key>].

Defines the local data type <t> in the program as an internal table with the access type <tabkind>, the line type <linetype>, and the key <key>.

U

ULINE

Places a horizontal line on the output list.

Syntax

ULINE [AT [/][<pos>][(<len>)]].

Without additions, generates a new line on the current list and fills it with a horizontal line. The additions allow you to insert a line break and specify the starting position and length of the line.

UNPACK

Converts variables from type P to type C.

Syntax

```
UNPACK <f> TO <g>.
```

Unpacks the packed field <f> and places it in the string <g> with leading zeros. The opposite of PACK.

UPDATE

Modifies lines in database tables.

Syntax

UPDATE <dbtab> SET <s_i> = <f>
|<s_i> = <s_i> + <f>
|<s_i> = <s_i> - <f> [WHERE <cond>].

Sets the value in <s_i> to <f>, increases it by <f>, or decreases it by <f> for all selected lines. The WHERE addition determines the lines that are updated. If you omit the WHERE addition, all lines are updated.

Syntax

UPDATE <dbtab> FROM <wa>.

UPDATE <dbtab> FROM TABLE <itab>.

Overwrites the line with the same primary key as <wa> with the contents of <wa>, or all lines with the same primary key as a line in the internal table <itab> with the corresponding line of itab. The work area <wa> or the lines of the table <itab> must have at least the same length and the same alignment as the line structure of the database table.

W

WHEN

Introduces a statement block in a CASE control structure.

Syntax

WHEN <f_1> [OR <f_2> OR...] | OTHERS.

The statement block after a WHEN statement is executed if the contents of the field <f> in the CASE statement are the same as those of one of the fields <f_i>. Processing then resumes after the ENDCASE statement. The WHEN OTHERS statement block is executed if the contents of <f> do not correspond to any of the fields <f_i>.

WHILE

Introduces a loop.

Syntax

WHILE <logexp> [VARY <f> FROM <f1> NEXT <f2>].

Introduces a statement block that is concluded with ENDWHILE. The statement block between WHILE and ENDWHILE is repeated for as long as the expression <logexp> is true, or until a termination statement such as CHECK or EXIT occurs. The VARY addition allows you to process fields that are a uniform distance apart within memory.

WINDOW

Displays a list as a modal dialog box.

Syntax

```
WINDOW STARTING AT <x1> <y1> [ENDING AT <x2> <y2>].
```

Can only be used in list processing. The current detail list is displayed as a modal dialog box. The top left-hand corner of the window is positioned at column <x1> and line <y1>. The bottom right-hand corner is positioned at column <x2> and line <y2> (if specified).

WRITE

Creates list output.

Syntax

```
WRITE [AT [/][<pos>][(<len>)]] <f> [AS CHECKBOX|SYMBOL|ICON|LINE]
                                   [QUICKINFO <g>].
                                   [<format>]
```

The contents of the field <f> are formatted according to their data type and displayed on the list. The additions before the field allow you to specify a line break, the starting position, and the length of the field. The additions after the field allow you to display checkboxes, symbols, icons, and lines. The <format> addition can contain various other formatting options. The QUICKINFO addition allows you to assign a quickinfo <g> to the field.

WRITE TO

Assigns strings.

Syntax

```
WRITE <f1> TO <f2> [<format>].
```

Converts the contents of the data object <f1> to type C and assigns the resulting string to the variable <f2>. You can use the same formatting options available in the WRITE statement.

ABAP System Fields

The ABAP system fields are active in all ABAP programs. They are filled by the runtime environment, and you can query their values in a program to find out particular states of the system. Although they are variables, you should not assign your own values to them, since this may overwrite information that is important for the normal running of the program. However, there are some isolated cases in which you may need to overwrite a system variable. For example, by assigning a new value to the field SY-LSIND, you can control navigation within details lists.

The names and data types of the system fields are contained in the ABAP Dictionary structure SYST. To address them in an ABAP program, use the form SY-<fieldname>. Within screen flow logic, you can also use the form SYST-<fieldname>.

System Fields in Alphabetical Order

System Fields in Thematic Order

System Fields in Alphabetical Order

The following table contains an alphabetical list of the fields in the ABAP Dictionary structure SYST.

The first column indicates how you can use the field in an ABAP program:

- ✔ The system field is set by the runtime environment. You can use its value in an ABAP program, but you must not change it.
- The system field is set by the runtime environment. You can both use and change its value in the ABAP program to affect the runtime behavior of the program.
- The system field must be set from the ABAP program. After that, it can be used by the runtime environment and within your program.
- The system field is for internal use only, and must not be used in ABAP programs.
- The system field is obsolete. No values are assigned to it, and it must not be used in ABAP programs.

“Name” stands for the component name. “Type” and “Length” are the ABAP Dictionary type and length of the field. The “Use” column indicates the contexts in which the system fields can be set, and the “Description” column contains a short description of the field's function.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

	Name	Type	Length	Use	Description
✔	ABCDE	CHAR	26	Constants	Alphabet (A,B,C,...)

🗑	APPLI	RAW	2	Obsolete	
✔	BATCH	CHAR	1	Background processing	Is program running in the background?
🗑	BATZD	CHAR	1	Obsolete	
🗑	BATZM	CHAR	1	Obsolete	
🗑	BATZO	CHAR	1	Obsolete	
🗑	BATZS	CHAR	1	Obsolete	
🗑	BATZW	CHAR	1	Obsolete	
✔	BINPT	CHAR	1	Batch input	Is program running in the background?
🗑	BREP4	CHAR	4	Obsolete	
🗑	BSPLD	CHAR	1	Obsolete	
✔	CALLD	CHAR	1	ABAP program	Call mode of the ABAP program
✔	CALLR	CHAR	8	Printing lists	ID for print dialog function
🗑	CCURS	DEC	9	Obsolete	
🗑	CCURT	DEC	9	Obsolete	
🗑	CDATE	DATS	8	Obsolete	
🔒	CFWAE	CUKY	5	Internal	
🔒	CHWAE	CUKY	5	Internal	
✔	COLNO	INT4	10	Creating lists	Current list column
✔	CPAGE	INT4	10	Processing lists	Current page number
✔	CPROG	CHAR	40	ABAP program	Program that called the current external procedure
🗑	CTABL	CHAR	4	Obsolete	
🗑	CTYPE	CHAR	1	Obsolete	
✔	CUCOL	INT4	10	Screens	Horizontal cursor position in PAI
✔	CUROW	INT4	10	Screens	Vertical cursor position in PAI
✔	DATAR	CHAR	1	Screens	Displays user input
✔	DATLO	DATS	8	Date and time	User’s local date
✔	DATUM	DATS	8	Date and time	Current application server date
✔	DAYST	CHAR	1	Date and time	Flag for summer (daylight saving) time
✔	DBCNT	INT4	10	Database access	Number of database rows processed
✔	DBNAM	CHAR	20	ABAP program	Logical database linked to the program

System Fields in Alphabetical Order

✔	DBSYS	CHAR	10	R/3 System	Name of the central database system
🗑	DCSYS	CHAR	4	Obsolete	
🔒	DEBUG	CHAR	1	Internal	
🔒	DSNAM	CHAR	8	Internal	
✔	DYNGR	CHAR	4	ABAP program	Screen group of the current screen
✔	DYNNR	CHAR	4	ABAP program	Number of the current screen
🔒	ENTRY	CHAR	72	Internal	
✔	FDAYW	INT1	3	Date and time	Day in the factory calendar
✔	FDPOS	INT4	10	Strings	Offset in a string
🔒	FFILE	CHAR	8	Internal	
🔒	FLENG	INT4	10	Internal	
🗑	FMKEY	CHAR	3	Obsolete	
🔒	FODEC	INT4	10	Internal	
🔒	FOLEN	INT4	10	Internal	
🔒	FTYPE	CHAR	1	Internal	
🔒	GROUP	CHAR	1	Internal	
✔	HOST	CHAR	8	R/3 System	Name of application server
✔	INDEX	INT4	10	Loops	Current loop pass
🔒	INPUT	CHAR	1	Internal	
✔	LANGU	LANG	1	R/3 System	User’s logon language
✔	LDBPG	CHAR	40	ABAP program	Logical database program
✔	LILLI	INT4	10	Processing lists	List line selected
✔	LINCT	INT4	10	Creating lists	Page length in a list
✔	LINNO	INT4	10	Creating lists	Current line
✔	LINSZ	INT4	10	Creating lists	Line width in a list
✔	LISEL	CHAR	255	Processing lists	Contents of the chosen line
✔	LISTI	INT4	10	Processing lists	Index of the chosen list
🗑	LOCDB	CHAR	1	Obsolete	
🗑	LOCOP	CHAR	1	Obsolete	
✔	LOOPC	INT4	10	Screens	Number of lines visible in the table

	LPASS	CHAR	4	Internal	
	LSIND	INT4	10	Processing lists	Index of the detail list
	LSTAT	CHAR	16	Processing lists	ID for list levels
	MACDB	CHAR	4	Obsolete	
✔	MACOL	INT4	10	Printing lists	Columns from the SET MARGIN statement
✔	MANDT	CLNT	3	R/3 System	Current client
	MARKY	CHAR	1	Obsolete	
✔	MAROW	INT4	10	Printing lists	Rows from the SET MARGIN statement
✔	MODNO	CHAR	1	R/3 System	Index of the external sessions
✔	MSGID	CHAR	20	Messages	Message class
✔	MSGLI	CHAR	60	Messages	Message line
✔	MSGNO	NUMC	3	Messages	Message number
✔	MSGTY	CHAR	1	Messages	Message type
✔	MSGV1	CHAR	50	Messages	Message variable
✔	MSGV2	CHAR	50	Messages	Message variable
✔	MSGV3	CHAR	50	Messages	Message variable
✔	MSGV4	CHAR	50	Messages	Message variable
	NEWPA	CHAR	1	Internal	
	NRPAG	CHAR	1	Internal	
	ONCOM	CHAR	1	Internal	
✔	OPSYS	CHAR	10	R/3 System	Operating system of application server
✔	PAART	CHAR	16	Printing lists	Print formatting
	PAGCT	INT4	10	Obsolete	
✔	PAGNO	INT4	10	Creating lists	Current page
	PAUTH	NUMC	2	Internal	
✔	PDEST	CHAR	4	Printing lists	Output device
✔	PEXPI	NUMC	1	Printing lists	Spool retention period
✔	PFKEY	CHAR	20	Screens	Current GUI status
	PLAYO	CHAR	5	Internal	
	PLAYP	CHAR	1	Internal	

System Fields in Alphabetical Order

✔	PLIST	CHAR	12	Printing lists	Name of spool request
🔒	PNWPA	CHAR	1	Internal	
✔	PRABT	CHAR	12	Printing lists	Cover sheet: Department
✔	PRBIG	CHAR	1	Printing lists	Selection cover sheet
✔	PRCOP	NUMC	3	Printing lists	Number of copies
✔	PRDSN	CHAR	6	Printing lists	Name of the spool dataset
🗑	PREFX	CHAR	3	Obsolete	
🔒	PRI40	CHAR	1	Internal	
✔	PRIMM	CHAR	1	Printing lists	Output immediately
🔒	PRINI	NUMC	1	Internal	
🔒	PRLOG	CHAR	1	Internal	
✔	PRNEW	CHAR	1	Printing lists	New spool request
✔	PRREC	CHAR	12	Printing lists	Recipient
✔	PRREL	CHAR	1	Printing lists	Delete after output
✔	PRTXT	CHAR	68	Printing lists	Text for cover sheet
🔒	REPI2	CHAR	40	Internal	
✔	REPID	CHAR	40	ABAP program	Current main program
🔒	RSTRT	CHAR	1	Internal	
✔	RTITL	CHAR	70	Printing lists	Program from which you are printing
✔	SAPRL	CHAR	4	R/3 System	R/3 Release in use
✔	SCOLS	INT4	10	Screens	Number of columns
🗑	SFNAM	CHAR	30	Obsolete	
🔒	SFOFF	INT4	10	Internal	
✔	SLSET	CHAR	14	Selection screens	Variant name
✔	SPONO	NUMC	10	Printing lists	Spool number
🗑	SPONR	NUMC	10	Obsolete	
✔	SROWS	INT4	10	Screens	Number of lines
✔	STACO	INT4	10	List processing	First column displayed
✔	STARO	INT4	10	List processing	Topmost line displayed
✔	STEPL	INT4	10	Screens	Index of current table line

	SUBCS	CHAR	1	Internal	
✔	SUBRC	INT4	10	Return code	Return code following an ABAP statement
	SUBTY	RAW	1	Internal	
✔	SYSID	CHAR	8	R/3 System	Name of the R/3 System
	TABID	CHAR	8	Internal	
✔	TABIX	INT4	10	Internal tables	Current line index
✔	TCODE	CHAR	20	ABAP program	Current transaction code
	TFDSN	CHAR	8	Obsolete	
✔	TFILL	INT4	10	Internal tables	Current number of lines
✔	TIMLO	TIMS	6	Date and time	User's local time
✔	TITLE	CHAR	70	Screens	Text in the title bar
✔	TLENG	INT4	10	Internal tables	Line size
	TLOPC	INT4	10	Internal	
	TMAXL	INT4	10	Obsolete	
	TNAME	CHAR	30	Obsolete	
✔	TOCCU	INT4	10	Internal tables	Initial memory requirement
	TPAGI	INT4	10	Obsolete	
	TSTIS	INT4	10	Internal	
	TTABC	INT4	10	Obsolete	
	TTABI	INT4	10	Obsolete	
	TVAR0	CHAR	20	Creating lists	Text variable for titles
	TVAR1	CHAR	20	Creating lists	Text variable for titles
	TVAR2	CHAR	20	Creating lists	Text variable for titles
	TVAR3	CHAR	20	Creating lists	Text variable for titles
	TVAR4	CHAR	20	Creating lists	Text variable for titles
	TVAR5	CHAR	20	Creating lists	Text variable for titles
	TVAR6	CHAR	20	Creating lists	Text variable for titles
	TVAR7	CHAR	20	Creating lists	Text variable for titles
	TVAR8	CHAR	20	Creating lists	Text variable for titles
	TVAR9	CHAR	20	Creating lists	Text variable for titles

✔	TZONE	INT4	10	Date and time	Difference between local time and GMT
✔	UCOMM	CHAR	70	Screens	Function code that triggered PAI
✔	ULINE	CHAR	255	Constants	Horizontal line with length 255
✔	UNAME	CHAR	12	R/3 System	Username of current user
✔	UZEIT	TIMS	6	Date and time	Current application server time
✔	VLINE	CHAR	1	Constants	Vertical line
🗑	WAERS	CUKY	5	Obsolete	
🗑	WILLI	INT4	10	Obsolete	
🗑	WINCO	INT4	10	Obsolete	
🗑	WINDI	INT4	10	Obsolete	
🗑	WINRO	INT4	10	Obsolete	
🗑	WINSL	CHAR	79	Obsolete	
🗑	WINX1	INT4	10	Obsolete	
🗑	WINX2	INT4	10	Obsolete	
🗑	WINY1	INT4	10	Obsolete	
🗑	WINY2	INT4	10	Obsolete	
✔	WTITL	CHAR	1	Creating lists	Flag for standard page header
🔒	XCODE	CHAR	70	Internal	
🔒	XFORM	CHAR	30	Internal	
🔒	XPROG	CHAR	40	Internal	
✔	ZONLO	CHAR	6	Date and time	User’s time zone

System Fields in Thematic Order

The system fields are grouped thematically below with notes on their use:

System Information

- Information About the Current R/3 System
- Information About the Current Terminal Session
- Information About the Current Date and Time
- Information About the Current ABAP Program

System Information

Information About the Current R/3 System

SY-DBSYS

Central database system (such as INFORMIX or ORACLE)

SY-HOST

Application server (such as HS0333, PAWDF087 …)

SY-OPSYS

Operating system of the application server (such as HP-UX, SINIX)

SY-SAPRL

R/3 Release in use (such as 30D, 46A, …)

SY-SYSID

R/3 System name (such as B20, I47, ...)

Information About the Current Terminal Session

SY-LANGU

One-character language key with the user's logon language (such as D, E, F…)

SY-MANDT

Client in which the user is logged on (such as 000, 400…)
When you use Open SQL to access the database, SY-MANDT is used as the first key field in the WHERE clause.

SY-MODNO

Index of the external sessions. The first session has the index zero. The value is increased by one each time you choose *System* → *Create session* or start a transaction by entering /o<tcode>. If you have deleted sessions, the system fills free numbers before increasing the count further. Sessions started using CALL TRANSACTION … STARTING NEW TASK begin again at 0.

SY-UNAME

Username of the current user, such as KELLERH, BC400-01…

Information About Current Date and Time

The following system fields are always set automatically. The GET TIME statement synchronizes the time on the application server with the time on the database server and writes it to the field SY-UZEIT. SY-DATUM and the system fields for the local timezone (SY-TIMLO, SY-DATLO, and SY-ZONLO) are also reset.

SY-DATLO

User's local date, for example 19981129, 19990628, …

SY-DATUM

Current application server date, for example 19981130, 19990627, …

SY-DAYST

X during summertime, otherwise space.

SY-FDAYW

Factory calendar day of the week: Monday = 1 … Friday = 5.

SY-TIMLO

User's local time, for example 154353, 225312, …

SY-TZONE

Time difference in seconds between local time and Greenwich Mean Time (UTC), for example, 360, 10800.

SY-UZEIT

Current application server time. for example 164353, 215312, …

SY-ZONLO

User's time zone, for example, EST, UTC, …

Information About the Current ABAP Program

SY-CALLD

X if the program was started using CALL TRANSACTION, CALL DIALOG, or SUBMIT … [AND RETURN]. Space if the program was started using LEAVE TO TRANSACTION or using a transaction code from a screen. SY-CALLD is always space when a batch input session is being processed.

SY-CPROG

The name of the calling program in an external routine, otherwise the name of the current program.

SY-DBNAM

The name of the logical database linked to an executable program.

SY-DYNGR

Screen group to which the current screen belongs. You can assign several screens to one screen group, for example, to allow you to modify them all identically.

SY-DYNNR

Number of the current screen. During selection screen processing, SY-DYNNR contains the screen number of the current selection screen. During list processing, it contains the number of the container screen. During subscreen processing, SY-DYNNR contains the number of the subscreen. This also applies to tabstrip controls.

SY-LDBPG

In executable programs, the database program of the associated logical database.

SY-REPID

Name of the current ABAP program. For externally-called procedures, it is the name of the main program of the procedure. If you pass SY-REPID as an actual parameter to an external procedure, the formal parameter does not contain the name of the caller, but that of the main program of the procedure. To avoid this, assign SY-REPID to an auxiliary variable and use that in the call, or use the system field SY-CPROG.

SY-TCODE

The current transaction code.

Background Processing

SY-BATCH

X if the ABAP program is running in the background, otherwise space

Batch Input

SY-BINPT

X while a batch input session is running and when an ABAP program is called using CALL TRANSACTION USING, otherwise space.

- OPTIONS FROM in the CALL TRANSACTION USING statement can set SY-BINPT to space either for the entire duration of the program, or at the end of the BDC data.
- SY-BINPT is always space during a CATT procedure.

ABAP Programming

Constants

SY-ABCDE

Contains the alphabet. You can use this field with offset to retrieve a letter of the alphabet regardless of codepage.

SY-ULINE

Contains a horizontal line with length 255 that you can use when creating lists.

SY-VLINE

Contains a vertical line (|) that you can use when creating lists.

Loops

SY-INDEX

In a DO or WHILE loop, SY-INDEX contains the number of loop passes including the current pass.

Strings

SY-FDPOS

Location of hit in string operations.

- When you use CO,CN, CA, NA, CS, NS, CP, and NP, offset values are assigned to SY-FDPOS depending on the search result.
- SEARCH … FOR … sets SY-FDPOS to the offset of the search string.

Internal Tables

SY-TABIX

Current line of an internal table. SY-TABIX is set by the statements below, but only for index tables. The field is either not set or is set to 0 for hashed tables.

- APPEND sets SY-TABIX to the index of the last line of the table, that is, it contains the overall number of entries in the table.

- COLLECT sets SY-TABIX to the index of the existing or inserted line in the table. If the table has the type HASHED TABLE, SY-TABIX is set to 0.
- LOOP AT sets SY-TABIX to the index of the current line at the beginning of each loop lass. At the end of the loop, SY-TABIX is reset to the value that it had before entering the loop. It is set to 0 if the table has the type HASHED TABLE.
- READ TABLE sets SY-TABIX to the index of the table line read. If you use a binary search, and the system does not find a line, SY-TABIX contains the total number of lines, or one more than the total number of lines. SY-INDEX is undefined if a linear search fails to return an entry.
- SEARCH <itab> FOR sets SY-TABIX to the index of the table line in which the search string is found.

SY-TFILL

After the statements DESCRIBE TABLE, LOOP AT, and READ TABLE, SY-TFILL contains the number of lines in the relevant internal table.

SY-TLENG

After the statements DESCRIBE TABLE, LOOP AT, and READ TABLE, SY-TLENG contains the length of the lines in the relevant internal table.

SY-TOCCU

After the statements DESCRIBE TABLE, LOOP AT, and READ TABLE, SY-TLENG contains the initial amount of memory allocated to the relevant internal table.

Database Access

SY-DBCNT

SQL statements set SY-DBCNT to the number of table entries processed. In an Open SQL SELECT loop, SY-DBCNT is not set until after the ENDSELECT statement. In Native SQL, SY-DBCNT is not set until after the ENDEXEC statement.

- DELETE sets SY-DBCNT to the number of deleted lines.
- FETCH sets SY-DBCNT to the number of lines read by the corresponding cursor.
- INSERT sets SY-DBCNT to the number of lines inserted.
- MODIFY sets SY-DBCNT to the number of lines processed.
- UPDATE sets SY-DBCNT to the number of lines changed.

Return Code

SY-SUBRC

Return code, set by the following ABAP statements. As a rule, if SY-SUBRC = 0, the statement was executed successfully.

- ASSIGN sets SY-SUBRC to 0 if the field symbol assignment was possible, otherwise to 4.
- AUTHORITY-CHECK sets SY-SUBRC to 0 if the user has the required authorization, otherwise to 4, 8, 12, 16, 24, 28, 32, or 36 depending on the cause of the authorization failure.

System Fields in Alphabetical Order

- CALL DIALOG with USING sets SY-SUBRC to 0 if the processing is successful, otherwise to a value other than 0.
- CALL FUNCTION sets SY-SUBRC in accordance with the defined exception handling.
- CALL METHOD sets SY-SUBRC in accordance with the defined exception handling.
- CALL SELECTION-SCREEN sets SY-SUBRC to 0 if the user chooses *Enter* or *Execute*, and 4 if the user chooses *Cancel*.
- CALL TRANSACTION with USING sets SY-SUBRC to 0 if the processing is successful, otherwise to a value other than 0.
- CATCH SYSTEM-EXCEPTIONS sets SY-SUBRC after the ENDCATCH statement if a system exception occurs. The value is set in the program.
- COMMIT WORK sets SY-SUBRC to 0.
- COMMIT WORK AND WAIT sets SY-SUBRC to 0 if the update is successful, otherwise to a value other than 0.
- COMMUNICATION INIT DESTINATION … RETURNCODE sets SY-SUBRC as specified.
- CONCATENATE sets SY-SUBRC to 0 if the result fits into the target variable, otherwise to 4.
- CREATE OBJECT sets SY-SUBRC if the exceptions of the instance constructor are handled in the program.
- CREATE OBJECT in OLE2 sets SY-SUBRC to 0 if an external object could be created, otherwise to 1, 2, or 3, depending on the cause.
- DELETE sets SY-SUBRC to 0 if the operation is successful, otherwise to 4 or another value other than 0, depending on the cause.
- DEMAND … MESSAGES INTO sets SY-SUBRC to 0 if the message table is empty, otherwise to a value other than 0.
- DESCRIBE LIST sets SY-SUBRC to 0 if the line or list exists, otherwise to 4 or 8.
- EXEC SQL - ENDEXEC sets SY-SUBRC to 0 in nearly all cases. It does, however, set SY-SUBRC to 4 if no entry is read in a FETCH statement.
- FETCH sets SY-SUBRC to 0 if at least one line was read, otherwise to 4.
- GENERATE SUBROUTINE POOL sets SY-SUBRC to 0 if the generation was successful, otherwise to 8.
- GET CURSOR sets SY-SUBRC to 0 if the cursor is correctly positioned, otherwise to 4.
- GET PARAMETER sets SY-SUBRC to 0 if a corresponding value exists in SAP memory, otherwise to 4.
- IMPORT sets SY-SUBRC to 0 if the import is successful, otherwise to 4.
- INSERT sets SY-SUBRC to 0 if the operation is successful, otherwise to 4.
- LOAD REPORT sets SY-SUBRC to 0 if the operation is successful, otherwise to 4 or 8 depending on the cause of the error.
- LOOP sets SY-SUBRC to 0 if there is at least one pass through the extract. Otherwise, it is set to a value other than 0.

- LOOP AT sets SY-SUBRC to 0 if there is at least one loop pass through the internal table, otherwise to 4.
- MODIFY sets SY-SUBRC to 0 if the operation is successful, otherwise to 4.
- MODIFY LINE sets SY-SUBRC to 0 if a line in the list was changed, otherwise it sets it to a value other than 0.
- MODIFY sets SY-SUBRC to 0 if the operation is successful, otherwise to 4.
- OLE2 Automation: Bundled commands set SY-SUBRC to 0 if all commands could be executed successfully, otherwise 1, 2, 3, or 4, depending on the cause of the error.
- OPEN DATASET sets SY-SUBRC to 0 if the file could be opened, otherwise to 8.
- Open SQL statements set SY-SUBRC to 0 if the operation is successful, otherwise to a value other than 0.
- OVERLAY sets SY-SUBRC to 0 if at least one character is overlaid, otherwise to 4.
- READ DATASET sets SY-SUBRC to 0 if the read operation was successful, otherwise to 4 or 8, depending on the cause of the error.
- READ LINE sets SY-SUBRC to 0 if a list line exists, otherwise to a value other than 0.
- READ TABLE sets SY-SUBRC to 0 if table lines are found, otherwise to 2, 4, or 8, depending on the context and cause of the error.
- REPLACE sets SY-SUBRC to 0 if the search string was replaced, otherwise to a value other than 0.
- SCROLL sets SY-SUBRC to 0 if the scrolling within the list was successful, otherwise to 4 or 8, depending on the cause.
- SEARCH sets SY-SUBRC to 0 if the search string was found, otherwise to 4.
- SELECT sets SY-SUBRC to 0 if at least one line was read, otherwise to 4, or possibly 8 in SELECT SINGLE FOR UPDATE.
- SET COUNTRY sets SY-SUBRC if the country code exists in table T005X, otherwise to 4.
- SET BIT sets SY-SUBRC to 0 if the bit could be set, otherwise to a value other than 0.
- SET TITLEBAR sets SY-SUBRC to 0 if the title exists, otherwise to 4.
- SHIFT … UP TO sets SY-SUBRC to 0 if the position could be found within the string, otherwise to 4.
- SPLIT sets SY-SUBRC to 0 if the sizes of the target fields are adequate, otherwise to 4.
- UPDATE sets SY-SUBRC to 0 if the operation is successful, otherwise to 4.
- WRITE … TO sets SY-SUBRC to 0 if the assignment is successful, otherwise to 4.

Screens

Screens

Screens

A group of system fields is set in the PAI event of each screen. With the exception of SY-DATAR, SY-LOOPC, and SY-STEPL, you can also use all of them in interactive list processing

SY-CUCOL

Horizontal cursor position. Counter begins at column 2.

SY-CUROW

Vertical cursor position. Counter begins at line 1.

SY-DATAR

X if at least one input field on the screen was changed by the user or other data transport, otherwise space.

SY-LOOPC

Number of lines currently displayed in a table control.

SY-PFKEY

GUI status of the current screen. This can be set in the PBO event using the SET PF-STATUS statement.

SY-SCOLS

Number of columns on the current screen.

SY-SROWS

Number of rows on the current screen.

SY-STEPL

Index of the current line in a table control. This is set in each loop pass. SY-STEPL does not have a meaningful value outside the loop (for example, during the POV event for a table line).

SY-TITLE

Text that appears in the title bar of the screen. This is the program name for selection screens and lists, and SAP R/3 otherwise. Can be set using the SET TITLEBAR statement.

SY-UCOMM

Function code that triggered the PAI event. There is a unique function code assigned to every function that triggers the PAI event with one exception: *ENTER* triggers the PAI, and various function codes can be passed to SY-UCOMM:

- If the user has entered a command in the command field, this is passed to SY-UCOMM.
- If there is no entry in the command field and a function code has been assigned to the ENTER key, this function code is passed to SY-UCOMM.

- If there is no entry in the command field and there is no function key assigned to the ENTER key, **the content of SY-UCOMM remains unchanged**.

Selection Screens

SY-SLSET

Variant used to fill the selection screen.

Creating Lists

When you create a list, you can use the following system fields to control navigation. They help you to ensure that output statements do not overwrite existing output, or that data is not written outside the list. The system fields SY-COLNO and SY-LINNO always contain the current output position, and they are reset by each output statement. The other system fields contain other values used for creating lists.

SY-COLNO

Current column during list creation. The counter begins at 1. The field is set by the following output statements:

- WRITE, ULINE, and SKIP set SY-COLNO to the next output position.
- BACK sets SY-COLNO to 1.
- POSITION <col> sets SY-COLNO to <col>. If <col> is beyond the page border, any subsequent output statements are ignored.
- NEW-LINE sets SY-COLNO to 1.
- NEW-PAGE sets SY-COLNO to 1.

SY-LINCT

Page length of the list. For a default list of indefinite length, SY-LINCT is 0. If the page length is defined, SY-LINCT has that value.

- LINE-COUNT in the REPORT, PROGRAM, or FUNCTION POOL statement sets SY-LINCT for the current program.
- LINE-COUNT in the SUBMIT statement sets SY-LINCT for the program called in the statement.

SY-LINNO

Current line during list creation. The counter begins at 1, and includes the page header. SY-LINNO is set by the following output statements:

- WRITE, ULINE, and SKIP increase SY-LINNO by one at each line break.
- BACK sets SY-LINNO to the first line following the page header. If you use BACK with RESERVE, SY-LINNO is set to the first line of a block.
- NEW-LINE increases SY-LINNO by one.
- SKIP TO LINE <lin> sets SY-LINNO to <lin>. If <lin> is not between 1 and the length of the page, the statement is ignored.

System Fields in Alphabetical Order

SY-LINSZ

Line width in the list. The default value is the default window width. SY-LINSZ = SY-SCOLS, for SY-SCOLS >= 84, and 84 for SY-SCOLS < 84

You can change the line width of the list using the LINE-SIZE addition in the REPORT or NEW-PAGE statement.

- LINE-SIZE in the REPORT, PROGRAM, or FUNCTION POOL statement sets SY-LINSZ for the current program.
- LINE-SIZE in the SUBMIT statement sets SY-LINSZ for the program called in the statement.

SY-PAGNO

Current page during list creation.

- WRITE, ULINE, and SKIP increase SY-PAGNO by one at each page break.
- NEW-PAGE increases SY-PAGNO by one, but only if there is at least one output line on both the current page and the intended next page.
- NEW-SECTION in the statement NEW-PAGE PRINT ON sets SY-PAGNO to 1.

SY-TVAR0 ... SY-TVAR9

You can assign values to these variables in your programs. Their contents replace the placeholders in the list and column headers of the program in the TOP-OF-PAGE event.

SY-WTITL

The REPORT, PROGRAM, and FUNCTION-POOL statements set this variable to N if you use the NO STANDARD PAGE HEADING addition. Otherwise, it has the value space. NEW-PAGE does not set SY-WTITL.

List Processing

The following system fields are filled at each interactive list event and at the READ LINE statement:

SY-CPAGE

Page number of the topmost page in the display of the list on which the event was triggered. The counter begins at 1.

SY-LILLI

Line at which the event was triggered. The counter begins at 1 and includes the page header.

SY-LISEL

Contents of the line in which the event was triggered (restricted to the first 255 characters).

SY-LISTI

Index of the list in which the event was triggered.

SY-LSIND

Index of the list currently being created. The basic list has SY-LSIND = 0, detail lists have SY-LSIND > 0. The field is automatically increased by one in each interactive list event. You may change the value of SY-LSIND yourself in the program to enable you to navigate between lists.

Changes to SY-LSIND do not take effect until the end of a list event. It is therefore a good idea to place the relevant statement at the end of the corresponding processing block.

SY-LSTAT

Program-controlled name for list levels. You can assign values to SY-LSTAT during list creation. The value of SY-LSTAT when the list is finished is saved with the list. In an interactive list event, SY-LSTAT is set to the value assigned to it during creation of the list in which the event occurred. **SY-LSTAT is no longer maintained, and you should not use it in your programs.**

SY-STACO

Number of the first displayed column of the list from which the event was triggered. The counter begins at 1.

SY-STARO

Number of the topmost displayed line of the topmost displayed page of the list from which the event was triggered. The counter begins at 1, not including the page header.

Printing Lists

When you print lists, the spool and runtime systems require certain internal information that is set in the following system fields when you start printing.

SY-CALLR

Contains a value indicating where printing was started, for example, NEW-PAGE for program-controlled printing, or RSDBRUNT for printing from a selection screen.

SY-PRDSN

Contains the name of the spool file during printing.

SY-SPONO

Contains the spool number during printing.

SY-MAROW, SY-MACOL

The SET MARGIN statement fills the system fields SY-MAROW and SY-MACOL. These determine the number of lines in the top margin and the number of columns in the left-hand margin respectively.

Print Parameters

The print parameters are passed to the spool system by the runtime environment, using a structure with the ABAP Dictionary type PRI_PARAMS. Before this structure existed, system fields were used instead. When you start printing, some of the fields from PRI_PARAMS are still written into system fields with the same names. However, you should not use these system fields yourself.

Messages

When the MESSAGE statement occurs, the following system fields are set. When the MESSAGE … RAISING statement occurs in a function module or method, these fields are also set in the calling program if it is to handle the exception.

System Fields in Alphabetical Order

SY-MSGID

SY-MSGID contains the message class

SY-MSGNO

SY-MSGNO contains the message number

SY-MSGTY

SY-MSGTY contains the message type

SY-MSGV1,...,SY-MSGV4

SY-MSGV1 to SY-MSGV4 contain the fields used to replace the placeholders in the message.

Special Actions that Fill Message Fields

- When you use an ENQUEUE function module to set a lock, and the FOREIGN_LOCK exception occurs, the field SY-MSGV1 contains the name of the owner of the lock.
- When you use CALL TRANSACTION or CALL DIALOG with the USING addition, a message that occurs during the screen chain is returned in the fields SY-MSGID, SY-MSGTY, SY-MSGNO, and SY-MSGV1 ... SY-MSGV4.
- In Remote Function Call (RFC), error messages are retrieved from the remote system and placed in the system fields SY-MSGID, SY-MSGTY, SY-MSGNO,SY-MSGV1, SY-MSGV2, SY-MSGV3, SY-MSGV4. The fields are also set when a short dump or a message with type X occurs.

Internal System Fields

Internal system fields are exclusively for internal use in the ABAP runtime environment and the system kernel. They must not be overwritten in any circumstances, and should also not be read in ABAP programs.

SY-CFWAE

Undocumented

SY-CHWAE

Undocumented

SY-DEBUG

Undocumented

SY-DSNAM

File name for spool output

SY-ENTRY

Undocumented

SY-FFILE

Flat file (USING/GENERATING DATASET).

SY-FLENG

Length of a field

SY-FODEC

Number of decimal places in a field

SY-FOLEN

Output length of a field

SY-FTYPE

Data type of a field

SY-GROUP

Bundling

SY-INPUT

Undocumented

SY-LPASS

Undocumented

SY-NEWPA

Undocumented

SY-NRPAG

Undocumented

SY-ONCOM

Undocumented

SY-PAUTH

Undocumented

SY-PLAYO

Undocumented

SY-PLAYP

Undocumented

SY-PNWPA

Undocumented

SY-PRI40

Undocumented

System Fields in Alphabetical Order

SY-PRINI

Undocumented

SY-PRLOG

Undocumented

SY-REPI2

Undocumented

SY-RSTRT

Undocumented

SY-SFOFF

Undocumented

SY-SUBCS

Call status of an executable program

SY-SUBTY

Call type of an executable program

SY-TABID

Undocumented

SY-TLOPC

Undocumented

SY-TSTIS

Undocumented

SY-XCODE

Extended function code, filled by user actions on a list (like SY-UCOMM). Before the length of SY-UCOMM was extended from 4 to 70 characters, SY-XCODE was used internally to accommodate long entries in the command field. You should always use SY-UCOMM in application programs.

SY-XFORM

SYSTEM-EXIT subroutine.

SY-XPROG

SYSTEM-EXIT program.

Obsolete System Fields

Some of the system fields in the R/3 System were originally adopted from R/2, but are no longer filled with values. They are obsolete, and must not (indeed cannot) be used.

SY-APPLI

R/2 - ID for the SAP applications installed. Not filled in R/3.

SY-BATZD

R/2 - flag for daily background scheduling. Not filled in R/3.

SY-BATZM

R/2 - flag for monthly background scheduling. Not filled in R/3.

SY-BATZO

R/2 - flag for one-time background scheduling. Not filled in R/3.

SY-BATZS

R/2 - flag for immediate background scheduling. Not filled in R/3.

SY-BATZW

R/2 - flag for weekly background scheduling. Not filled in R/3.

SY-BREP4

R/2 - root name of the report requesting background processing. Not filled in R/3.

SY-BSPLD

R/2 - flag for spool output from background processing. Not filled in R/3.

SY-CCURS

R/2 - exchange rate and result field for CURRENCY CONVERSION. Not filled in R/3.

SY-CCURT

R/2 - table exchange rate for CURRENCY CONVERSION. Not filled in R/3.

SY-CDATE

R/2 - exchange rate date for CURRENCY CONVERSION. Not filled in R/3.

SY-CTABL

R/2 - exchange rate table for CURRENCY CONVERSION. Not filled in R/3.

SY-CTYPE

R/2 - exchange rate type for CURRENCY CONVERSION. Not filled in R/3.

SY-DCSYS

Dialog system of the R/2 System. Not filled in R/3.

System Fields in Alphabetical Order

SY-FMKEY

Formerly the current function code menu. Not filled in R/3.

SY-LOCDB

Local database. Not implemented.

SY-LOCOP

Local database operation. Not implemented.

SY-MACDB

Formerly the file name for matchcode access. Not filled in R/3.

SY-MARKY

Current line letter for the MARK statement. The MARK statement will not be supported for much longer.

SY-TMAXL

Formerly the maximum number of entries in an internal table. Not filled in R/3.

SY-TFDSN

Formerly the name of an external storage file for extracts. Not filled in R/3.

SY-PAGCT

R/2 - the maximum number of pages per list. Not filled in R/3.

SY-PREFX

ABAP prefix for background jobs. Not filled in R/3.

SY-SFNAM

Undocumented

SY-SPONR

In R/2, you could process spool files with the TRANSFER statement, which also set SY-SPONR. Not filled in R/3.

SY-TNAME

Formerly the name of an internal table following access. Not filled in R/3.

SY-TTABC

Formerly the index of the last line of an internal table to be read. Not filled in R/3.

SY-TTABI

Formerly the offset of internal tables in the roll area. Not filled in R/3.

SY-TPAGI

Formerly flagged whether an internal table had been moved to the paging area. Not filled in R/3.

SY-WAERS

Formerly the company code currency after reading a posting segment. Not filled in R/3.

SY-WILLI

R/2 - the number of the list line selected from a detail list. Use SY-LILLI instead.

SY-WINCO

R/2 - cursor position on a detail list. Use SY-CUCOL instead.

SY-WINDI

R/2 - index of the list for a detail list. Use SY-LSIND instead.

SY-WINRO

R/2 - cursor position for a detail list. Use SY-CUROW instead.

SY-WINSL

R/2 - contents of the selected line for detail list in a window. Use SY-LISEL instead.

SY-WINX1

R/2 - window coordinates for a detail list in a window. No corresponding field in R/3.

SY-WINX2

R/2 - window coordinates for a detail list in a window. No corresponding field in R/3.

SY-WINY1

R/2 - window coordinates for a detail list in a window. No corresponding field in R/3.

SY-WINY2

R/2 - window coordinates for a detail list in a window. No corresponding field in R/3.

ABAP Glossary

A

ABAP

Advanced Business Application Language. The SAP programming language for **application programming**.

ABAP Dictionary

Central information repository for application and system data. The ABAP Dictionary contains data definitions (**metadata**) that allow you to describe all of the data structures in the system (like **tables**, **views**, and **data types**) in one place. This eliminates redundancy.

ABAP Editor

Tool in the ABAP Workbench in which you enter the source code of ABAP programs and check their syntax. You can also navigate from the ABAP Editor to the other tools in the ABAP Workbench.

ABAP Memory

ABAP Memory is a memory area in the internal session (roll area) of an ABAP program. Data within this area is retained within a sequence of program calls, allowing you to pass data between programs that call one another. It is also possible to pass data between **sessions** using **SAP Memory**.

ABAP Objects

The ABAP runtime environment or the object-oriented extension of ABAP. ABAP Objects uses **classes** to create **objects**.

ABAP Processor

The ABAP processor executes the **processing logic** of the application program, and communicates with the database interface.

ABAP Query

Allows you to create simple **reports** without having to know any ABAP.

ABAP Workbench

The ABAP Workbench is a fully-fledged **development environment** for applications in the ABAP language. With it, you can create, edit, test, and organize application developments. It contains a range of tools including the **Screen Painter**, **ABAP Editor**, and **Repository Browser**.

Aggregated Data Types

Data types built up of a series of **single field types**. An aggregated data type can be a **structure** or an **internal table**. With aggregated types, you can either access the whole object or individual fields.

Application Logic

Generic term for the parts of application programs that process application-relevant data.

Actual Parameter

When you pass parameters to a procedure, the actual parameters are the parameters that you pass from the program. They are passed to the **formal parameters** within the routine.

Application Server

The level within the client/server architecture in which application logic runs.

Attribute

A component of a **class** in **ABAP Objects**. Attributes contain the data of a class. They are used by the **methods** of classes.

B

Background Processing

Program processing without user dialogs.

Bottom-Up

Development method that starts with the lowest components in a hierarchy and becomes more generalized as the development project continues. The opposite of **top-down**.

Business API (BAPI)

Standard interface in the R/3 System that allows the system to communicate with components of other business software suites.

C

Casting

A reference assignment in **ABAP Objects**. It is checked for correctness at runtime, not in the syntax check.

Class

Template for **objects** in **ABAP Objects**. You can define classes either locally in an **ABAP program**, or globally using the **Class Builder** in the **ABAP Workbench**. You can either define a new class from first principles, or derive one using **inheritance**. Classes can implement **interfaces**.

Class Builder

ABAP Workbench tool used to create **classes** and **interfaces**.

Class Reference

Data type of a **reference variable** or **object reference** that allows you to access all of the visible components of an **class**.

Client

A unit within an R/3 System that is complete in a legal and organizational sense, and which has its own user masters and own **tables**.

Client Handling

Users working in a particular client are only allowed to use certain transactions, as specified in the administration data for that client. The R/3 System automatically enforces this restriction. When you program database accesses, it is possible to bypass automatic client handling.

Client/Server Architecture

An architecture in which data and applications are distributed over different hosts in order to make the most efficient use of each host's resources.

Cluster

An object-oriented collection of **tables**.

Comment

Explanatory notes in the source code of a program that make it easier to understand. This makes subsequent maintenance and support easier. You can make a whole line into a comment by placing a * at the beginning of the line. Alternatively, if you place a " anywhere in a line, the rest of that line becomes a comment.

Company Code

A legally-independent unit within a client that has its own fiscal year end.

Constant

A **data object** declared statically with a declarative statement. They allow you to store data under a particular name within the memory area of a program. The value of a constant must be defined when you declare it. It cannot subsequently be changed.

Container

A file containing several programming units that all have the same type. For example, you can have a **container** for **subroutines**.

Contexts

A technique for avoiding repeated database access or calculations with data in a program. You create contexts using the Context Builder in the **ABAP Workbench.** They contain key fields, definitions of the relationships between the fields, and other fields that can be derived or calculated using the key field values.

Control Levels

A series of lines in an **internal table** or **extract dataset** that forms a group based on the contents of one or more of its **fields**.

Control Level Processing

Used in an **internal table** or **extract dataset** to form groups of entries.

D

Data Control Language (DCL)

Statements for authorization and consistency checks. Not used in the R/3 System, since the system is itself responsible for checking data.

Data Definition Language (DDL)

Language for defining the attributes of a database management system and creating and administering **database tables**. It is not contained in **Open SQL**.

Data Dictionary

See **ABAP Dictionary**.

Data Manipulation Language (DML)

Statements for reading and changing data in **database tables**.

Database

Set of data that is part of a database system and is managed by the database management system.

Database Commit

A COMMIT WORK in a relational database system.

Database Logical Unit of Work (LUW)

A logical unit of work is a set of database operations. They belong together, and are either all executed (commit) or all canceled (rollback).

Database Rollback

A ROLLBACK WORK in a relational database system.

Database Cursor

A mechanism for passing data from the database to an ABAP program. An open cursor is linked to a multiple-line selection in the database table for which it was opened. You can place the lines of the selection one by one into a flat target object and process them.

Database Interface

The part of a **work process** that links it to the database. It converts **Open SQL** into **Standard SQL**, and allows the application server to communicate with the **database**.

Database Server

Host on which the database is installed.

Database Table

Most **databases** that are used for business applications are based on the relational database model, in which the real world is represented by **tables**.

Data Element

Describes the business function of a **table field**. Its technical attributes are based on a **domain**, and its business function is described by its field labels and documentation.

Data Object

Name for an instance of a **data type** in **ABAP**. A data object occupies a **field** in memory.

Data Type

Describes the technical attributes of a **data object**. ABAP uses the data type of a **field** to interpret its contents. There are **single field types**, **aggregated types**, and **object types**.

Declaration Part

Part of every program or procedure. It contains data, **selection screen**, and **class** definitions that are visible throughout the program or procedure.

Deep Structure

A **structure** that contains an **internal table** as a component.

Deep Tables

An **internal table** whose line type is a **deep structure**.

Dialog Module

Statement block that describes the different states (**PBO**, **PAI**, user input) of a **screen**. A **module pool** contains a set of dialog modules.

Dialog Program

A program that contains (or consists entirely of) **dialog modules**.

Dispatcher

Link between work processes and users. It receives user interaction from the SAPgui, and directs it to a free work process.

Domain

Specifies the technical attributes of a data element - its data type, length, possible values, and appearance on the screen. Each data element has an underlying domain. A single domain can be the basis for several data elements. Domains are objects in the **ABAP Dictionary**.

Double Byte Character Set

Two byte code system for extensive character sets such as Japanese or Chinese.

E

Elementary Types

There are eight predefined elementary types: Character string (C), numeric string (N), date field (D), time field (T), hexadecimal field (X), integer (I), floating point number (F), and packed number (P). You can use these types as the basis for further types that you create either locally in a **program** or globally in the **ABAP Dictionary**.

Encapsulation

Property of objects in object-oriented programming. Each object has an external interface. The implementation of the object is encapsulated, that is, invisible externally.

Event Block

A series of statements that are processed when a particular event occurs when a program runs or in **selection screen** and **list** processing. Each event block begins with an event keyword, and ends at the introductory keyword of the next event block.

Extract

Sequential dataset in the memory area of a program. An extract dataset consists of a sequence of records of a pre-defined structure. However, the structure need not be identical for all records. In one extract dataset, you can store records of different length and structure one after the other.

F

Field

Area in memory with address and length. **Data objects** in **ABAP** occupy fields. The contents of a field are interpreted according to the **data type** of the relevant **data object**.

Field Symbol

Placeholder or symbolic name for a **field**. Field symbols do not occupy any space, but instead point to a **data object**.

Flat Structure

Structure consisting only of **elementary data types**.

Flow Logic

See **Screen Flow Logic**.

Foreign Key

One or more **fields** in a **table** that occur as **key fields** in another table.

Formal Parameter

Placeholder for passing values to a **procedure**. Formal parameters declare the number and **types** of the actual parameters that will be passed to the **procedure**.

Forward Navigation

Forward navigation allows you to access an object by double-clicking its name in the source code of an ABAP program or in an object list.

Function Builder

ABAP Workbench tool used to create, display, modify, and delete **function modules** and **function groups**.

Function Group

Program with type F that contains **function modules**. Created using the **Function Builder**.

Function Library

Library of existing **function modules** in the **ABAP Workbench**.

Function Module

Procedure that can only be created within a **type F** program, but which can be called from any **ABAP program** within the R/3 System. You create them using the **Function Builder** in the **ABAP Workbench**. A function module has a defined **parameter interface**.

G

Gateway

This is the interface for the R/3 communication protocols (RFC, CPI/C). It can communicate with other **application servers** in the same R/3 System, with other R/3 Systems, with R/2 Systems, or with non-SAP systems.

Generic Attributes

Attributes that are not fully typed. The missing attributes are specified dynamically at runtime.

Global Data

Data that is visible throughout a program (and also in procedures that it calls).

GUI Status

Each screen has a GUI status. The status is a collection of interactive interface elements that allows the user to select functions. A GUI Status is an independent development object within an **ABAP program**. You create and edit them using the **Menu Painter** in the **ABAP Workbench**.

GUI Title

Title of a screen that appears in the title bar of the window. A GUI title is an independent development object within an **ABAP program**. You create and edit them using the **Menu Painter** in the **ABAP Workbench**.

H

Hashed Table

An **internal table** whose entries you can access by specifying the key. The system manages the table using a hash algorithm. The advantage of this is that the search overhead does not increase with the size of the table.

Header Line

Work area for table access. You can place the successive lines of an internal table into the header line and process them.

I

Include Program

A technique for **modularizing** the source code. **Program** with type I. Allows you to reuse code in different programs. The source code of an include program is fully incorporated into the source code of the program in which the include statement occurs.

Index Table

An internal table for which the system maintains a linear index. You can use the index to access entries in the table. There are two kinds of index tables: **Standard** tables and **sorted** tables.

Inheritance

A special way of defining a **class**. You can use an existing class to derive a new class. Derived classes (**subclasses**) inherit the **attributes** and **methods** of the original class (**superclass**). They may also have new methods, or redefine existing ones.

Interface

Part of a **class** definition in **ABAP Objects**. You can define interfaces either locally in an **ABAP program**, or globally using the **Class Builder** in the **ABAP Workbench**. **Classes** can implement interfaces. They then adopt all of the components of the interface. The **class** must implement all of the **methods** of the interface itself.

Interface Reference

Data type of a **reference variable** or **object reference** that allows you to access all of the visible components of an **interface**.

Interface Work Area

A special data object that you use to pass data between **screens** or **logical databases** and **ABAP programs**.

Internal Tables

Data object consisting of a set of lines with the same **data type**. There are different access types for internal tables: Sorted and unsorted **index tables**, and **hashed tables**, and also different line types: **Vectors**, “real” tables, with **flat structures**, and **deep tables**. You should use internal tables whenever you need to use structured data within a program.

J

Join

A technique for linking two or more **tables**. The tables involved must have at least one common column.

K

Kernel

Runtime environment for all R/3 applications. It is independent of hardware, operating system, and database. Has the following functions: Running applications, administration of users and processes, database access, communication with other applications, control and administration of the R/3 System.

Key

Selected **fields** of a **table** used to identify table records. A key may be either unique or non-unique.

L

List

Lists are output-oriented screens which display formatted, structured data. They are defined, formatted, and filled using ABAP commands. Although they are output-oriented, you can include input fields in a list.

Literal

Data object, without a name, that you create in the source code of a program. It is fully defined by its value. You cannot change the value of a literal.

Local Data

Data that is only visible in the current program (including its **subroutines**) or procedure.

Logical Database Builder

Tool in the **ABAP Workbench** for creating and editing **logical databases.**

Logical Database

An **ABAP program** that retrieves data from various **database tables**. The dataset covered by a logical database program is sometimes referred to itself as the logical database. The lines of the relevant tables are passed one by one to the program that is using the logical database. You can use a logical database with any number of executable programs.

M

Macro

A technique for **modularizing** the source code. The DEFINE statement introduces a set of statements that you can then use at any point within a program. Unlike **include programs**, macros can only be used in the program in which you define them.

Menu Painter

A tool in the **ABAP Workbench** that allows you to create menus and assign functions to function keys and pushbuttons.

Metadata

Data that describes other data. Data definitions are metadata. They are stored in the **ABAP Dictionary**.

Method

A **procedure** that is a component of a **class** in **ABAP Objects**. Methods represent the functions of a class. They work with the **attributes**. Methods can only be defined in the implementation parts of classes.

Modularization

Splitting a program into various parts (modules). There are two kinds of modules - those that are included in the source code of the program when it is generated (**macros** (local) or **include programs** (global)), and those that are called as independent modules at runtime. These are known as **procedures**. Modularization makes large programs easier to understand and maintain, and reduces the amount of redundancy.

Module Pool

Type M **program** containing the **dialog modules** of the **screens** in the program.

N

Native SQL

An access method that uses database-specific statements. Native SQL statements are not interpreted, and are passed to the database without any checks. You should not use Native SQL statements, since they are database-specific (your program will no longer be portable), and ABAP does not check to ensure that the statements are correct.

Nested Structure

A **structure** that itself contains a **structure** as a component.

O

Object

A piece of code containing data (**attributes**) and providing services (**methods**). Objects are instances of **classes** in **ABAP Objects**.

Object Reference

Object references are used to access the attributes and methods of an **object** in **ABAP Objects**. Object references are contained in **reference variables**.

Object Types

Object types are used to describe objects. They contain both data and functions. **Classes** and **interfaces** are both object types. They are part of **ABAP Objects**.

Open SQL

A set of statements that allows you to access the database from an ABAP program. Open SQL statements are fully portable to any of the database platforms supported by SAP. Open SQL is a subset of Standard SQL (without the DDL part), but also contains SAP-specific extensions.

P

Parameter

1. A data object in a program whose value can either be entered by the user at runtime or be passed from another program.
2. A value passed when you call a procedure or program from another program. See also **actual parameter** and **formal parameter**.

Parameter Interface

A method by which programs and the **procedures** that they call can exchange data. When you write **subroutines** or **procedures**, you define **formal parameters**, which define the type, number, and order of the **actual parameters** that should be passed.

Polymorphism

An attribute of **objects** in object-oriented programming. Identically-named **methods** behave differently in different **classes**.

Presentation Server

The host that receives input from the user and presents output from the system.

Procedure

A **modularization** technique. Unlike source code modules (macros, include programs), procedures have interfaces for data transfer. ABAP contains the following kinds of procedures:

- **Subroutines**: Local modularization (FORM)
- **Function modules** Global modularization
- **Methods:** Contain the functions of classes and their instances in **ABAP Objects**.

Process After Input (PAI)

Part of the screen **flow logic**. PAI processing determines what happens after the user has chosen a function on the screen.

Process Before Output (PBO)

The part of the **screen flow logic** that determines the processing steps that occur before a screen is displayed.

Processing Block

Any ABAP statement that does not belong to the declaration part, that is, the **event blocks**, **dialog modules**, and **procedures**.

Processing Logic

Processing logic implements the business functions of the R/3 System. It is contained in ABAP programs.

Program Types

- Type 1 **Report**, executable program (input data → process data → display data)
- Type M: **Module pool**, started using a **transaction code**.
- Type F: **Function group**, container for function modules. Function modules may only be programmed within a function group. You create them using the **Function Builder** in the **ABAP Workbench**.
- Type K: **Class** definitions, containers for global classes in **ABAP Objects**. You create them using the **Class Builder** in the **ABAP Workbench**.
- Type J: Interface definitions, containers for global interfaces in ABAP Objects. You create them using the **Class Builder** in the **ABAP Workbench**.
- Type S: Subroutine pools, containers for subroutines. Non-executable.
- Type I: Include programs. Non-executable.

Q

R

R/3 Repository

Part of the database containing all development objects of the ABAP Workbench, such as programs, screens, and **function modules**.

RABAX

A program in the **ABAP runtime environment** that catches runtime errors and triggers a **short dump**.

Reference

See **object reference**.

Reference Variable

Data object that contains an **object reference**. The data type of a reference variables can be a **class reference** or an **interface reference**.

Remote Function Call (RFC)

An interface protocol based on CPI-C that allows programs in different systems to communicate with one another. External applications and tools can use ABAP functions, and the R/3 System can access external systems.

Report

Executable program with a three-stage function: Data input → data processing → data output. Reports read and calculate using data from **database tables**, without actually changing it.

Repository Browser

A tool in the ABAP Workbench that provides a categorized list of development objects for a development class, program, function group, or class. When you double-click an object in the list, it is opened (along with the correct Workbench tool) for display or editing. Transaction SE80.

Routine

See **procedure**.

Runtime Error

Error that occurs while a program is running, often because the error could not be determined statically by the syntax check.

Runtime Environment

Umbrella term for **screen** and **ABAP processors**.

S

SAPgui

The SAP Graphical User Interface is the frontend of the R/3 System. It allows users to run applications and enter and display data.

SAP LUW

A logical unit consisting of dialog steps, whose changes are written to the database in a single **database LUW**.

SAP Memory

This is a memory area to which all sessions within a **SAPgui** have access. You can use SAP memory either to pass data from one program to another within a **session** (as with **ABAP memory**) or to pass data from one session to another.

SAP Transaction

A dialog-driven program, or any program started using a **transaction code**.

Screen

A screen (also referred to sometimes as a dynamic program or “dynpro”) consists of the screen itself and the **flow logic**, which controls how the screen is processed. You create both the screen and its flow logic in the **Screen Painter**.

Screen Flow Logic

The screen flow logic consists of **PBO** and **PAI**, and controls most user interaction. The R/3 Basis system contains a special language for programming screen flow logic.

Screen Painter

Tool in the **ABAP Workbench**, used to create **screens** and their **flow logic**.

Screen Processor

The screen processor executes the screen flow logic. It takes control of communication between the **work process** and the **SAPgui**, calls **processing logic** modules, and passes the contents of the screen fields back to the program for processing.

Screen Table

Table on a screen, created using the step loop or table control technique.

Screen Type

There are three screen types in the R/3 System: **Screens**, **selection screens**, and **lists**.

Selection Screen

Special **screens** used to enter values in **ABAP programs**. Unlike normal screens, they are not defined in the **Screen Painter**, but using ABAP statements in the program.

Selection View

Selection views are a collection of fields from different **database tables**. You can create them in the **Repository Browser** or the **Logical Database Builder** in the **ABAP Workbench**.

Session

The R/3 window in the **SAPgui** represents a session. After logging on, the user can open up to five further sessions (R/3 windows) **within** the single SAPgui. These behave almost like independent SAPguis.

Shared Memory

The memory area shared by all **work processes**.

Short Dump

The text that accompanies a **runtime error**. This should enable you to find and correct the error.

Single Field Type

A **data type** in **ABAP** whose **data objects** occupy a single **field**. **Elementary types** and **references** are single field types.

Sorted Table

An **index table** that is always stored correctly sorted according to its table key.

SPA/GPA Parameters

Values stored in the user-specific **ABAP memory**. You create SPA/GPA parameters using the **Repository Browser** in the **ABAP Workbench**.

SQL Report

An **executable program** that does not use a **logical database.** Instead, it defines its own **selection screen**, and reads its own data from the database using **Open SQL** statements.

Standard SQL

A largely standardized language for accessing relational databases. It consists of three parts:

- **Data Manipulation Language** (DML)
- **Data Definition Language** (DDL)
- **Data Control Language** (DCL)

Standard Table

An unsorted **index table**.

Statement

A line of an **ABAP program**. ABAP contains the following kinds of statements: **declarative statements**, control statements, call statements, operational statements, database statements, and modularization statements.

Structure

A structure is a logically-connected set of fields. It is a **data object** containing a sequence of any **data types**. Structures can be **flat**, **deep**, or **nested**.

Structured Query Language (SQL)

Standard language for database access See also **Native SQL**, **Open SQL**, **Standard SQL**.

Subclass

The resulting **class** when you use **inheritance** to derive a class from a **superclass**.

Subquery

A special **Open SQL** selection statement that you can use under certain conditions to form an extra query.

Subroutine

A **procedure** that you define in a program using the FORM statement, and which you can call any number of times from any ABAP program using the PERFORM statement. When you call the subroutine, you can pass parameters to it. Subroutines are normally used locally, that is, called in the same program in which they are defined.

Superclass

The **class** from which a **subclass** is derived in **inheritance**.

T

Text Pool

A set of texts belonging to a **program**. You address them using **text symbols** in the source code. The text pool can be translated into other languages.

Text Symbol

A **data object** that is generated when you start a program from the **text pool** of the program.

Top-Down

A development method that starts with the highest level of a hierarchy and becomes more specialized as the development project goes on. The opposite of **bottom-up**.

Transaction

A set of work steps that belong together. In the context of database changes, the term stands for a change of state in the database. It is also used as an abbreviation of **SAP transaction**.

Transaction Code

A sequence of letters and digits that starts an **SAP transaction** when you enter it in the command field. Transaction codes are normally used to start dialog-driven programs. However,

you can also use them to start **reports**. You create transaction codes in the **Repository Browser**. They are linked to an **ABAP program** and an initial **screen**.

Transparent Table

You define transparent tables in the ABAP Dictionary. They are then created in the database. You can also use transparent tables independently of the R/3 System. Cluster techniques are not used, since they cannot be read using **Open SQL**.

U

V

Variable

A named **data object** that you can declare statically using declarative statements, or dynamically while a program is running. They allow you to store changeable data under a particular name within the memory area of a program.

Vector

An **internal table** whose line type is an **elementary type**.

View

A virtual table that does not contain any data, but instead provides an application-oriented view of one or more **ABAP Dictionary** tables.

W

Work Process (Dialog)

Consists of a **screen processor**, **ABAP processor**, and database interface. Part of an **application server**, it executes the dialog steps of application programs.

X

Y

Z

Syntax Conventions

The conventions for syntax statements in this documentation are as follows:

Key	Definition
STATEMENT	Keywords and options of statements are uppercase.
<variable>	Variables, or words that stand for values that you fill in, are in angle brackets. Do not include the angle brackets in the value you use (exception: field symbols).
[]	Square brackets indicate that you can use none, one, or more of the enclosed options. Do not include the brackets in your option.
|	A bar between two options indicates that you can use either one or the other of the options.
()	Parentheses are to be typed as part of the command.
,	The comma means you may choose as many of the options shown as you like, separating your choices with commas. The commas are part of the syntax.
<f_1> <f_2>	Variables with indices mean that you can list as many variables as you like. They must be separated with the same symbol as the first two.
.....	Dots mean that you can put anything here that is allowed in the context.

In syntax statements, keywords are in upper case, variables are in angle brackets. You can disregard case when you type keywords in your program. WRITE is the same as Write is the same as write.

Output on the output screen is either shown as a screen shot or in the following format:

Screen `output`.