Adabas D

	Version 13	SQL-PL

software AG

This document applies to Adabas D Version 13 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Table of Contents

SQL-PL 1
SQL-PL 1
Field of Application 2
Field of Application 2
SQL-PL Objects 4
SQL-PL Objects 4
General Properties of SQL-PL Modules 5
DB Procedures 5
Triggers 6
DB Functions 6
Procedures 7
Functions 7
Forms 7
HELP Forms 7
Menus 8
Syntax of the SQL-PL Objects 8
SQL-PL Components 10
SQL-PL Components 10
The Workbench 11
The Workbench 11
The Handling of the Workbench 11
The Function Keys of the Workbench 13
The Workbench for Beginners 14
The Version Administration 15
The Action Bar of the Workbench 16
The ’Object’ Menu Item 17
The ’Selection’ Menu Item 22
The ’Privileges’ Menu Item 27
The ’Test’ Menu Item 29
The ’Tools’ Menu Item 31
The ’SCROLL’ Menu Item 33
The ’INFO’ Menu Item 34
Commands 34
User-specific Set Parameters 43
The SQL-PL Language 50
The SQL-PL Language 50
Basic Elements 50
Comments 51
Names 51
Literals 51
Variables 54
Arithmetic Expressions 56
String Expressions 57
Boolean Expressions 57
Variable Declaration 60
Basic Statements 60
Value Assignments 62
The IF Statement 62

The CASE Statement 62
The REPEAT Statement 63
The WHILE Statement 63
The FOR Statement 64
The SKIP Statement 64
The RETURN Statement 64
The STOP Statement 65
The Statements LTSORT and GTSORT 66
Calling Procedures, Forms, and Functions 66
The Statements CALL, SWITCH, and SWITCHCALL 67
Parameters for CALL, SWITCH, and SWITCHCALL 68
Calling Functions 69
Calling Stored Procedures 70
DB Procedures 70
Triggers 71
DB Functions 71
Embedding SQL 71
Database Accesses 71
Dynamic SQL Statements 73
Dynamic FETCH Statements 75
Mass Fetch 75
Support of the Adabas Data Type LONG 76
Multi-DB Operation 78
SQL Error Handling 79
Procedures for SQL Error Handling 80
Catching Runtime Errors (TRY-CATCH) 81
Query Call 81
Stored Commands 82
Report Formatting 83
Further Facilities 85
Master/Detail REPORT 88
Editor Call 89
Line-oriented Input and Output 91
Processing Files 95
Calling Operating System Commands 97
SQL-PL System Functions 98
Arithmetic Functions 98
String Functions 99
Date and Time Functions 103
SET Function 104
Time Measuring Functions 105
System or $ Variables 106
Module Options 110
The LOOP Option 110
The Option AUTOCOMMIT OFF 111
The Trace Options MODULETRACE and SQLTRACE 111
The TEST DBPROC Option 112
The LIB Option 112
The KEYSWAP Option 113
The SUBPROC Option 113

Stored Procedures 114
Stored Procedures 114
Creating Stored Procedures 115
DB Procedures 115
Triggers 115
DB Functions 117
Parameter Declaration 118
Calling Subprocedures and Functions 118
Calling DB Procedures from Stored Procedures 119
Differences between CALL PROC and CALL DBPROC in Stored Procedures . . . 119
Setting the Error Number 119
Restrictions 120
Calling a DB Procedure 122
Calling a Trigger 122
Calling a DB Function 122
The Debugger 123
The Debugger 123
Procedure 123
Functions of the Debugger 125
Displaying and Modifying Variable Contents 125
Breakpoints 126
Single-step Mode 127
Continuing the Execution 127
Displaying the Call Sequence 127
Displaying System Variables 127
Displaying Parameters 128
Interrupting the Program 128
Displaying Help Information 128
Forms 129
Forms 129
General Points on Forms and Menus 129
The Form Layout 132
Form Fields and Messages (MESSAGE, ERROR) 133
Form Fields of Variable Lengths (>>) 134
Multi-line Form Fields (") 134
Vector Components with Constant Index 135
Vector Components with Dynamic Index (FIELD/OFFSET) 136
Field Numbers Instead of Variable Names 137
Definition of LAYOUT Control Characters 138
Form Processing Statements 141
The FIELD Statement 143
Dividing into Field Groups (GROUP) 154
BEFORE/AFTER GROUP 155
Ignoring the Input Check (IGNORE) 155
Initializing the Cursor Position (MARK) 156
Key Activation (ACCEPT) 156
The KEYSWAP Statement 157
Leaving the Form (RETURNONLAST) 158
Scrolling Support (AUTOPAGE, PAGE) 159
Header Lines and Bottom Lines (HEADERLINES, BOTTOMLINES) 160
Situation-dependent Display Attributes (SPECIALATTR) 161

Displaying the Input Mode (INSERTMODE) . . . 162
Controlling the Dialog Sequence (CONTROL, CASE) . . . 162
The PAGE Statement . . . 165
PICK/PUT Mechanism in Forms (PICK/PUT/AUTOPUT) . . . 165
The NEXTFIELD Statement . . . 168
The NEXTGROUP Statement . . . 169
The SCROLLFIELD Statement . . . 169
The KEYS Statement . . . 171
Options for Form Calls . . . 171
Suppressing the INIT Phase (NOINIT) . . . 172
Cursor Control (MARK, $CURSOR) . . . 173
Overriding Keys (ACCEPT) . . . 174
Overriding Display Attributes (ATTR) . . . 178
The Window Options SCREENPOS, SCREENSIZE, and CLEAR . . . 178
Form Segments (FORMPOS) . . . 180
Automatic Framing by FRAME . . . 181
Superimposing Forms (BACKGROUND) . . . 182
Restoring the Form Background (RESTORE) . . . 182
Form Output via Printer (PRINT) . . . 182
Overriding the Active Input Fields (INPUT/NOINPUT) . . . 184
Activating the Action Bar (ACTION) . . . 185
HELP Forms as Pick Lists . . . 185
Action Bar with Pulldown Menus and BUTTON Bar . . . 186
Defining a Separate MENU Module . . . 187
Defining the Action Bar within a Form . . . 188
Defining an Action Bar and a Pulldown Menu . . . 189
Examples of Action Bars with Pulldown Menus . . . 193
Using an Action Bar with Pulldown Menus . . . 195
The BUTTON Bar . . . 196
Module Options . . . 196
The LIB Option . . . 196
The WRAP Option . . . 197
Language-dependent Programs . . . 198
Language-dependent Programs . . . 198
Language-dependent Literals in Procedures . . . 199
Language-dependent Literals in Forms . . . 199
Calling SQL-PL from Precompiled Programs . . . 201
Calling SQL-PL from Precompiled Programs . . . 201
EXEC SQLPL . . . 201
EXEC SQL PROC . . . 203
Examples . . . 204
Examples . . . 204
Restrictions . . . 205
Restrictions . . . 205

SQL-PL

Field of Application

SQL-PL Objects

SQL-PL Components

The Workbench

The SQL-PL Language

Stored Procedures

The Debugger

Forms

Language-dependent Programs

Calling SQL-PL from Precompiled Programs

Examples

Restrictons

Field of Application

In client-server configurations, the performance of a database application depends essentially on the number of client-server interactions.

The number of these interactions can be drastically reduced if a major proportion of SQL statements, which in well-structured programs are concentrated in an access layer, are executed in the server.

SQL-PL is the procedural SQL extension of the database system Adabas for creating database procedures, database functions, and triggers (in the following referred to as DBprocedures, DB functions, and triggers respectively or generically as stored procedures).

Database procedures are SQL-PL procedures that are executed by the server. From the perspective of the application developer, a database procedure is treated like an SQL statement. Database functions can be specified as user-defined functions within an SQL statement. They are processed along with the SQL statement into which they are embedded. Triggers, by contrast, are not called explicitly from an application program, but implicitly at the end of an INSERT, UPDATE or DELETE statement.

DB procedures serve to:

- reduce the interactions between client and server.
- centrally describe, provide with access rights, and centrally administer the application objects in the form of abstract data types.
- extend the facilities of the database in a customized way.

DB functions allow for:

- extending SQL statements in a customized way.
- computing values within SQL statements.

By means of triggers it is possible to:

- test complicated integrity rules.
- start derived database modifications for the current or any other row.
- implement complicated rules to protect against unauthorized access.

The SQL-PL workbench supports the development of stored procedures by means of an appropriate user interface with integrated version management.

In addition, SQL-PL offers an ideal test environment for DB procedures, DB functions, and triggers with an integrated development environment for stored procedures and interactive programs.

SQL-PL is a structured programming language with Pascal-like control structures. The SQL-PL language offers the following facilities:

- the standard functions

- the entire SQL language
- the simplified call of DB procedures
- the screen component FORM
- the integrated report generator
- the possibility of calling stored commands of Query
- an integrated debugger
- an integrated editor
- the possibility of calling any arbitrary system editor
- calling SQL-PL programs from the workbench, the operating system level or a precompiled program

Some of the mentioned elements of the SQL-PL language are restricted to the usage in SQL-PL programs and cannot be used in stored procedures.

This manual describes the development of stored procedures and SQL-PL programs and their execution in the workbench. The call syntax for calling SQL-PL programs from the operating system is contained in the "User Manual Unix" or "User Manual Windows". Likewise, the call syntax for calling from precompiled programs can be found in the appropriate manual.

The call syntax for calling DB procedures from other programs is described in the respective manuals on the precompilers, ODBC and the other Adabas components.

The call of DB functions within SQL statements can be found in the "Reference" manual. Triggers cannot be started explicitly.

SQL-PL Objects

The translation units written in SQL-PL are called modules. There are the following different types of modules:

- DB procedures
- triggers
- DB functions
- procedures
- functions
- forms
- HELP forms
- menus

Each new module is assigned to a definite program. The program is a collection of modules. A new program is implicitly established with its first module.

A program is executed when one of its modules is called. If no particular module is specified with a program call, SQL-PL assumes that the module with the name START has to be called. The start module can be a procedure or a form, but not a menu.

DB procedures, triggers, and DB function special role among the programs. They are written as modules in SQL-PL; they can be tested and modified as long as they have not been created in the database. After the workbench has created them as stored procedures in the database, they are explicitly or implicitly executed in the database kernel.

This chapter covers the following topics:

- General Properties of SQL-PL Modules
- DB Procedures
- Triggers
- DB Functions
- Procedures
- Functions
- Forms
- HELP Forms

- Menus
- Syntax of the SQL-PL Objects

General Properties of SQL-PL Modules

The programs of a user form his private SQL-PL library. For the programs in the private library, the user can grant the execute privilege to other users. These users can call the programs, but they cannot modify or delete them.

SQL-PL programs can only be called by users who are known to the database system. For users to be able to build their own programs, they must have the RESOURCE or DBA privilege; i.e., they must be able to create private tables in the database.

Each module consists of a module header and a series of statements. In the module header are specified the kind of module, its membership of a program, name within the program, the formal parameters, and the options, if any. The administration of the SQL-PL objects in the workbench depends on the specifications in the module header.

Only DB procedures, triggers, and DB functions can be created by the workbench to be executed in the database kernel. They are syntactically checked for being suited as stored procedures. Their parameters must be SQL data types. LONG columns are not valid for parameters. File processing, output to the screen, as well as statements for the administration of sessions and transactions are not allowed.

DB Procedures

DB Procedures are identified in the module header by the keyword DBPROC.

DB Procedures serve to formulate comp operations on application objects (abstract data types). Frequently recurring sequences of control structures and SQL statements used by a great number of users are collected in DB Procedures and executed by the database server. Thus communication between client server can be reduced.

To be able to execute a DB Procedure, the user must have the call privilege for it. This call privilege is granted by the owner of the DB Procedure and is independent of the privileges which the user may have for the tables and columns used in the DB Procedure. It can therefore happen that a user is allowed to execute SQL statements via a DB Procedure which outside this DB Procedure are not available to him.

DB Procedures are explicitly called from the application programming language. Within a DB Procedure, all SQL statements (DDL and DML) are available without restriction (DDL statements, however, only make sense for the owner of a DB Procedure). The call of further DB Procedures is also supported.

As in the case of any SQL statement, it must be ensured for a DB Procedure call that this call has the desired effects, if successful, and that it does not affect the database at all, if an error occurs. Therefore, DB Procedures are subject to the transaction management as it is common in Adabas D. To differentiate this behavior for DB Procedures, Adabas provides nested transactions (subtransactions).

A DB Procedure whose effect depends on a decision within the DB Procedure must be formulated with subtransactions. Each subtransaction can be reset or closed and remains subject to the superior transaction. Subtransactions can be nested to any degree in DB Procedures.

In DB Procedures, it is also possible to call procedures, functions, or further DB Procedures. By using SQL statements, triggers and DB functions can be applied. Operating system commands or application programs (C, Cobol) can be called asynchronously from DB Procedures.

The call of a DB Procedure from an application program works like an SQL statement. In particular, a DB Procedure call produces a return code as if an SQL statement had been executed. In addition to the return codes used by SQL, the developer of DB Procedures can use own return codes within a specific range of numbers.

DB Procedures cannot only create single rows but also tables as the result. It is recommended to use the feature that variables can be used as result table names in SELECT statements (see the "Reference" manual).

As long as DB Procedures have not been made known to the Adabas server and activated as such, they can be tested and executed with the debugger, like normal SQL-PL procedures.

Triggers

Triggers are identified in the module header by the keyword TRIGGER.

Triggers are special DB Procedures that are implicitly called by the Adabas server after the table or column associated with the trigger has been modified (INSERT, UPDATE, DELETE). Triggers are used to test complicated integrity rules, to start derived database modifications for the current row or any other row and to implement complicated rules for access protection.

A trigger can be created for each table and INSERT, UPDATE, or DELETE statement. The same trigger can also be valid for several statements on the same table. The context for the call can be defined in apredicate within the statements of a trigger.

The trigger parameters must correspond to the associated table columns. Within the trigger, processes can be performed with both the old column values (UPDATE, DELETE) and the new column values (UPDATE, INSERT).

A trigger can implicitly call other triggers and explicitly call procedures, functions, and DB Procedures. Each trigger implicitly constitutes a subtransaction. A trigger is an integral component of the releasing SQL statement. The releasing SQL statement is assumed to have failed and has no effect on the data if the trigger ends with a return code <> 0 set by the user.

The same functional limitations valid for DB Procedures are valid for triggers.

DB Functions

DB functions are identified in the module header by the keyword DBFUNC.

DB functions serve the customized extension of the functions that can be executed in SQL statements. Their result is a value with an SQL data type; the value is also used in the embedding SQL statement. A return code can be set in a DB function. This return code can cause the failure of the SQL statement. DB functions must neither contain SQL statements nor call procedures. The call of functions is allowed if these comply with the restrictions.

The user needs DBA privileges to create a DB function in the Adabas server. After its creation, the DB function can be executed by all users without more privileges.

Procedures

Procedures are indicated in the module header by the keyword PROC.

SQL-PL procedures control the program flow, access to the data and the preparation of result data for output.

Procedures communicate with other procedures or forms of the same program either via common variables that are global within the program or via explicit module parameters.

Procedures can also call functions and DB Procedures. In this case, communication takes exclusively place via parameters.

Branching from one procedure of a program to any procedure of another program is possible, as long as the user has the call privilege. The type of CALL statement determines whether or not this other program returns to the calling program after having been executed.

Functions

Functions are distinguished from SQL-PL procedures by the initial keyword FUNCTION . A value can be returned to the calling environment using the RETURN statement of the function.

Functions can call functions, but no procedures or forms. The call of SQL statements is allowed.

Facilities required in the entire application (i.e. in several programs), can be collected in a library of functions. After calling a function, the control is always returned to the calling module (see Section, "Calling Functions").

Forms

A form is introduced in the module header by the keyword FORM. Forms consist of a layout part and a processing part.

A form can call procedures, functions, DB Procedures, forms, HELP forms and menus. Forms and procedures of a program can communicate via common global variables.

A detailed description is contained in Section, "Forms".

HELP Forms

HELP forms are identified in the module header by the keyword HELPFORM.

HELP forms are forms that support the output of help information by automatically positioning the information on the screen, according to the current field.

In HELP forms, only local variables are available, and only further HELP forms can be called.

Menus

Menus serve to define pulldown menus. Pulldown menus can also be defined in forms. In separate menu modules, however, they have the advantage of being able to be called by various forms. A detailed description is contained in Section, "General Points on Forms and Menus" Section "Menus".

Syntax of the SQL-PL Objects

```
<db procedure> ::= DBPROC <prog name>.<mod name>
                   [PARMS (<dbproc parm decl>,...)]
                   [OPTIONS (<module option>,...)]
                   [<var section>]
                   <lab stmt list>

<dbproc parm decl> ::= <dir> <name> <data type>

<dir> ::= IN | OUT | INOUT

<data type> ::=  FIXED [ ( <unsigned integer> [, <unsigned integer> ] ) ]
              |  FLOAT [ ( <unsigned integer> ) ]
              |  CHAR  [ ( <unsigned integer> ) ] [ BYTE ]
              |  DBYTE [ ( <unsigned integer> ) ]
              |  BOOLEAN
              |  DATE
              |  TIME

<module option> ::= see "Module Options" (5.14)
                    in Section "The SQL-PL Language"

<var section> ::= see "Variable Declaration" (5.2)
                  in Section "The SQL-PL Language"

<lab stmt list> ::= [<label>] <compound> [<lab stmt list>]

<compound> ::= BEGIN <stmt>;... END | <stmt>

<trigger> ::= TRIGGER <prog name>.<mod name>
              [PARMS (<trigger parm decl>,...)]
              [OPTIONS (<module option>,...)]
              [<var section>]
              <lab stmt list>

<trigger parm decl> ::= IN <trigger column spec> <data type>

<trigger column spec> ::= <column name>
                        | NEW.<column name>
                        | OLD.<column name>

<db function> ::= DBFUNC <prog name>.<mod name>
                  [PARMS] (<dbfunc parm decl>,...) : <data type>
                  [OPTIONS (LIB <prog name>)]
                  [<var section>]
                  <lab stmt list>

<dbfunc parm decl> ::= IN <name> <data type>
```

```
<procedure> ::= PROC <prog name>.<mod name>
                [[PARMS] (<parm decl>,...)]
                [OPTIONS (<module option>,...)]
                [<var section>]
                <lab stmt list>

<parm decl> ::= <varname> [()]

 <function> ::= FUNCTION <prog name>.<mod name>
                [PARMS (<parm decl>,...)]
                [OPTIONS (<module option>,...)]
                [<var section>]
                <lab stmt list>

 <form> ::= FORM <prog name>.<mod name>
                 [OPTIONS (<form option>,...)]
                 [PARMS (<parm decl>,...)]
            [<var section>]
            <form layout>
            [ <processing spec>;... ]

<help form> ::= HELPFORM <prog name>.<mod name>
                         [OPTIONS (<form option>,...)]
                         [PARMS (<parm decl>,...)]
                [<var section>]
                <form layout>
                [ <processing spec>;... ]

<menu> ::= MENU <prog name>.<mod name>
           [PARMS (<formal parameter>,)]
           <actionbar>
           [ <pulldown> ]
```

SQL-PL Components

SQL-PL comprises different components.

The Workbench and Its Tools

When calling SQL-PL, the built-in development environment of SQL-PL (in the following referred as workbench) is displayed. The built-in editor, debugger, and Express program generator can be used to develop stored procedures and application programs. The workbench is an application program written in SQL-PL.

The SQL-PL Compiler

The SQL-PL compiler is called by the workbench if a module is to be checked for correct syntax and stored in the database. The compiler performs a special syntax check for modules that are to be created as DB Procedures, DB functions or triggers in order to check their suitability. The compiler generates a binary program code that is stored in the Adabas database in addition to the source text of the module.

While creating a stored procedure in the database, the compiler translates the called subprocedures and subfunctions and checks their syntax. The workbench then updates the relationships and dependencies in the catalog of the database.

The SQL-PL Interpreter for Application Programs

The interpreter loads application programs as binary program code from the database. The application programs are executed within the workbench context. The workbench is an application program and therefore completely stored in the database. It is executed by the SQL-PL interpreter in the operating system context.

The SQL-PL Interpreter in the Database Kernel

There is a variant of the SQL-PL interpreter in the Adabas database kernel to process stored procedures. This interpreter is called whenever a stored procedure is to be executed explicitly (DB Procedure or DB function) or implicitly (trigger). The corresponding binary program code is loaded from the internal tables without network communication and is then executed in the context of the current session and transaction.

The Workbench

This chapter covers the following topics:

- The Handling of the Workbench
- The Function Keys of the Workbench
- The Workbench for Beginners
- The Version Administration
- The Action Bar of the Workbench
- Commands
- User-specific Set Parameters

The Handling of the Workbench

The SQL-PL workbench has a menu-driven user interface. After calling SQL-PL (see the "User Manual Unix" or "User Manual Windows") an action bar is displayed that contains a list of all SQL-PL programs owned by the user. The display can be extended to all SQL-PL programs for which the user has got the execute privilege using the ’Selection/Show/ Program List/Owner’ menu item.

Example: Program List

```
 ____________________________________________________________________________
|                                                                            |
| Object.. Selection.. Privilege.. Test.. Tools.. Scroll.. Info..            |
|____________________________________________________________________________|

 ____________________________________________________________________________
|              |                          |                                  |
| Owner        |       Program            |          Comment                 |
|______________|__________________________|__________________________________|
|                                                                            |
| BROWN                CUSTOMER                    customer register         |
| BROWN                CARD                                                  |
| MILLER               ADDRESS                     address management        |
| GAMES                TIC_TAC_TOE                 play program              |
| WBDEMO               CALC                        desk calculator           |
|                                                                            |
|                                                                            |
|                                                                            |
|___*.*________ customerdb:BROWN _______________ TEST _____________001-005 __|
```

If there are no own programs, only a list of those programs is displayed for which a user has got an execute privilege from other users (GRANT EXECUTE). If there is not even an execute privilege for foreign programs, an empty program list is displayed.

Stored procedures are visible if the user has the execute right for the programs.

There are various possibilities of branching from the program list to the module list:

- Pressing the ENTER key displays the module list of the program selected by means of the cursor.
- Via the ’Object’ menu item the user selects the ’Show’ function which has the effect of the ENTER key. To switch between the menu bar and the form the F12 key is used.
- Via the ’Selection’ menu item ’Show’ and ’Module List’ functions a dialog box appears. The list of the desired modules can be selected here by explicitly specifying the arguments (or search patterns).
- In the command line which is activated by means of the ’Tools/Workbench Commands’ menu item the user enters, e.g., the command ’ml c.*’, releasing it with ENTER.

Example: Selection/Show/Module List and Program Name: C*

```
 ____________________________________________________________________________
|                                                                            |
| Object.. Selection.. Privilege.. Test.. Tools.. Scroll.. Info..            |
|____________________________________________________________________________|

 ____________________________________________________________________________
|           |             |       |       |      |                           |
| Program   | Module      | Type  | State | Vers | Last Change               |
|___________|_____________|_______|_______|______|___________________________|
| CARD        CHECK         FUNC    RUN     PROD    02/07/10 15:23            |
| CARD        INFO          HELP    RUN     PROD    02/07/10 15:23            |
| CARD        START         PROC    RUN     PROD    02/07/10 15:24            |
| CUSTOMER    FIRST         PROC     +DB    PROD    02/08/03 12:31            |
| CUSTOMER    INPUT         FORM    RUN     PROD    02/08/03 11:19            |
| CUSTOMER    NEXT          DBPROC  ->DB    PROD    02/08/03 12:31            |
| CUSTOMER    OUTPUT        FORM    EDIT    TEST    02/08/03 13:03            |
| CUSTOMER    START         PROC    DEBUG   PROD    02/08/03 13:12            |
|                                                                            |
|__ BROWN.C* ____________ customerdb: BROWN ___________ WORK ________ 001-008 __|
```

The following diagram shows how a user can get in a very simple way from the program list to the module list and from there to the editor in order to edit an existing module.

```
_________________            _________________             _________
|               |            |               |             |         |
| Program List  |  ENTER ->  | Module List   | ENTER ->    | Editor  |
|               | __________ |               |___________  |         |
|               |  <- END    |               | <- END exit |         |
|               |  <- F3     |               | <- F3 exit  |         |
|_______________|            |_______________| <- F9 store |_________|
```

All workbench functions can be activated either via the action bar and pulldown menus or explicitly via the command line. For a series of functions, such as ’Help’, ’End’ etc., the function keys are provided as an alternative.

The Function Keys of the Workbench

The most important functions are also assigned to function keys.

Keys are assigned to the following functions:

Key	Function
F1	Help
F3	END or Back
F4	Printing the currently shown workbench display
F5	Starting the module or program
F8	Marking
F9	Refreshing
F12	Switching between action bar and form
PgUp	Scrolling one page up
PgDn	Scrolling one page down
CMD	Activating the command line not available in all systems (in some systems CTRL/F1)

In this manual, a key symbol with a label which does not correspond to a key of the standard keyboard, e.g., the SAVE key, refers to visible buttons in the form to which a key or a choice character has been assigned as a rule.

Workbench function keys within the editor

The detailed description of the built-in editor is included in the "User Manual Unix" or "User Manual Windows". The following editor keys are important for the SQL-PL workbench:

F2 Module Save

The currently edited module is stored in the database by means of the SAVE key (corresponds to the command SAVE) without checking it syntactically and converting it into the internal code. Afterwards the module is in the ’EDIT’ state.

F5 Module Test

When the TEST key is pressed, the current module is syntactically checked and then started. One can correct and test a module, until the result is satisfactory; then the module can be stored, whereby the editor will be left. When a module and the program is tested, all modifications made to database contents are rolled back at the end of the test run.

F6 Module Values

This function key is not displayed, until the test execution has been terminated. Pressing VALUES generates a list of all variables of the module displayed within the editor. The list contains the current values of the last TEST execution. The display of the variables can be restricted by specifying a search argument.

F9 Module Store

When the STORE key is pressed (corresponds to the STORE command), the current module is syntactically checked and, if this check has been successful, stored in the database in its internal code. Thereby all pieces of information about the relations are maintained in the Data Dictionary (DOMAIN), and the editor is left.

In the editor, stored procedures in ’->DB’ status cannot be saved, tested or stored using the same name. Before changing them, they must be removed from the database.

The Workbench for Beginners

For a user who does not yet have own programs, SQL-PL displays an empty program list. To rapidly get to know SQL-PL, one can proceed in the following way:

1. One sets the desired language for the SQL-PL messages (’Tools’ menu item ’Set Parameters’ function).

2. One defines one or more tables. For this purpose it is recommended to use the component Domain. But this can also be done via the integrated SQL window (or, of course, by means of the components Load or Query).

3. One calls the tool ’EXPRESS’ (’Tools’ menu item) and generates a program or only a form.

When returning from EXPRESS, the program list contains the programs generated by EXPRESS for maintaining the master data. These programs can be extended at will.

The Version Administration

The workbench has three version levels: the test version, the production version, and the historical version.

The purpose of these versions is to allow a user to work with the production version of a program, while the developer of the program develops it further in its test version, until the tested version can be released as extended production version at a future point in time. Thereby the historical version is the backup of the last production version.

The modulees of stored procedures are also subject to the version administration. The version has no influence on the execution in the database kernel. In any case, there is only one executable version of a stored procedure in the database kernel.

As long as the ’Release Version’ function has not been used, all modules exist as test version only.

The Test Version:

All modules currently edited are first saved as test version. Only modules of the test version can be processed with the debugger. A module of the production version can be edited, but only be modified as test version.

The Production Version:

By means of the ’Release Version’ function under the ’Object’ menu item a program is converted from its test version to the production version. The test version will be deleted.

A production version can be recalled by means of the function ’Recall Version’. Thereby the historical version, if any, is activated as production version. To be able to do so, there must not be any module of the program as test version.

When generating the production version, the current usage relations are entered into the Data Dictionary. This usage information can be easily retrieved using Domain.

The following relations are concerned:

-	MODULE	CALLS	DBPROCEDURE
-	MODULE	CALLS	MODULE
-	MODULE	USES	COLUMN
-	MODULE	USES	QUERYCOMMAND
-	MODULE	USES	TABLE
-	DBPROCEDURE	CONTAINS	PARAMETER
-	TRIGGER	CONTAINS	PARAMETER
-	USER	USES	MODULE
-	USER	USES	DBPROCEDURE

The Historical Version:

The historical version always contains the previously valid complete production version and serves as backup copy.

Working with the Test and Production Version

When calling SQL-PL for the first time, the default setting is the TEST version. The user can switch between the different versions by modifying the version setting in the Set menu (’Tools/Set Parameter’ menu item). If the version is set to TEST, at runtime the modules and programs are always looked for in the order of test version, production version. By this means only the modified modules are kept twice which, together with the unchanged modules of the production version, form the current test version. When working in this way, the version identification ’WORK’ is visible in the bottom frame of the workbench window.

The desired version can also be specified when selecting a program list or module list which allows quickly changing from a list of the production version to a list of the test or historical version. For better handling it is possible to display lists merged from the production and test version, since these module lists exactly reflect the modules that have been called for testing. For this purpose one uses the version identification ’WORK’. This is the current development view at the modules. Similarly, the programs or modules of all versions are displayed, when the version identification ’ALL’ has been specified.

If it should be necessary to completely delete a program of the ’PROD’ version and there are still modules for this program in ’TEST’ version, this program is shown in the program display in ’PROD’ version although there are no modules visible. When changing to the version identification ’WORK’, the modules will be displayed.

Further particulars of the versions of a program are described in connection with the different functions.

The Action Bar of the Workbench

The action bar of the workbench appears in the program list as well as in the module list. First the cursor is positioned in the list. The action bar can be activated in various ways:

- Each menu item can be selected directly by simultaneously pressing the highlighted letter and the key CTRL (or Strg or Control), thus displaying the pulldown menu.

- Pressing the F12 key activates the first menu item. Here the desired menu item can be selected either by means of the cursor keys or by pressing the highlighted letter. Pressing ENTER displays the pertinent pulldown menu. If the action bar is activated, the F12 key returns to the program list or module list.

The cursor key graphics/sqlpl41.gif pressed in a pulldown menu displays the pulldown menu of the next level, and if this is not available, the adjacent pulldown menu of the first level. The graphics/sqlpl41.gif key pressed in a further level also displays the adjacent pulldown menu of the first level. The cursor key graphics/sqlpl42.gif returns from the second level to the first level and from there to the left adjacent first-level pulldown menu.

Control is circular, this means, the graphics/sqlpl41.gif key pressed in the right-hand pulldown menu returns to the first pulldown menu at the left.

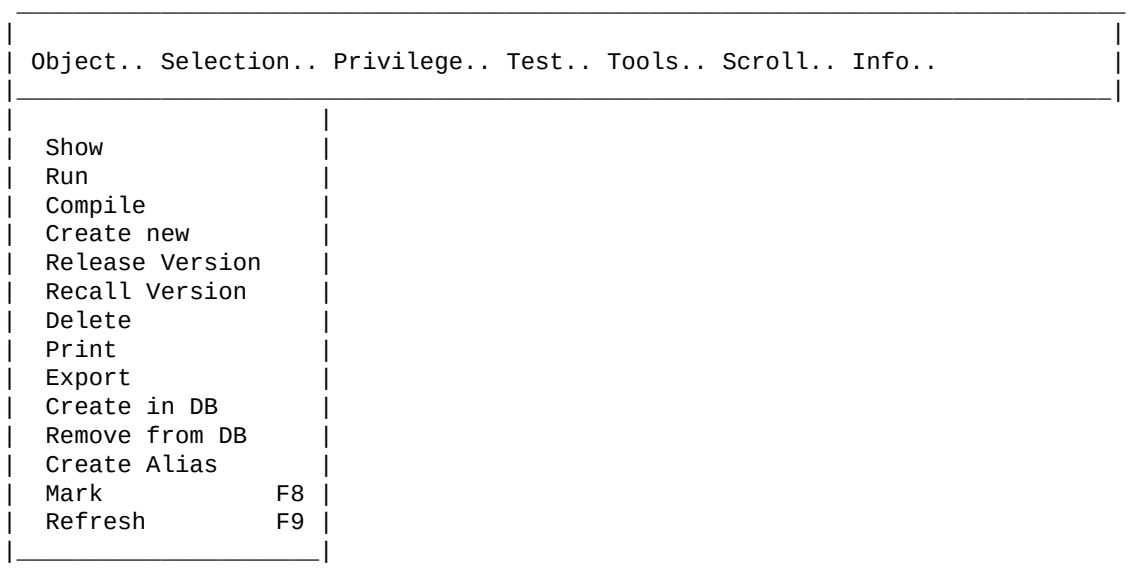
A function of a pulldown menu is activated either by positioning the cursor and pressing ENTER or by selecting the highlighted letter.

This section covers the following topics:

- The ’Object’ Menu Item
- The ’Selection’ Menu Item
- The ’Privileges’ Menu Item
- The ’Test’ Menu Item
- The ’Tools’ Menu Item
- The ’SCROLL’ Menu Item
- The ’INFO’ Menu Item

The ’Object’ Menu Item

After selecting the ’Object’ menu item the following pulldown menu appears:



```
|                       |
|  Back      F3,END     |
|  Leave Workbench      |
|_______________________|
```

The first group of functions of the ’Object’ menu item refers to the object on which the cursor is positioned, thus behaving context-specifically. The other functions ’Back’ and ’Leave Workbench’ have the same meaning everywhere.

Object/Show

displays, in the program list, all modules of the program on which the cursor is positioned (see Section, The’Selection/Show/Module List ’Menu Item"). ’Show’ in the module list has the effect that that module is displayed on which the cursor is positioned. The module is displayed within the built-in editor so that all editing functions are available.

(Command: ’PLIST’ or ’MLIST’)

Object/Run

in the program list has the effect that the module START of the current program is started. If this module does not exist, an error message is output. If ’Run’ is selected within the module list, the module designated by the cursor bar is started.

(Command: ’RUN’)

Object/Compile

in the program list has the effect that all modules of the program selected by means of the cursor bar are converted into the internal code. In the module list, according to the other functions, only the module designated by the cursor bar is converted. If there are marked objects in the list, the function is executed just for these objects.

(Command: ’STORE’)

Object/Create new

has the effect within the program list as well as within the module list that the editor is called enabling the user to edit and store a new module.

(Command: ’EDIT’)

Object/Release Version

refers to a program and can therefore only be activated within the program list. This function generates a production version from the TEST version of the designated program. If there is already a production version available, this will be saved as historical version. For the specified program there must be at least one module in the TEST version. Modules which have the EDIT or DEBUG state are translated and usage information relating to these modules is entered into the Data Dictionary. Afterwards all test version modules of the program are stored as production version and there is no test version module for this program any more. This function affects marked objects as well.

(Command: ’MKPRODUCT’)

Object/Recall Version

cancels the release version, i.e. the historical version becomes a production version again and the previous production version becomes the test version. The command is rejected, if there are already modules in the test version.

(Command: ’GETHIST’)

Object/Delete

has the effect that the program (in the program list) or the module (in the module list) designated by the cursor bar is deleted. This function must be confirmed - in order to prevent a handling error. This function affects marked objects as well.

(Command: ’DROP’ or ’DELETE’)

Object/Print

has the effect that a program (all modules of the program) or a module is printed. This function must be confirmed. It affects marked objects as well.

(Command: ’PRINT’)

Object/Export

A dialog box appears into which the file name of the export file to be generated has to be entered. The function exports either a whole program or a single module. The function affects marked objects as well.

(Command: ’EXPORT’)

Object/Create in DB

allows the DB Procedure, DB function or trigger designated by the cursor to be created in the database kernel. This function can only be selected in the module list. A module can be created in the DB kernel, when it was defined with the module type ’DBPROC’, ’DBFUNC’ or ’TRIGGER’ and when there is no module with the same name of another version in the DB kernel.

When processing this function, the module and all procedures and functions called by it are newly checked for syntactical correctness in the context of a stored procedure and then be entered in the system tables. In the workbench, stored procedures successfully created are identified with the ’->DB’ state. Called subprocedures are given the ’+DB’ state. If the module or a called module does not satisfy the restrictions of stored procedures, this function fails.

Stored procedures which have been created in the DB kernel cannot be modified. To be able to change them, they must be removed from the DB kernel. This function affects marked objects as well.

(Command: ’PCREATE’ or ’TCREATE’)

DB Procedures

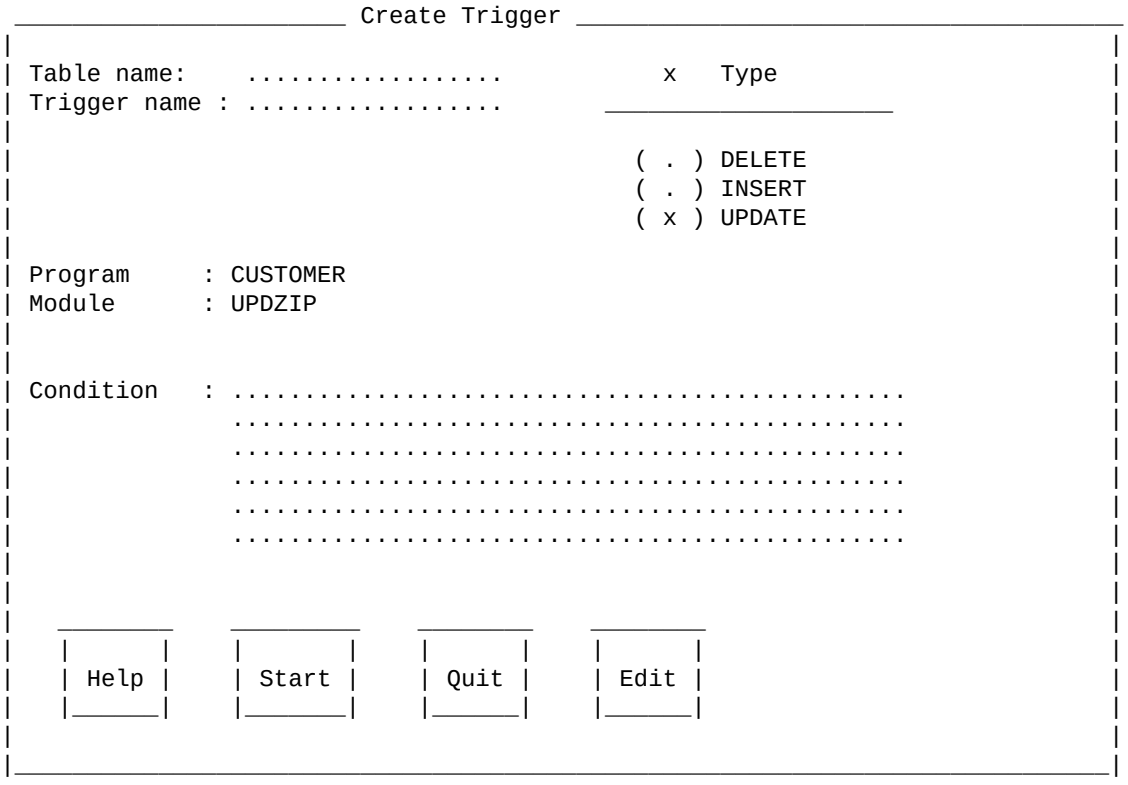
For DB Procedures, the function generates the message whether the creation was successful or not; if need be, an error message is output.

DB Functions

For DB functions, the name can be defined to be used when calling a DB function in the database kernel. This name must be unique in the catalog. The module name is the default.

Trigger

If the module type is ’TRIGGER’, the following screen is output for the specification of the table, the trigger name and trigger type, as well as a restricting condition.



If the help function is used while the cursor is on the field table name or trigger name, the list of all tables or of all triggers is displayed for support. The trigger name is used for identification purposes in the catalog; it must be unique. It has no meaning for the handling of the trigger.

The trigger is executed after successful processing of the specified SQL statement, if the condition that can be specified in the form is satisfied.

Triggers of the type ’INSERT’ or ’DELETE’ are released when a row is to be inserted into the specified table or deleted from the table. The corresponding values can be accessed in the trigger module using the prefix ’NEW’ or ’OLD’.

If the trigger type ’UPDATE’ is selected, the list of all column specified table is displayed to mark the columns for which the trigger is to be called when they are updated. The trigger is released when the new value is not equal to the old one for at least one of the specified columns. An UPDATE with the same values is optimized by the database kernel; it can happen that the trigger is not released.

The syntax of the expression that can be entered in the field ’Condition’ must correspond to the search condition> (see the "Reference" manual). The column values NEW.<columnname> and OLD.<columnname> can be used within the <search condition> (see Section, "Triggers"). The trigger is called if the condition is satisfied.

Object/Remove from DB

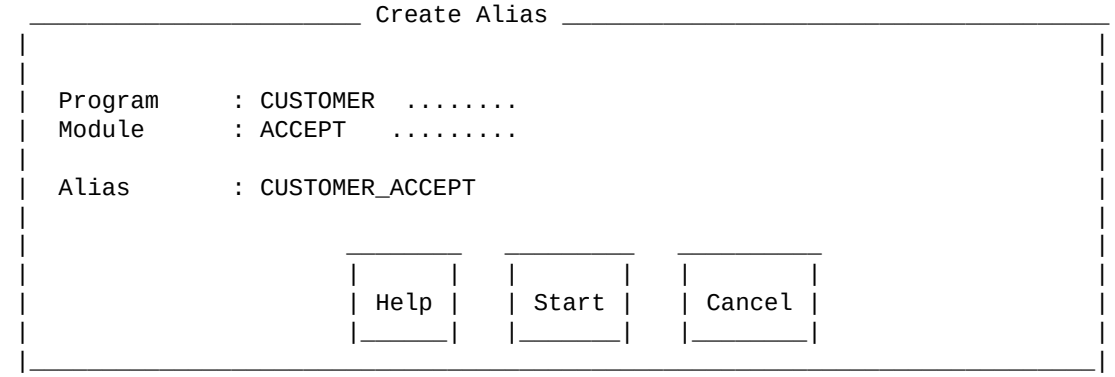
removes the DB Procedure or the trigger from the database kernel. The module can be changed afterwards.

Subprocedures are only removed from the DB kernel, if they are not used by another DB Procedure, DB function or trigger created in the DB kernel. After successful execution the state in the module list is set to ’RUN’ again. This function affects marked objects as well.

(Command: ’PDROP’, ’FDROP’ or ’TDROP’)

Object/Create Alias

defines an alias name for a DB Procedure. This name is used to identify the DB Procedure from programs that use the ODBC or JDBC library. The alias name comprises the <prog name>.<mod Name> notation to form one name. This is necessary to comply with the ODBC or JDBC call syntax owner>.<aliasname>. If there is already an alias name, a message is displayed instead of the dialog box. The alias name is automatically dropped when the DB Procedure is dropped. The ’Selection/Show’ menu item can be used to display the alias names for all DB Procedures.



Object/Mark

serves to mark the object (program or module) on which the cursor is positioned. An object is marked when the frame line in the line of the object is interrupted by an ’*’. The functions ’Compile’, ’Release Version’, ’Delete’, ’Print’, ’Export’, ’Create in DB’, and ’Remove from DB’ can be applied to marked objects. Marking a line that is already marked removes the mark. This function can also be executed by means of the F8 key.

Object/Refresh

serves to rebuild the current list. When doing so, all marks are removed. This function can also be executed by means of the F9 key.

Object/Back

leads back from the current list to the list from which one has come. Thus one can trace back the hierarchy of the lists that have been selected one after the other. At the first hierarchy level the function ’Object/Back’ is equivalent to ’Leave Workbench’ and can therefore not be chosen.

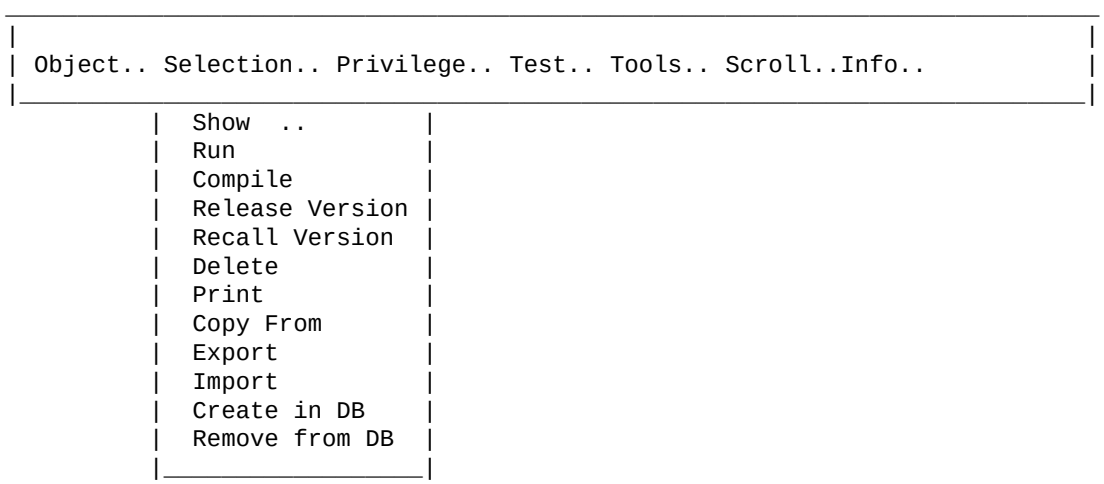
Object/Leave Workbench

terminates the workbench after confirmation.

(Command: ’EXIT’)

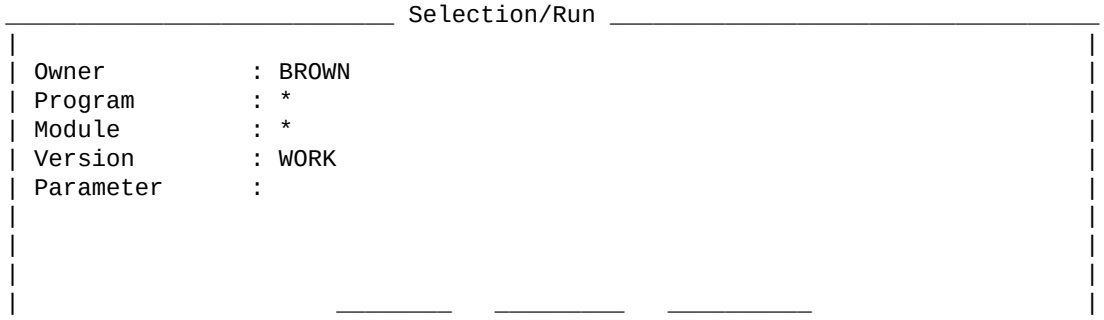
The ’Selection’ Menu Item

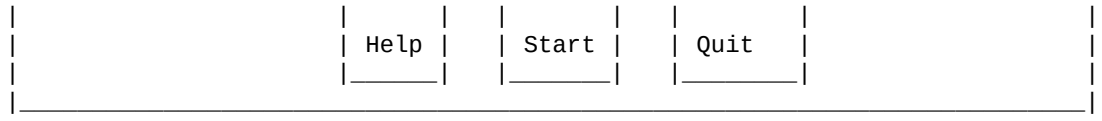
After selecting the ’Selection’ menu item the following pulldown menu appears:



The pulldown menu ’Selection’ refers to a set of objects which has to be specified in a dialog box by means of search patterns and search arguments. Here basically the same functions are offered as with ’Object’.

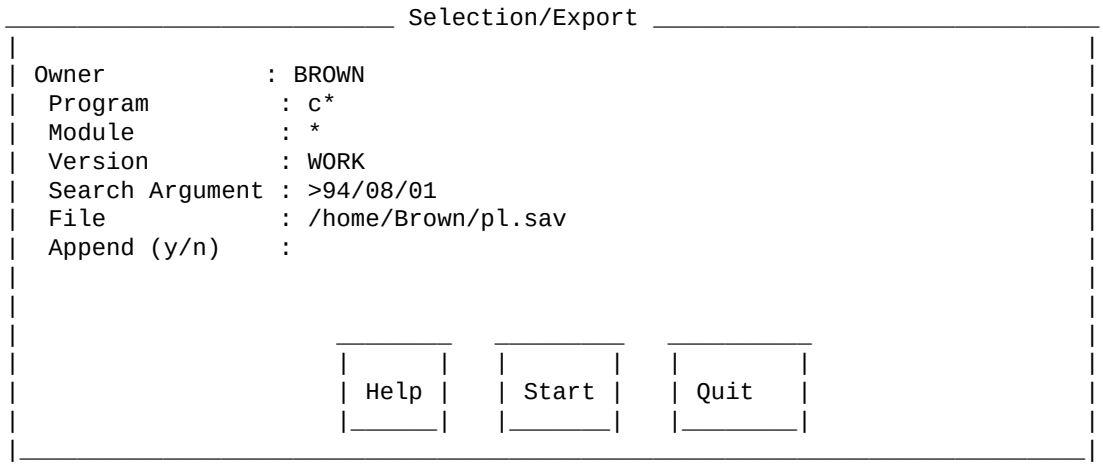
Example: Defaults of a Dialog Box





As default the current database user and the version of the last displayed list are proposed. Instead of unique program and module names search patterns can be used to limit the eligible programs and modules. '*' stands for any number of characters and '?' for exactly one arbitrary character.

Example: Filled Dialog Box



Version can be set to:

Version = TEST | PROD | HIST | WORK | ALL

As search argument the type of module, the state or the date can be specified.

```
search argument = FORM    | PROC   | HELP    | FUNC
                | DBPROC  | DBFUNC | TRIGGER | EDIT
                | DEBUG   | RUN    | ->DB    | +DB
                | NOCALL  | =date  | <date   | >date
```

The date must be entered in the format as specified in Set and displayed in the module list. The date must be preceded by an operator with the following meaning:

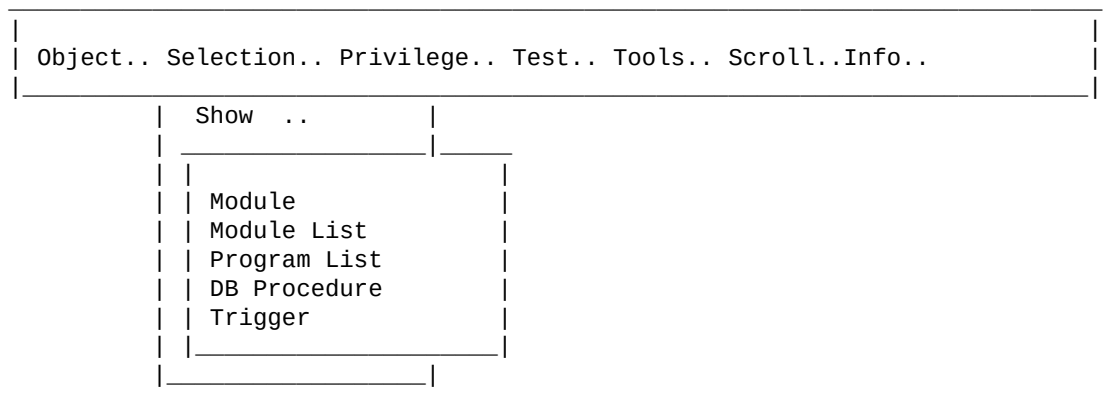
```
'=' : corresponds exactly to the specified date
                    '<' : older than the specified date
               '>' : younger than the specified date
```

The search argument 'NOCALL' refers to all modules which are not called by other modules, e.g. start modules or dead code.

If the form was correctly filled in and confirmed with START or the ENTER key, the selected function is executed.

Selection/Show

provides the following submenu:



Selection/Show/Module

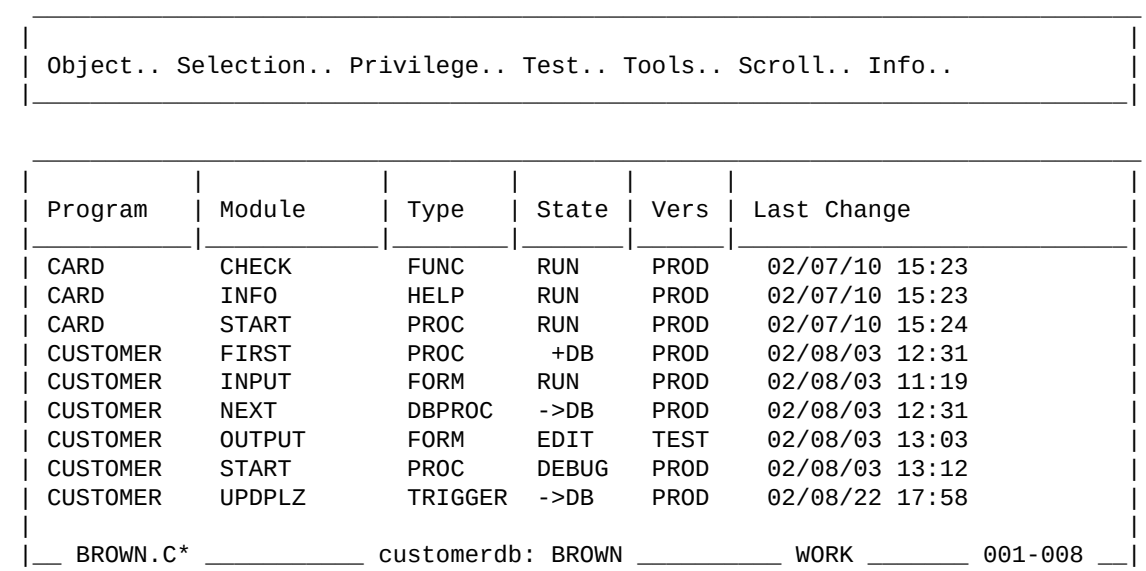
A dialog box for specifying a certain module is displayed. The input arguments must describe a particular module, i.e. they cannot be search patterns. The editor is called in order to modify the specified module. The editing functions are described in the "User Manual Unix" or "User Manual Windows".

(Command: ’EDIT’)

Selection/Show Module List

A dialog box appears where a search pattern for program and module names, the desired version and, if needed, a search pattern can be specified in order to have a list of modules displayed.

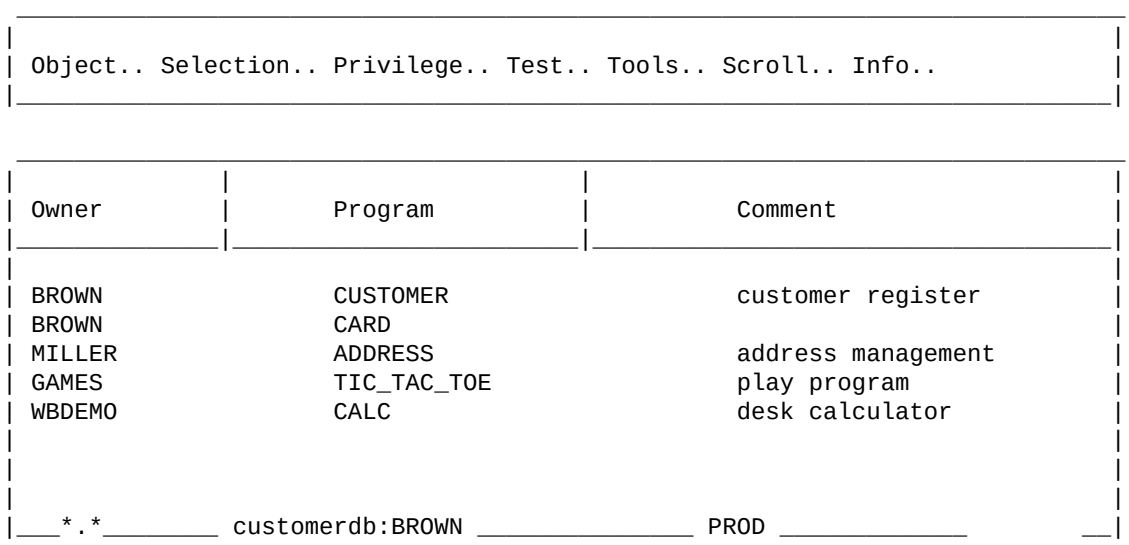
(Command: ’MLIST’)



Selection/Show/Program List

A dialog box appears where a search pattern for the program name and the desired version can be specified in order to have a list of programs displayed.

(Command: ’PLIST’)



Selection/Show/DB Procedure

requires that in a dialog box a search pattern is specified for the program name of one or more DB Procedures. Then a list is output containing all DB Procedures with this name stored in the database.

(Command: ’PSHOW’)

Example:

OWNER	PROGRAM	DBPROCNAME	ALIASNAME	PARAMETER	EXECUTABLE	GRANT
BROWN	CUSTOMER	NEXT	CUST_NEXT	0	YES	YES

Selection/Show/Trigger

requires that in a dialog box a search pattern is specified for the program name of one or more triggers. Then a list is output containing all triggers stored in the database that underlie these programs.

(Command: ’TSHOW’)

Example:

OWNER	TABLENAME	TRIGGERNAME	TYPE	PROGRAM	DBTRIGGER
BROWN	HOTEL	NEWZIP	UPDATE	CUSTOMER	UPDTRIG

Selection/Run

A dialog box for the specification of a certain module is displayed. The input arguments must describe a particular module, i.e. they cannot be search patterns. Parameters separated from each other by blanks can be specified. They are assigned to the formal parameters of the procedure. The module must have the RUN state (see state in the module list) and belong to a program for which one has the execute privilege. If no module name is specified, the module ’START’ is called, if such a module exists. For programs of other users a module name has to be specified.

To be able to test with the SQL-PL debugger, the command DEBUG ON must have been issued and the module to be tested must have been stored again. The debugger will be activated when executing the program. For precise information see Section, "TheDebugger".

(Command: ’RUN’)

Selection/Compile

A dialog box appears for specifying a search pattern for program and module names, the desired version optional search argument. The set of modules described by the arguments is syntactically checked and, if this check was successful, stored as executable object with the ’RUN’ state. In the case of an error, an error text is output and the module obtains the state ’EDIT’.

The modules of the PROD and HIST version can also be recompiled e.g. in order to make changed literal contents known to the program.

(Command: ’STORE’)

Selection/Release Version

makes a production version from a test version. In this case the arguments in the dialog box must describe a program or a set of programs. If the field ’->HIST’ is filled with ’Y’, a production version which might already exist will be saved as historical version. For the specified program there must be at least one module in the TEST version. Modules which are in EDIT or DEBUG state are compiled and entered into the Data Dictionary.

(Command: ’MKPROD’)

Selection/Recall Version

cancels the release version, i.e. the historical version becomes a production version again and the previous production version becomes the test version. The command is rejected, if there are already modules in the test version.

(Command: ’GETHIST’)

Selection/Delete

removes the specified program or module from the user’s SQL-PL library. This function must be confirmed - in order to prevent a handling error. Applied to a foreign program, the execute privilege is deleted.

(Command: ’DROP’ or ’DEL’)

Selection/Print

outputs the contents of the specified set of modules on the printer specified in Set. This function must be confirmed - in order to prevent a handling error.

(Command: ’PRINT’)

Selection/Copy From

copies the desired program of the specified owner into the user’s own SQL-PL library. Prerequisite is that the owner has granted a copy privilege. All modules are copied in the original state and existing version.

(Command: ’COPY’)

Selection/Export

writes the selected set of programs or modules into an operating system file. The modules are separated from each other by ENDMODULE. Privileges granted are written into the export file in form of workbench commands. If in the dialog box in the field ’Append (y/n) :’ ’y’ is entered, the commands are added to an existing file.

(Command: ’EXPORT’)

Selection/Import

reads programs from an operating system file that contains exported modules. The individual modules are expected to be separated from each other by ENDMODULE. The privileges commands contained in the file are executed.

(Command: ’IMPORT’)

Selection/Create in DB

A module can be selected to be created in the database kernel (see ’Object/Create in DB’).

(Command: ’PCREATE’ or ’TCREATE’)

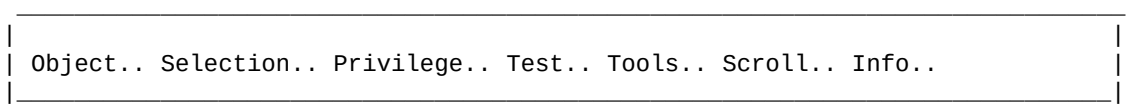
Selection/Remove from DB

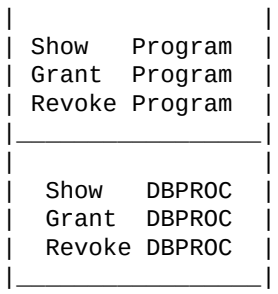
removes the DB Procedure or trigger from the database kernel. Afterwards the module can be changed (see ’Object/Remove from DB’).

(Command: ’PDROP’ or ’TDROP’)

The ’Privileges’ Menu Item

After selecting the ’Privileges’ menu item the following pulldown menu appears:





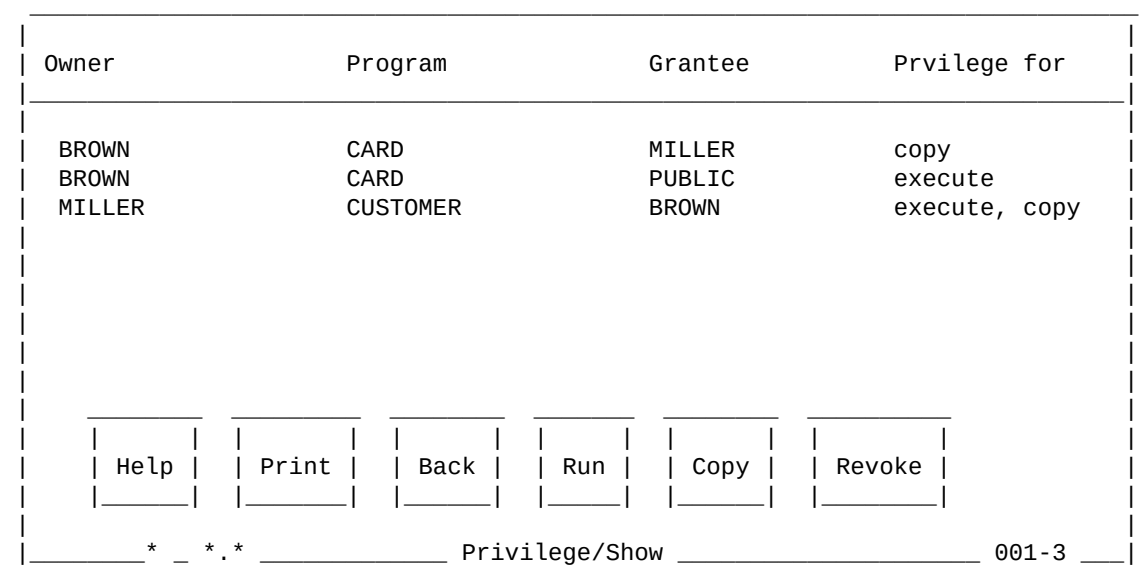
This pulldown menu comprises all functions related to the authorization of modules.

Privileges/Show Program

All execute and copy privileges for SQL-PL programs are displayed which one has granted to other users or has got from other users (grantees). A dialog box appears in which the result list can be restricted by specifying the owner, program, grantee, and privilege.

(Command: ’PRIVILEGES’)

The additional functions Run, Revoke, and Copy are provided by push buttons.



Privileges/Grant Program

grants the explicit execute or copy privilege for a program to a particular user or the implicit execute or copy privilege (PUBLIC) to all SQL-PL users. No execute privileges need to be granted for successor programs that are called from this program. The copy privilege also comprises the execute privilege

(Command: ’GRANT EXECUTE’, ’GRANT COPY’)

Privileges/Revoke Program

removes the explicit execute or copy privilege for a certain program from a particular user or the implicit execute or copy privilege for a certain program from all users (PUBLIC).

(Command: ’REVOKE EXECUTE’, ’REVOKE COPY’)

Privileges/Show DBPROC

All execute privileges for DB Procedures are displayed which one has granted to other users or has got from other users (grantees). A dialog box appears in which the result list can be restricted by specifying the owner, program, grantee, and privilege.

(Command: ’PPRIVILEGES’)

Privileges/Grant DBPROC

grants the explicit execute privilege for aDB Procedure to a particular user or the implicit execute privilege (PUBLIC) to all SQL-PL users. No privileges need to be granted for successor procedures that are called from this DB Procedure.

(Command: ’PGRANT’)

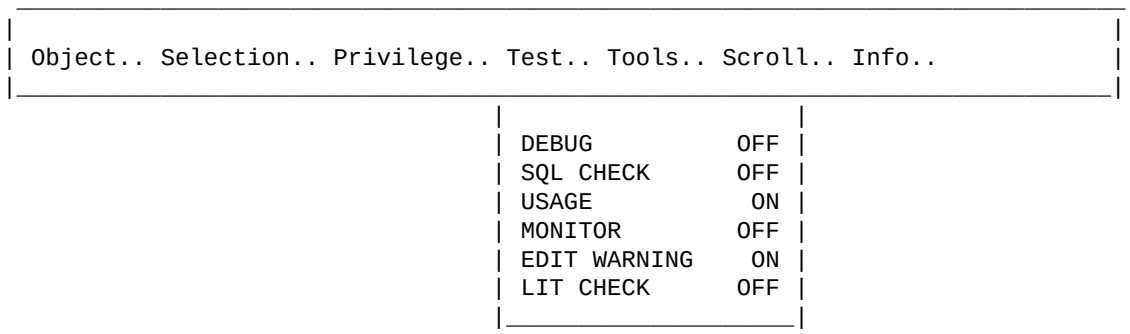
Privileges/Revoke DBPROC

removes the explicit execute privilege for a certain DB Procedure from a particular user or the implicit execute privilege for a certain DB Procedure from all users (PUBLIC).

(Command: ’PREVOKE’)

The ’Test’ Menu Item

After selecting the ’Test’ menu item the following pulldown menu appears:



The function group of the ’TEST’ menu item serves to display and set the test options. Here all settings can be found which are allowed for translation and during the execution of modules.

In the example above the default setting is represented as the user sees it when he has not yet modified any options. All settings remain valid up to the next modification. The setting is changed by positioning the cursor to a test option and pressing the ENTER key. The new setting is displayed in the message line.

Test/DEBUG

The DEBUG option allows modules to be compiled in such a way that they can be processed with the SQL-PL debugger. ’DEBUG on’ has the effect that all modules compiled subsequently can be debugged (state ’DEBUG’ in the module list) and that the debugger is called when starting with RUN. Default is DEBUG off.

(Command: ’DEBUG’)

Test/SQL CHECK

The SQL CHECK option allows the automatic check of the SQL syntax to be enabled or disabled for translation. The SQL CHECK option should always be disabled, when modules are compiled that access tables which are only created at runtime.

(Command: ’SOPT’)

Test/USAGE

The USAGE option serves to enable or disable the maintenance of usage information in the Data Dictionary. When DB Procedures and triggers are compiled, the usage information is maintained even with disabled USAGE option.

(Command: ’USAGE’)

Test/MONITOR

The MONITOR option enables the user to have information about the execution of a program displayed. When the monitor is switched on, the result displayed directly after the RUN execution of a program has terminated.

The following information is provided:

- The total runtime of the program.
- The waiting time for user inputs (think).
- The number of database orders for the program.
- The time required for these (only with MONITOR ON)
- The number of DB orders of the SQL-PL system.
- The time required for these
- The number of CALL and SWITCH calls
- The hit rate in the main memory in per cent
- The number in main memory
- The number of displaced routines and forms

(Command: ’MONITOR’)

Test/EDIT WARNING

With enabled EDIT-WARNING option a message is output when a program is called which has one or more modules in the EDIT state. In such a case the PROD version of these modules is accessed which may lead to unexpected effects. The program is not executed, until the warning has been confirmed.

(Command: ’EWARN’)

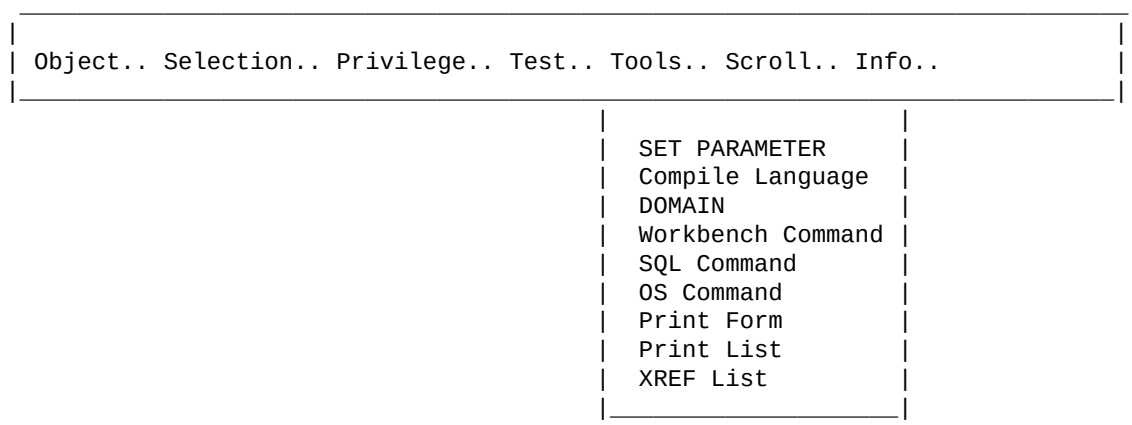
Test/LIT CHECK

If LIT-CHECK is ON, when storing, LITERAL entries are searched and a check is made as to whether these have already been entered in the literal table. If the entry is missing, a form for defining the literal is displayed.

(Command: ’LITERAL’)

The ’Tools’ Menu Item

After selecting the ’Tools’ menu item, the following pulldown menu appears:



Tools/Set Parameter

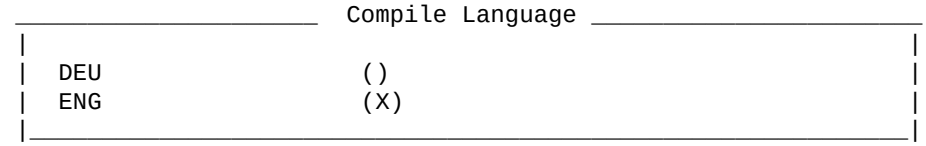
A form is displayed in which the user-specific settings relevant for SQL-PL (language, date representation, ...) can be modified.

A detailed description is contained in Section, "User-specific Set Parameters".

(Command: ’SET’)

Tools/Compile Language

This function can be used to set one or more default languages for the translation of modules with literals. This function can only be effective, if literals have already been defined.



All languages defined for the user are displayed. The languages desired for translation have to be marked with ’X’. The setting remains valid up to the next modification made by means of the function ’Compile Language’ or the LANGUAGE command.

(Command: ’LANGUAGE’)

Tools/Workbench Command

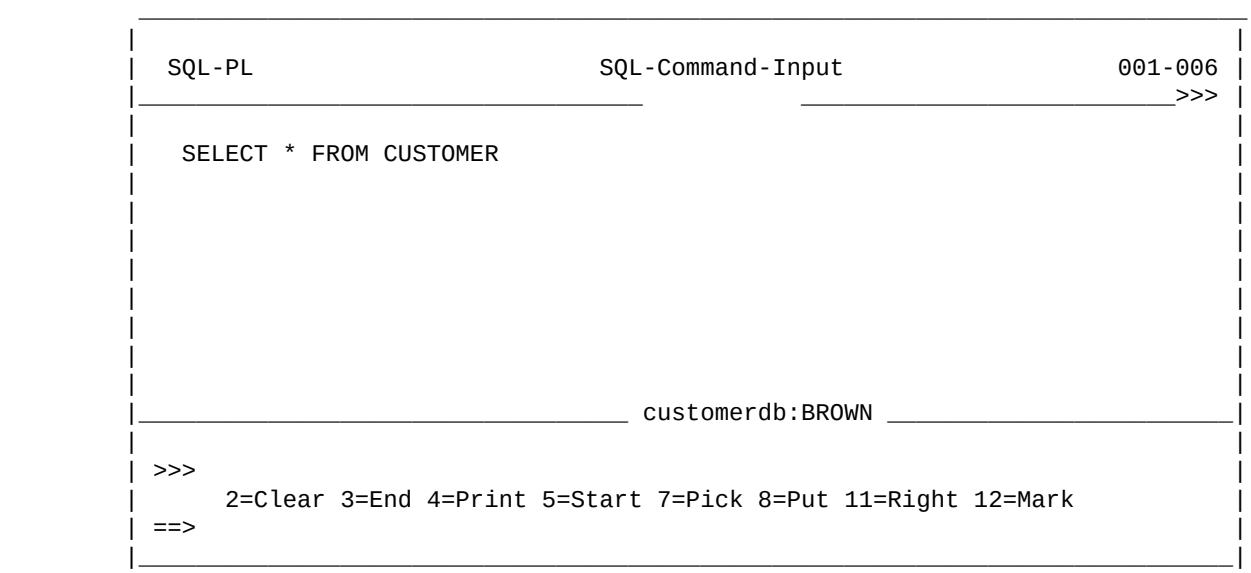
This function opens the command line for the input of workbench commands. Thus it corresponds to the CMD key.

Tools/SQL Command

provides the possibility of issuing database commands out of SQL-PL. After calling SQL a window for entering the DB queries is opened in the lower part of the screen.

(Command: ’SQL’)

Example:



The results of the query are output by means of the REPORT generator.

Tools/OS Command

This function allows to issue an operating system command out of SQL-PL.

(Command: ’EXEC’ or ’!’)

Tools/Print Form’

outputs the visible part of the program list or module list displayed on the screen on the printer specified in Set. Thus it corresponds to the F4 key.

Tools/Print List

outputs the complete content of the program list or module list on the printer defined via Set.

Tools/XREF List

With XREF the following displays are possible:

1. Which global variables are used in which modules of a program?

   The output is ordered according to the module names.

   (Function ’Mod->Var’)

2. As item 1; but the list is ordered according to the variables.

   (Function ’Var->Mod’)

3. Which modules are called by the modules of a program?

   (Function ’Mod->Mod’)

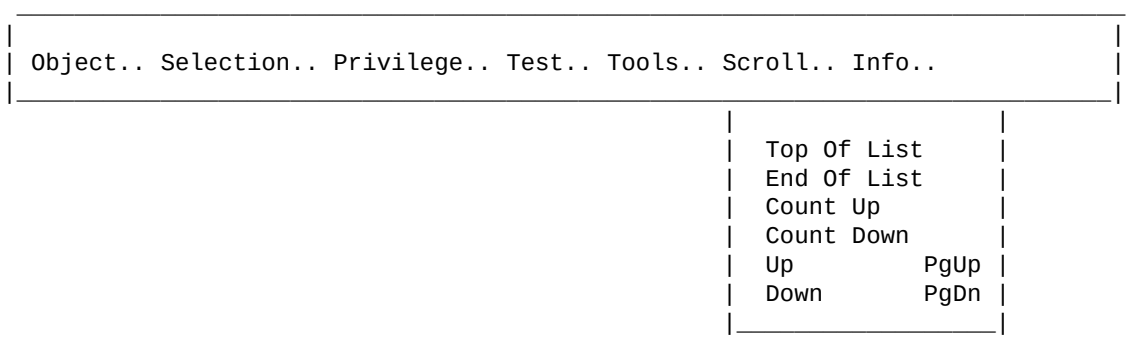
4. As item 3; but the list is ordered according to the called modules.

   (Function ’Mod<-Mod’)

(Command: ’XREF’)

The ’SCROLL’ Menu Item

After selecting the ’Scroll’ menu item the following pulldown menu appears:



Scroll/Top Of List

This function scrolls to the top of the current list.

Scroll/End Of List

This function scrolls to the bottom of the current list.

Scroll/Count Up

This function scrolls the specified number of rows to the top of the list.

Scroll/Count Down

This function scrolls the specified number of rows to the bottom of the list.

Scroll/Up

This function scrolls one page up. Thus it corresponds to the PgUp key.

Scroll/Down

This function scrolls one page down. Thus it corresponds to the PgDn key.

The ’INFO’ Menu Item

After selecting the ’Info’ menu item the following pulldown menu appears:

Info/Help

informs about theworkbench commands, the editor, the SQL-PL language syntax, the SQL statements, the REPORT commands, the debugger, the DB Procedures, and the triggers. This function can also be activated via the F1 key. This help information is provided with numerous examples which can be copied from the help display into the editor by means of the PICK key (GET command).

(Command: ’HELP’)

Info/Time

displays the current date and time format specified in Set.

(Command: ’TIME’)

Info/Workbench Version

displays the version number and the creation date of the workbench, until a key is pressed.

(Command: ’VERSION’)

Commands

The workbench commands can be written into the command line, or they can be specified directly as an argument of the call xpl.

When doing so, commands can be abbreviated as long as they remain unique (see uppercases in the command syntax).

Instead of unique program and module names it is possible to specify search patterns to limit the eligible programs and modles.

The version can be written directly after the module name specification, separated from it by a blank. The result list can be limited by specifying a search argument after the parameter symbol ’-p’.

Examples:

ml customer.* TEST -p FORM	all form modules of the program ’CUSTOMER’ are displayed in the test version.
print c????."start"	only the modules named ’start’ of all programs whose names begin with ’c’ and have a length of five characters are printed out.

Syntax:

```
<cmd> <program-name>.<module-name> <version>
      -p <search argument>
```

Description of the Workbench Commands:

ACREATE Command

With ACREATE, an alias name is created for a DB Procedure. This name is used to identify the DB Procedure from Windows programs that use the ODBC library.

```
Example:    ACR accept FOR DBPROC customer.accept

Syntax:    ACReate <alias name> FOR DBPROC <program>.<module>
```

COPY Command

With COPY the specified program of another user is copied to the own library.

```
Example:     COPY CUSTOMER FROM MILLER PROD

Syntax:      COpy <program> [FROM] <author> [<version>]
```

DEBUG Command

The DEBUG command activates (deactivates) an option which allows modules to be compiled in such a way that they can be processed with the SQL-PL debugger. DEBUG without the parameters ON/OFF displays the current state.

```
Example:    DEB ON

Syntax:     DEBug [ON | OFF]
```

DELETE Command

With DELETE the specified module is deleted (and the whole program, if this is the only module of the program).

```
Example:     DEL customer.reservation

 Syntax:     DELete <program>.<module> [<version>] [-p <search-argument>]
```

DOMAIN Command

Domain is an independent tool for the management of database objects and their relationships (see the "Domain" manual).

```
 Syntax:     DOmain
```

DROP Command

With DROP the whole program is dropped from the user's SQL-PL library. DROP applied to a program of another user has the effect of revoking the execute privilege.

```
Example:    DROP customer

Syntax:     DRop <program> [<version>]
```

EDIT Command

EDIT has the effect that the editor is called to edit either a new module or the specified module.

```
Examples:     EDIT customer.reservation
              EDIT             /* to generate a new module

Syntax:       Edit [<program>.<module>] [<version>]
```

EWARN Command

The EWARN command can be used to enable the option for EDIT-WARNING. Then a message is output, when a program is called and one or more modules of the same program have the EDIT state.

```
Example:     EWARN ON

Syntax:      EWARN [ON | OFF]
```

EXEC Command

EXEC can be used to issue operating system commands.

```
Syntax:      EXEC <command>    /* synchronous call or alternatively
                                  under UNIX!<command>
```

EXIT Command

EXIT terminates the current SQL-PL session. When EXIT is specified in the editor, the current content of the editing form is not saved.

```
 Syntax:     EXIt
```

EXPORT Command

EXPORT writes programs to an operating system file. The modules are separated from each other by ENDMODULE. Granted privileges are exported as workbench commands.

```
Examples:    EXPORT                  *.start all_menus TEST -p EDIT
             EXP                     customer customer.pl APP
```

```
Syntax:     EXPort <programm> [.<modul>] <dateiname>
                              [<version>] [APPEND]
                              [-p <search argument>]
```

GETHIST Command

The GETHIST command cancels the MKPROD command.

```
Example:   GET customer

Syntax:    GEthist <program>
```

GRANT Command

With GRANT ... TO either the explicit execute privilege for a program is granted to a definite user or the implicit execute privilege (PUBLIC) is granted to all SQL-PL users. GRANT without TO specification displays a list of those users who have an execute or copy privilege for the program concerned. GRANT COPY also grantsthe implicit EXECUTE privilege (GRANT EXECUTE).

```
Examples:    GRANT EXECUTE ON customer TO miller
             GRA COP ON customer

Syntax:      Grant Copy | Execute ON <program> TO <user>
             Grant Copy | Execute ON <program> TO PUBLIC
             Grant Copy ON <program>
             Grant Execute ON <program>
```

HELP Command

HELP provides information about the workbench functions, the SQL-PL language syntax, REPORT commands and SQL statements.

```
Example:     HELP

Syntax:      Help
```

IMPORT Command

This command reads in a program from a file generated by means of the EXPORT command. It expects that the individual modules are separated from each other by ENDMODULE. The specified privileges commands are executed.

```
Example:     IMP    customer.fil

Syntax:      Import <filename> [<version>]
```

LANGUAGE Command

The LANGUAGE command can be used to set a language for the translation of modules with literals.

```
Example:     LANG

Syntax:      LAnguage
```

LITERAL Command

LITERAL changes the LIT-CHECK option. If LIT-CHECK is ON, a module is searched for LITERAL entries and a check is made as to whether these are already entered in the literal table. If the entry is missing, a form for defining the literal is displayed. In the editor the LITERAL command can be called without parameters. In this case the current editing form is checked.

```
Example:     LIT CUSTOMER.S*

Syntax:      LITeral [ON | OFF ]
```

MKPROD Command

With the MKPROD command a production version is made from a test version. When doing so, a production version which might already exist is saved as historic version, if ’NOHIST’ has not been specified.

```
Example:     MKP customer

Syntax:      MKprod <program> [NOHIST]
```

MLIST Command

MLIST generates an index of modules for which the user has the execute privilege. It is possible to specify search patterns and search arguments.

```
Examples:    MLIST miller.customer.*
             MLIST c?er.start
             MLIST c*           -p FORM
             MLIST              -p SAVE
             MLIST *.start      -p 02/12/24

Syntax:      MList [<author>.] <program> [.<module>]
                                         [-p <search-argument>]
```

MONITOR Command

The MONITOR command enables the user to have information displayed about the execution of a program. When the monitor is switched on, the result displayed directly after the execution of a program.

```
Examples:     MON on
              MON off

Syntax:       MONitor  [ON] [OFF]
```

Command

The command PCREATE stores a DB Procedure in the database kernel. The state indication in the module list is changed from ’RUN’ to ’->DB’ or ’ +DB’.

```
Example:     PCR customer.insert

Syntax:      PCreate <program>.<module>
```

PDROP Command

The command PDROP cancels the command PCREATE, i.e. the DB Procedure is removed from the database kernel, thus becoming modifiable again.

```
Example:     PDR customer.insert

Syntax:      PDrop <program>.<module>
```

PGRANT Command

PGRANT can be used to grant the execute privilege for a DB proeedure.

```
Example:     PGRANT customer.accept TO PUBLIC

Syntax:      PGRANT <program>.<module> TO <user name>
```

PLIST Command

According to a search pattern, if any, PLIST provides a menu SQL-PL programs which the user is allowed to call.

```
Examples:     PLIST  *.*          displays all programs that can be called.
              PL     C*           displays only those programs of the user
                                  which begin with 'C'

Syntax:       PList  [<benutzer>.] <programm> [<version>]
```

PPRIV Command

PPRIV displays the privileges granted for DB Procedures.

```
Example:     PPRIV customer.accept TO PUBLIC

Syntax:      PPRIV <program>.<module> TO <user name>
```

PREVOKE Command

PREVOKE removes the execute privilege granted for a DB Procedure.

```
Example:     PREVOKE customer.accept TO PUBLIC

Syntax:      PREVOKE <program>.<module> TO <user name>
```

PRINT Command

With PRINT one or more modules or one or more programs are printed out.

```
Examples:    PRINT  customer.*
             PRINT  c?er.start
             PRINT  c*          -p PROC
             PRINT              -p FUNC
             PRINT  *.start     -p =02/12/24

Syntax:     PRINt <program> [.<module>] [<version>] -p
                            [-P <search argument>}
```

PRIVILEGES Command

PRIVILEGES provides an index of the execute privileges that have been granted for all or for a particular program. In this index privileges for own programs can be withdrawn by means of REVOKE or foreign programs be copied or called.

```
Example:     PRIV customer

Syntax:      PRIVileges [ [<user>] [<author>.] <program> ]
```

PSHOW Command

The command PSHOW displays all DB Procedures which the user has stored in the database kernel.

```
Example:     PS

Syntax:      PShow
```

QUIT Command

With QUIT the editor is left without saving the current content of the form. The effects of previous SAVE commands are kept. The QUIT command can only be used in the editor.

```
Syntax:     Quit
```

REVOKE Command

REVOKE ... FROM withdraws either the explicit execute privilege from a particular user or the implicit execute privilege for a definite program from all users (PUBLIC). REVOKE without FROM specification displays a menu of all users who have a privilege for these programs.

```
Examples:     REVOKE COPY ON C* FROM miller
              REVOKE EXECUTE ON Customer FROM PUBLIC

Syntax:       Revoke Copy | Execute ON <program> FROM <user>
              Revoke Copy | Execute ON <program> FROM PUBLIC
              Revoke Copy | Execute ON <program>
```

RUN Command

RUN starts the execution of the specified module.

Parameters can be specified which will be assigned to the formal parameters of the procedure or form. The parameter specification begins with ’-p’. The individual parameters are separated from each other by blanks.

```
Examples:    RUN charles.customer.reservation
             RUN customer.reservation
             R customer
             R customer -p charles miller

Syntax:      Run [<author>.]<program>[.<module>]
                 [-p <parameter>]
```

SAVE Command

With SAVE the currently edited version of the module is saved without checking it for executability. The module obtains the state EDIT (see Section, "The Selection/Show/Module List" Menu Item). The command can only be used within the editor; the editor is not left.

```
Syntax:     SAVE
```

SET Command

The SET command has the effect that a form is displayed in which the user-specific settings relevant for SQL-PL (language, date format, ...) can be modified. If a valid version (TEST, PROD, HIST) is specified after the SET command, the version setting is changed directly without displaying the Set menu.

A detailed description is given in Section, "User-specific Set Parameters".

```
Example:    SET

Syntax:     SET [<version>]
```

SOPT Command

The SOPT command can be used to enable or disable the automatic SQL syntax check.

```
Example:    SOPT ON

Syntax:     SOPT [ON | OFF]
```

SQL Command

SQL provides the possibility of issuing database commands out of SQL-PL. After calling SQL a window opens in the lower half of the screen to enter the DB queries. The results of the query are output by means of REPORT.

```
Syntax:     SQL
```

STORE Command

STORE performs a syntax check for a set of SQL-PL modules and stores them.

In the editor STORE checks the current module for executability. If an error is detected, this is marked on the screen and an error message is output; otherwise the module is stored obtaining the state RUN.

```
Examples:   ST customer.*
            STO c?er.start
            ST k*       -p DEBUG
            STORE       -p SAVE
            ST *.start  -p <02/12/24

Syntax:     STore <program> [.<module>] [<version>] [-p
```

Command

The command TCREATE activates a trigger module. In a dialog box the table name and the column names to which the trigger shall be applied must be specified. After successful activation, the state in the module list is changed from ’RUN’ to ’->DB’.

```
Example:     TCR  customer.insert

Syntax:      TCreate <program>.<module>
```

TDROP Command

The command TDROP cancels the TCREATE command, i.e. the trigger is removed from the kernel and deactivated.

```
Example:     TDR  customer.insert


Syntax:      TDrop <program>.<module>
```

TEST Command

TEST performs a syntax check for the currently edited module and executes it at once. If an error is detected, this is marked and an error message is output. Parameters which are to be passed with the call must be written into the command line. Then TEST is started with the function key. The editor is not left.

All modifications made to database contents during the test run are reset at the end of the run.

The TEST command can only be used within the editor.

```
Syntax:     TEST
```

TIME Command

TIME outputs the current date and time of day in the middle of the screen.

```
Syntax:     TIme
```

TSHOW Command

The command TSHOW displays alle triggers which the user has activated in the database kernel. For the program name and the module name search patterns can be specified as parameters.

```
Examples:    TSH
             TSH c*
             TSH *.insert

Syntax:      TShow <program>.<module>
```

USAGE Command

With the USAGE command the maintenance of the used-relations in the Data Dictionary can be enabled or disabled.

```
Example:     USAGE ON

Syntax:      USAGE [ON | OFF]
```

VALUE Command

VALUE generates an index of all variables of the module displayed in the editor. The index contains the current values from the last TEST execution. The command can only be used in the editor. The display of variables can be restricted by specifying a search argument in the command line.

```
Syntax:      VALUE
```

VERSION Command

VERSION outputs the creation date and time of the current workbench version in the middle of the screen.

```
Syntax:      Version
```

XREF Command

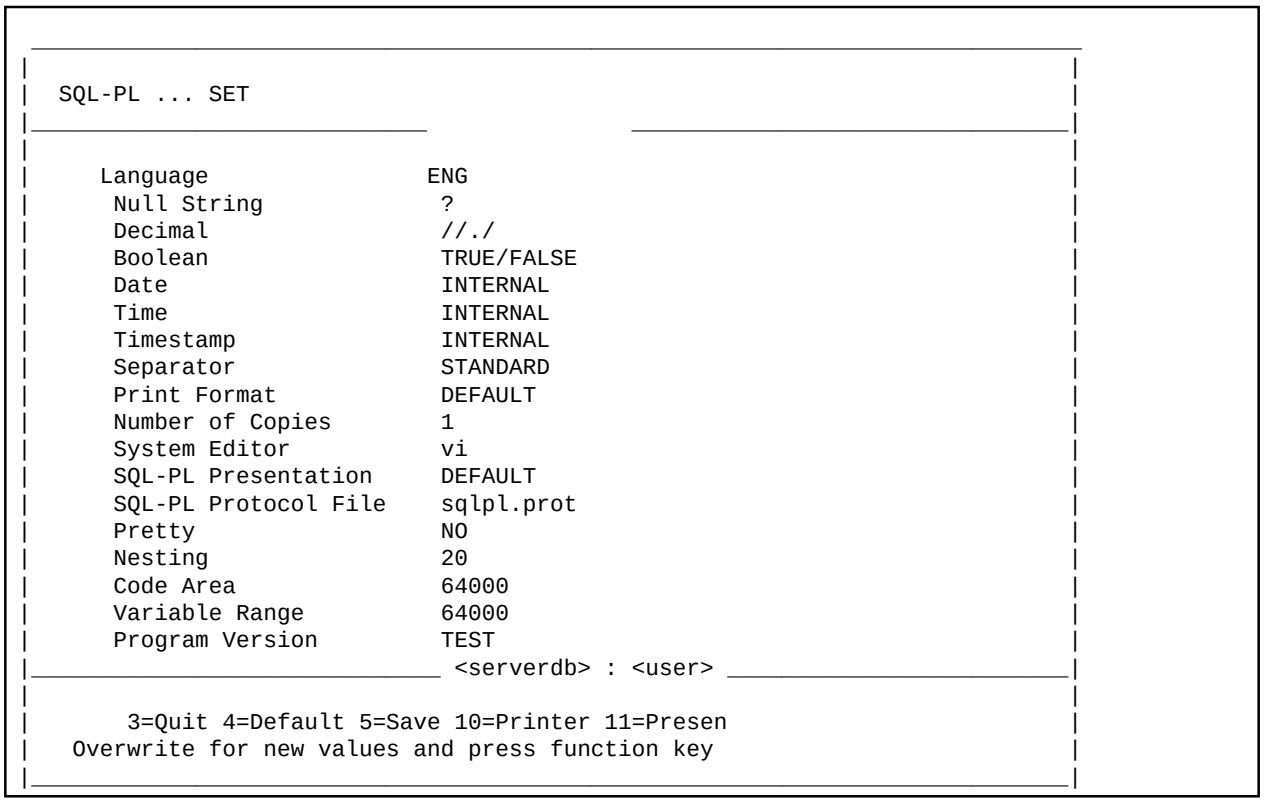
XREF shows how the global variables of a program are used in the modules. This command can also be used to find out which variables are used by a certain module. With a function key, the grouping can be toggled between module or variable.

```
Examples:    XREF customer

Syntax:      XRef<program>
```

User-specific Set Parameters

The SET command provides the user with a form that contains a series of control parameters. SQL-PL produces a default setting for each of these parameters. Every user can modify these settings according to his own requirements. The new values remain valid beyond a session’s end.

After issuing the command SET the following form containing the default settings of the set parameters is displayed:



The displayed values of the Set parameters can be modified by overwriting them. Outside the input fields the display form is write-protected.

The individual Set parameters have the following meanings:

1. Language defines the language for the output of the database and SQL-PL messages: ENG stands for English, DEU for German. A language can only be specified if messages are actually available for it.

2. Null String defines the character string for the representation of NULL values from the database. This string may have a maximum length of 20 characters.

3. Boolean defines the character strings for the representation of BOOLEAN values from the database. The character strings may have a maximum length of 10 characters. In case of <true>/<false>, <true> defines the character string for values that are true, and <false> defines the character string for values that are false.

4. Decimal defines the characters to be used for decimal numbers: in case of /<t>/<d>/, <t> defines the character for the thousands separator and <d> the character for the decimal sign; <t> may be omitted.

5. Date defines the format in which DATE column values are represented in REPORT or the DATE function and accepted in SQL statements.

   The name of a standard format or a user-defined format can be specified. If a standard representation is chosen, this is automatically applied to DATE and TIME parameters. In SQL statements user-defined formats are treated as INTERNAL.

Standard formats are:

```
ISO      which corresponds to  YYYY-MM-DD,
USA      which corresponds to  MM/DD/YYYY,
EUR      which corresponds to  DD.MM.YYYY,
JIS      which corresponds to  YYYY-MM-DD,
INTERNAL which corresponds to  YYYYMMDD
```

Thereby D stands for D(ay), M for M(onth), and Y for Y(ear).

If three positions are specified for the month, then the name of the month will be output in its common abbreviation (Oct for October). User-defined formats need not contain each of the three symbols for the date portions.

6. Time defines the format in which TIME column values are represented in Report or the TIME function and accepted in SQL statements.

```
ISO      which corresponds to  HH.MM.SS,
USA      which corresponds to  HH:MM AM (PM),
EUR      which corresponds to  HH.MM.SS,
JIS      which corresponds to  HH:MM:SS,
INTERNAL which corresponds to  HHHHMMSS.
```

Thereby H stands for H(our), M for M(inute), and S for S(econd).

7. Timestamp defines the format in which TIMESTAMP column values are to be input and output. This format is valid for both Query commands and SQL statements.

Standard formats are:

```
ISO       which corresponds to YYYY-MM-DD-HH.MM.SS.NNNNNN,
USA       which corresponds to ISO,
EUR       which corresponds to ISO,
JIS       which corresponds to ISO,
INTERNAL  which corresponds to YYYYMMDDHHMMSSNNNNNN
```

where N stands for milliseconds and microseconds; the other letters have the same meaning as explained for date and time.

8. Separator defines the character string which is used to separate result table columns from each other. If this string is to contain blanks at its end, it has to be enclosed in single quotes. The string may have a maximum length of 20 characters. The default value ’STANDARD’ corresponds to the string ’|’ with the special feature that on the screen the column separations appear as a continuous line, if the monitor is capable of representing semigraphics.

9. Print Format defines the format of the printout. Here the user can specify either a print format provided with the installation or a user-defined print format. Up to eight print formats can be defined - see the description of the PRINTER Key at the end of this section.

10. Number of Copies defines how many copies are to be made on printing.

11. For System Editor the user can define an editor of his selection. This editor will be called with the command SYSED.

12. SQL-PL Presentation allows a user to specify a presentation for his personal usage in SQL-PL presentation name designates a certain setting of screen colors and attributes. This setting can be modified enabling the user to adapt the aspect of SQL-PL according to his own liking.

    With the installation various presentations are provided which are immediately available to every user. These presentations can be paged through or redefined. Up to eight presentations can be defined - see the description of the PRESEN Key at the end of this section.

13. SQL-PL Protocol File allows the user to choose the name of the protocol file.

14. With PRETTY it is determined whether the sequence of statements should be made more attractive by means of automatic indentations and capitalization of main entries, when storing a module.

15. With the parameter Nesting the maximum depth of the call hierarchy (CALL or SWITCHCALL) is determined.

16. With the parameter Code Area the size of the memory area is set in which the interpreter holds the program to be executed. A changed parameter only becomes effective in the next session.

17. With the parameter Variable Range the maximum memory size for the variables available at one time is set. A changed parameter only becomes effective in the next session.

18. With the parameter Program Version SQL-PL is told with which version of the program the user wants to work now.

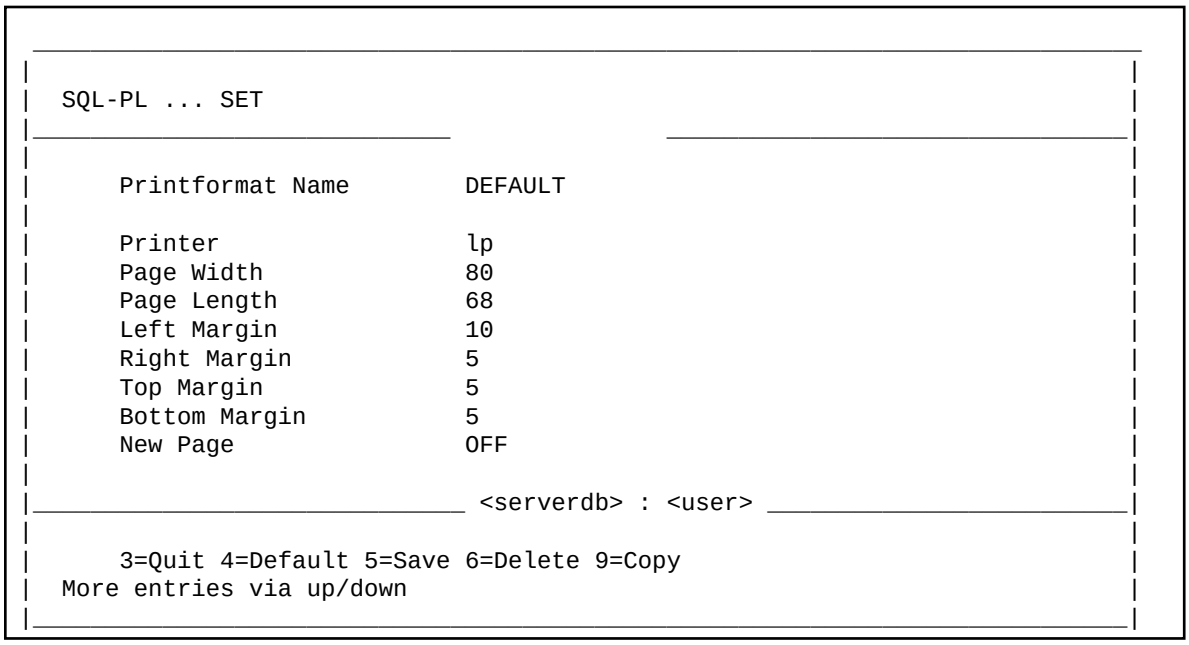
The Save key accepts the newly entered values and leaves the Set mode.

The Quit key leaves the Set mode without having the modifications come into effect.

The Default key sets all displayed parameters to predefined default values. These must not be identical with the values displayed after the first call of SQL-PL, because the system administrator is allowed to choose other default settings which will be displayed for any users who have not yet defined a parameter set of their own.

The keys Printer and Presen branch to further forms and are described in the following.

The Printer key switches from Set mode to a menu where the user can define the print formats.



At first the currently set print format is displayed. If more formats are defined, a message informs the user about it. He can switch from one format to the other by means of the scroll keys.

The settings can be modified by overwriting the entries. The following settings can be defined in such a format:

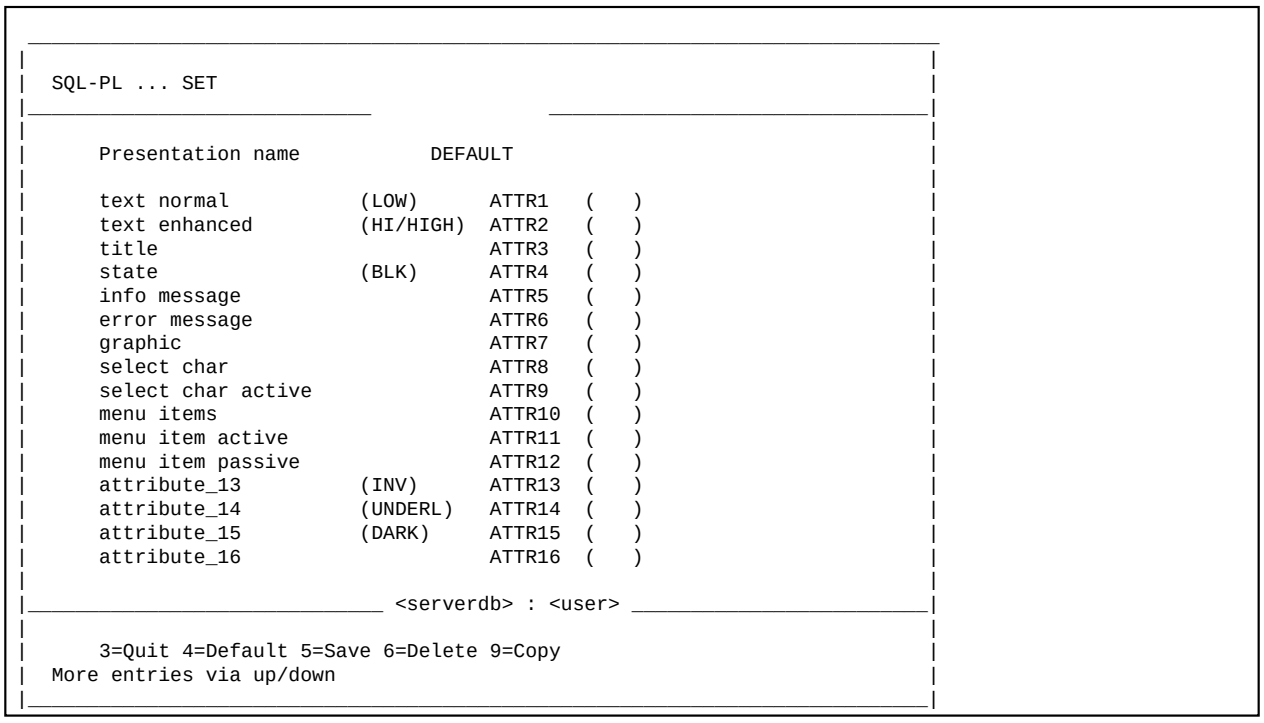
1. For Printformat Name that name is displayed which was given to the defined format.
2. Printer specifies the desired printer. This specification has to be made according to the installation.
3. Page Width defines the width of a print page. The value may be 254 at the most.
4. Page Length defines the complete length of a print page in number of lines.
5. Left and Right Margin define the number of blanks to be output to the left and to the right of the text.
6. Top and Bottom Margin define the number of blank lines to be output above and below the text.
7. New Page defines whether (ON) or not (OFF) a form feed is to be performed for each separate print job.

The keys Quit, Default, and Save have the same meanings as in the superior Set form. If the user returns to the first form by means of Save, the last displayed format becomes the current format, i.e. its name is displayed for Print Format.

Defined formats can be deleted by means of the Delete key.

The Copy key generates a new entry in which the format name is not yet assigned. The other parameters are taken over from the setting previously displayed and can be modified at will.

The resen(tation) key switches from Set mode to a menu where the user can define the presentations.



At first the currently set presentation is displayed. If more presentations are defined, a message informs the user about it. He can switch from one presentation to the other by means of the scroll keys.

In such a presentation the different physical properties are assigned to the sixteen logical attribute names. Each logical attribute name (ATTR1 to ATTR16) is depicted in the menu together with the attributes and colors assigned to it.

It depends on the used installation and system, what kinds of representation and colorings are available. If colors cannot be set, the keys Backgr and Foregr are not displayed.

To change such an assignment, mark one or more attributes with an "x" and press the keys Attribute, Foregr, or Backgr. Popup menus appear where the desired settings for the coloring and kind of representation can be chosen by checking them with an "x".

The toggle switch Mark has the effect that all attributes are marked with an "x". If all attributes are already marked with an "x", this key removes them instead.

The keys Quit, Default, Save, Delete, and Copy have the same functions as in the other Set forms.

Each of the first twelve attributes is employed by SQL-PL for a definite purpose which is identified by the first column of the menu. Of these fixed attributes, the first seven are used by all Adabas components in the same way.

The attributes from ’select char’ to ’menu item passive’ serve the presentation of the pulldown menus in SQL-PL.

It is recommended that the attribute ’select char’ and ’menu items’ be defined with the same background color and different foreground colors. The same holds for the pair of attributes ’select active’ and ’menu item active’. The attribute for ’menu item passive’ is used to depict a menu item that cannot yet be selected. For this reason, this attribute should be defined in a more reserved way than others.

When defining attribute characters in forms, the designations LOW, HIGH, BLK, INV, UNDERL, and DARK can still be used. The presentation menu shows to which of the logical attributes these designated attributes are assigned.

The SQL-PL Language

This chapter covers the following topics:

- Basic Elements
- Variable Declaration
- Basic Statements
- Calling Procedures, Forms, and Functions
- Calling Stored Procedures
- Embedding SQL
- Query Call
- Editor Call
- Line-oriented Input and Output
- Processing Files
- Calling Operating System Commands
- SQL-PL System Functions
- System or $ Variables
- Module Options

Basic Elements

This section covers the following topics:

- Comments
- Names
- Literals
- Variables
- Arithmetic Expressions
- String Expressions
- Boolean Expressions

Comments

Comments can be inserted in the module text at any place. They start with /* and end with the line in which they occur.

Names

The following applies to the names of programs, modules, variables, and all database objects (e.g. names of users, databases, tables or result tables):

The first character can be a letter or one of the characters ’$’, ’#’ or ’@’. After that, letters, digits and ’_’,’$’,’#’,’@’ can follow. No difference is made between upper and lower cases.

The name of a program, a module or a database object can also be composed of arbitrary characters in which upper and lower cases are distinguished when the name is enclosed in double quotes.

Note:
On keyboards without the ’@’ character, the section sign may be used instead.

For all names, the first 18 characters are significant.

```
valid names    : CNO, #k  , N_ALT, k2

invalid names  : 1a, 13, _name

names valid for database objects (table, column),
invalid for programs, modules, variables or skip labels:
                                    "1a", "_)(*&’", "customer_list"
```

Syntax:

```
<name>       ::= <firstchar>  [<name char> ... ]

<firstchar>  ::= <letter> | # | @ | $

<name char>  ::= <letter> | <digit> | _ | # | @ | $

<letter>     ::= a | .. | z | A |..| Z

<digit>      ::= 0 | .. | 9
```

Literals

A literal is either a string or a numeric value. A string can also be noted in hexadecimal digits. SQL-PL also provides the option of defining language-dependent strings.

Syntax:

```
<literal>  ::=  <numeric literal>
             |  <string literal>
             |  <hex literal>
             |  <langdep literal>
             |  <key literal>
```

This section covers the following topics:

- Numbers
- Character Strings
- Hexadecimal Strings
- Language-dependent Literals
- Key Literals

Numbers

Numbers can be specified with sign, decimal point, and exponent.

Examples:

```
100, -17.25, +5E-3, -0.123e12
```

It is to be noted that there are several ways of writing a number because of the freedom of types. For example, all the following values represent the numeric value 1:

```
 '01' , '1', 1.0, '1.'
```

Syntax:

```
<numeric literal>  ::=  <numeric>

<numeric>          ::=  <unsigned integer>
                      |  <fixed point>
                      |  <floating point>

<unsigned integer> ::=  <digit>  [<unsigned integer>]
                        maximum of 18 digits

 <digit>           ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

 <fixed point>     ::=  <unsigned integer>  [.<unsigned integer>]

 <floating point>  ::=  <fixed point> E <sign> <digit> [<digit>]

 <sign>            ::=  + | -
```

Character Strings

Character strings are enclosed in single quotation marks. They must contain at least one character. Their maximum length is only determined by the memory available to the SQL-PL interpreter. When used as parameters in SQL statements, their current length must not exceed the defined length. The single quotation mark is doubled if it is to be a character in the string.

Examples:

```
 'This is a string '
 'Say ''Yes'' '
```

Syntax:

```
<string literal> ::=  '<any char>...'

<any char>       ::=  any character on the keyboard
```

Hexadecimal Strings

Hexadecimal strings are identified by a leading 'X'.

Examples:

```
x'000001'
IF x'20' = ' ' THEN WRITE 'ASCII Code';
```

Syntax:

```
<hex literal>    ::=  X'<hex digit seq>'  |  x'<hex digit seq>'

<hex digit seq>  ::=  <hex digit><hex digit>[<hex digit seq>]

<hex digit>      ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
                    |  A | B | C | D | E | F | a | b | c | d | e | f
```

Language-dependent Literals

To build a program for various languages, language-dependent literals can be used. They are identified by a leading '!' and can be used like strings. At compile time, the name after '!' is replaced by the text defined in the table SYSLITERAL (see Section, "Language-dependentPrograms").

Examples:

```
!customernumber(s), !title(m)
```

Syntax:

```
 <langdep literal>  ::=  !<name> [ (<literal size>) ]

 <literal size>     ::=  S | M | L | XL
```

Key Literals

To find out the last key used, SQL-PL provides key literals. Basically, the key literals are named constants. Besides the value 'UNKNOWN', the system variable $KEY can only have one of the key literal values. A useful usage of key literals is only possible in interactive modules.

Further detailed descriptions are contained in Sections, "Key Activation (ACCEPT)" and, "Overriding Keys (ACCEPT)".

Example :

```
CASE $KEY OF
  ENTER, F5: ....
  F3: RETURN;
END;
```

Syntax:

```
<key literal>  ::=  ENTER
                  |  <basic key>
                  |  <additional hardkey>

  <basic key>  ::=  F1  | F2  | F3  | F4  | F5  | F6  | F7  | F8  |  F9
                  |  F10 | F11 | F12 | HELP | UP | DOWN

  <additional hardkey> ::= CMDKEY | ENDKEY | UPKEY | DOWNKEY | RIGHTKEY | LEFTKEY
```

Variables

An SQL-PL variable is either a simple variable (scalar), i.e. it contains a single value at any point in time, or it is a vector , i.e. it contains several values which can be accessed by means of an integer index value. The values of these variables are either strings or numbers and they can also be undefined (the NULL value). Before the first assignment of a value, every variable and every vector element is undefined.

The decision whether a scalar or a vector is involved is taken according to the kind of access when using the variable for the first time.

Variables can assume values of any type. These values are implicitly converted, if this is necessary. If this is not possible, the result is the NULL value. For screen output numbers are typically converted into strings, and for arithmetic expressions strings are typically converted into numbers. The NULL value cannot be converted into a number.

Examples:

```
a     :=  '01';
b(1)  :=  '01';
c     :=  a + b;   /* c equals 2
d(c)  :=  a & b;   /* d(2) equals '0101'
e     :=  NULL;
f     :=  a + e;   /* f equals NULL, since e cannot be
                   /* converted into a number
```

SQL-PL variables are distinguished by their scope of validity.

Global variables

are available in all procedures and forms of a program.

Within a program, the same name of a global variable designates precisely the same variable. Global variables cannot be used in stored procedures and functions.

Local-dynamic variables

are only known to the module in which they are used. Local variables are declared with the VAR statement following the module header. All parameters are implicitly declared as local-dynamic. In different modules of a program the same name of a local-dynamic variable designates different variables.

When a module is left, the values of the local-dynamic variables are deleted. When the module is recalled, they are undefined again.

Local-static variables

only differ from local-dynamic variables by the fact that their values are kept even after leaving the module, thus being available again when the module is recalled. These values are only deleted when the program is terminated. Local-static variables are also declared with the VAR statement following the module header. Local-static variables cannot be used in stored procedures.

In different modules of a program the same name of a local-static variable designates different variables. The value of a local-static variable can only be retrieved in the module in which it has been defined.

Local-static variables in functions maintain their values until the start program has been terminated. If a program system is constructed in which a start program branches to several subprograms which all use the same functions, then these subprograms address all the same local-static variables via functions.

If the variables are not explicitly defined by means of the VAR statement (see Section, "VariableDeclaration"), the naming conventions from earlier versions apply. A variable is declared as local dynamic, when it is used for the first time and its name starts with an ’@’ sign or when the variable is a parameter. When its name starts with ’@@’, the variable is declared as local static. Variables whose names do not start with an ’@’ sign are declared as global variables.

Example:

```
PROC example.auto_declare PARMS ( a, @b(), @@c )

  /* a, @@c are local-dynamic scalars
  /* @b is a local-dynamic vector

x(@y) := @@z;

/* x    is a global vector
/* @y   is a local-dynamic scalar
/* @@z  is a local-static scalar
```

The variables MESSAGE and ERROR

are special variables because they exist only once for the entire application. They can be assigned in any module and are always automatically output by the next form if they are not explicitly initialized with NULL.

The variable MESSAGE serves to display information. The ERROR variable serves to display error messages. Both variables are displayed in the same line of the screen, but with different screen attributes (see S menu, PRESENTATION menu, info message and error message in Section, "User-specificSetParameters"). If both variables have a value, only the ERROR variable is output.

The position of the system line for displaying the MESSAGE and ERROR variables is determined by the form interpreter if it has not been explicitly defined as a form output field of the variable MESSAGE (<MESSAGE).

The variables MESSAGE and ERROR are immediately reset to NULL after they have been displayed in a form.

The values of the MESSAGE and ERROR variables are kept beyond a program change made by means of SWITCH or SWITCHCALL (see Section, "Form Fields and Messages (MESSAGE, ERROR)").

Syntax:

```
<variable>      ::=  <variable name> [ (<expr>) ]
                   |  MESSAGE
                   |  ERROR

<vector slice>  ::=  <variable name> (<expr> .. <expr>)

<variable name> ::=  <name>
```

Arithmetic Expressions

Arithmetic expressions can be formed with the operators +, -, *, /, MOD, and DIV. The four last operators are stronger binding than the first two. Any processing sequence can be enforced by parentheses.

```
account := account + account * interest/100;
value := -(x+y) / ( 3.14*(a MOD b) );
```

An arithmetic expression usually provides a numeric value. It results in the NULL value , however, if

- one of its operands results in the NULL value.
- division by 0 is attempted.
- the valid range of values is exceeded.
- a literal or a variable value cannot be interpreted as a number.

-

-

-

-

Syntax:

```
<expr>       ::=  <num expr> | <str expr>

<num expr>   ::=  [<sign>]  <term>  [<term list>]

<term list>  ::=  <sign> <term> [<term list>]

<term>       ::=  <factor>  [<mult op> <term>]

<mult op>    ::=  * | / | MOD | DIV

<factor>     ::=  <numeric>
                |  <variable>
                |  (<expr>)
                |  <arith function>
                |  <function call>
                |  <dollar numeric variable>

<arith function> ::=  see "Arithmetic Functions" (5.12.1)
```

```
<function call>  ::=  %<function name>

<dollar numeric variable> ::=  see "System or $ Variables (5.13)
```

String Expressions

String expressions are formed by means of the concatenation sign ’&’ and a number of string functions. If a string has the NULL value, it is treated like a string of length 0 in the concatenation operation.

The functions that can be used here are described in detail in Section, "SQL-PL System Functions".

Syntax:

```
<str expr>  ::=  <basic str>  [& <expr>]

<basic str>  ::=  <value spec>
                |  <factor>
                |  <string function>
                |  <strpos function>
                |  <date function>
                |  <set function>

 <value spec> ::=  <literal>
                 |  <repeat string>
                 |  BLANK
                 |  <dollar numeric variable>
                 |  <dollar string variable>

<repeat string>  ::= <repeat char> (<expr>)

<repeat char> ::=  ’<any char>’ | BLANK
```

Boolean Expressions

A Boolean expression describes a condition which either is true or is not true at the time of evaluation. The flow of a module can be dynamically controlled by means of Boolean expressions in IF, REPEAT, and WHILE statements.

A frequent use is the comparison between two arithmetic expressions. If both expressions result in the NULL value, they are regarded as unequal. With the predicate IS NULL it can be checked whether a variable has been set to NULL.

```
cname  := NULL;             cname = cvname is not true
                    -->

cvname  :=  NULL;           cname IS NULL is true
```

In logical expressions, the NULL value is always interpreted as ’false’ and every other value as ’true’.

The value of a Boolean expression cannot be assigned directly to a variable. And a variable cannot be evaluated as a Boolean expression.

Hint for the Embedding of SQL and Boolean Values:

In SQL the data type Boolean is defined with the values NULL, FALSE, and TRUE. When assigning these values to SQL-PL host variables, NULL and FALSE are taken as NULL, and the current TRUE value of the column is taken. SQL-PL host variables with the NULL value, on the other hand, are interpreted as FALSE, and with a value unequal to NULL they are interpreted as TRUE.

The following examples show how simple conditions can be formulated in an SQL-PL module.

```
name = 'Miller'

account >=  100

account IS FIXED (4,2)

name LIKE 'S*'

today LIKE '__.__.9$'

account LIKE '\**' ESCAPE '\'

city IS ALPHA

cno IS NOT NULL

firstname IS BLANK OR IS ALPHA

input IS DATE (yyyymmdd)

input IS TIME

title IN ('Mrs', 'Mr', 'Company')

account BETWEEN -1000 AND +1000

name IS MODIFIED

FORM IS MODIFIED
```

These simple conditions can be combined with the operators AND, OR, and NOT to form more complex Boolean expressions. Explicit parentheses enforce the desired sequence of evaluation even in this case.

For LIKE, all facilities are available that are also known to Adabas in the LIKE predicate (see "Reference" manual).

IS ALPHA checks whether the expression only consists of letters and blanks.

The predicate IS DATE (IS TIME) checks whether the variable content is a date or a time in the specified format. If no format has been specified, a check is made for agreement with the format set in the Set parameters.

The predicate IS MODIFIED allows a request to be made whether a form input field has been changed by the user the last time the form was called. It can be specified that, for example, database insertions are only to be made when this request produces a particular result. Outside of a form, the predicate IS MODIFIED can only be applied to global variables of the form.

In addition, the predicate FORM IS MODIFIED can be used to find out in an easy way whether an input field of the form has been changed by the final user.

Predicates ALL, ANY, ONE

The predicates ALL, ANY, and ONE facilitate the Boolean expressions with regard to vector slices. The predicates EACH and SOME are synonyms for ALL and ANY resp.

Examples:

```
ALL cno(1..20) IS BETWEEN 1000 AND 9999

    the condition is true when   e v e r y
    vector component cno(1) to cno(20)
    satisfies the condition IS BETWEEN ..

ANY cno(1..20) IS NUMERIC

    the condition is true when
    a t   l e a s t   o n e   of the
    vector components satisfies the condition IS NUMERIC

ONE cno(1..20) IN (NULL,1..999)
    the condition is true when
    p r e c i s e l y   o n e   of the
    vector components satisfies the condition IN (NULL,1..999)
```

Syntax:

```
<boolean term>  ::=  (<boolean expr>)
                  |  <expr comparison>
                  |  <expr> <check cond>
                  |  ALL <vector slice> <check cond>
                  |  EACH <vector slice> <check cond>
                  |  ANY <vector slice> <check cond>
                  |  SOME <vector slice> <check cond>
                  |  ONE <vector slice> <check cond>
                  |  EOF (<fileid>)  see "Processing Files" (5.10)
                  |  $SQLWARN [(<expr>)]  see "System or $ Variables" (5.13)
                  |  FORM IS MODIFIED

<boolean expr>  ::= [NOT] <boolean term> [<logic op> <boolean expr>]

<expr comparison>  ::=  <expr> <comp op> <expr>

<check cond>  ::=  <simple cond> [<logic op> <check cond>]

<simple cond>  ::=  <is cond>
                 |  [NOT] <in cond>
                 |  [NOT] <between cond>
                 |  [NOT] <like cond>

<is cond>  ::=  IS [NOT]  ALPHA
             |  IS [NOT]  NUMERIC
             |  IS [NOT]  BLANK
             |  IS [NOT]  NULL
             |  IS [NOT]  MODIFIED
             |  IS [NOT]  FIXED (<expr>,<expr>)
             |  IS [NOT]  DATE [ (<date mask>) ]
             |  IS [NOT]  TIME [ (<time mask>) ]

<in cond>   ::=  IN (<value spec list>)

<value spec list>  ::=  <value spec> [..<value spec>] [,<value spec list>]
```

```
                    |   NULL  [,<value spec list> ]

<value spec>  ::=  <literal>
                |  <repeat string>
                |  BLANK
                |  <dollar numeric variable>
                |  <dollar string variable>

<between cond> ::=  BETWEEN <expr> AND <expr>

<like cond>  ::=  LIKE <expr> [ ESCAPE <expr> ]
```

Variable Declaration

The VAR statement defines module-local variables in an SQL-PL module. Without an explicit declaration of the variables, the local variables (dynamic as well as static) are subject to the naming conventions familiar from earlier versions.

Examples: Declaration of local-dynamic variables

```
PROC customer.card

VAR
    name, firstname();
...
```

Examples: Declaration of local-static variables

```
FUNCTION number.current PARMS ( op, no )

VAR STATIC
    curr_number;

IF op = 'PUT'
THEN curr_number := no;
RETURN (curr_number);
```

The VAR statement immediately follows the module header. A maximum of two VAR statements may be formulated to define local-dynamic variables as well as local-static variables.

Syntax:

```
 <var section>  ::=  VAR <var decl>,...;
                     [ VAR STATIC <var decl>,...; ]

<var decl> ::=  <varname> [<array spec>]

<array spec>  ::=  ()
```

Basic Statements

Syntax:

```
  <lab stmt list>  ::=  [<label>]  <compound>  [<lab stmt list>]


 <compound>  ::=  BEGIN <stmt>;... END | <stmt>
```

```
<stmt>  ::=  <assign stmt>
           |  <vector assign stmt>
           |  <if stmt>
           |  <case stmt>
           |  <repeat stmt>
           |  <while stmt>
           |  <for stmt>
           |  <skip stmt>
           |  <return stmt>
           |  <stop stmt>
           |  <vectsort stmt>
           |  <switchcall stmt>
           |  <proc call>
           |  <form call>
           |  <switch stmt>
           |  <function call>
           |  <dbproc call>
           |  <sql stmt>
           |  <connect stmt>
           |  <release stmt>
           |  <query stmt>
           |  <report stmt>
           |  <edit call>
           |  <write stmt>
           |  <read stmt>
           |  <open file stmt>
           |  <write file stmt>
           |  <read file stmt>
           |  <close file stmt>
           |  <exec command>
           |  <set stmt>
```

This section covers the following topics:

- Value Assignments
- The IF Statement
- The CASE Statement
- The REPEAT Statement
- The WHILE Statement
- The FOR Statement
- The SKIP Statement
- The RETURN Statement
- The STOP Statement
- The Statements LTSORT and GTSORT

Value Assignments

Variables, vector components and vector slices are assigned values in the way described in Section, "Variables". The variables/vectors can be defined globally, local-dynamically or local-statically.

The value to be assigned is placed to the right of ':=' and is defined by an arbitrary expression (see further examples in Capter, "ArithmeticExpressions").

Examples:

```
name := NULL;
name := UPPER ( name );

address (1..10) := NULL;
address (2..3)  := city (1..2);
```

Syntax:

```
<assign stmt>  ::=  <variable> := <assign expr>

<assign expr>  ::=  <expr> | NULL

<vector assign stmt>  ::=  <vector slice> := <assign expr>
                        |  <vector slice> := <vector slice>
```

The IF Statement

The IF statement first evaluates the logical condition. If the specified condition is satisfied, the statement defined in the THEN branch is executed. Otherwise the statement in the ELSE branch is executed, if any.

Example:

```
IF  $RC <> 0
THEN
     ERROR   := $RT
ELSE
     MESSAGE := 'Proceed with ENTER';
```

Syntax:

```
<if stmt>  ::=  IF <boolean expr>
                THEN <compound>
                [ELSE <compound>]
```

The CASE Statement

The CASE statement can be used to select a statement depending on the current value specified expression. The statement specified by the corresponding selector value is executed. After that, the execution is continued with the next statement after the CASE statement.

As selector value, any expression, including an interval, enumeration or NULL value may be specified. Since the value of the selector can only be determined at runtime, no check is made whether a selector value is defined more than once. If this is the case, the statement after the first occurrence of the selector value is executed.

As an option, an OTHERWISE branch can be defined which is chosen when the current value of the expression does not occur as selector.

Example :

```
CASE $KEY  OF
   ENTER, F5  : CALL PROC start;
   F6         : CASE  selection OF
                1,3..4  :  CALL PROC selection_1;
                2,6,8   :  CALL PROC selection_2;
                9..12   :  CALL PROC selection_3;
                END
   OTHERWISE  : RETURN;
 END;
```

Syntax:

```
<case stmt>  ::=  CASE <expr> OF
                            <case list>
                            [OTHERWISE <compound>]
                  END

 <case list>  ::= <value spec list> : <compound> [;<case list>] ;

 <value spec list>  ::=  <value spec> [..<value spec>] [,<value spec list>]
                       |  NULL  [,<value spec list> ]

 <value spec>  ::=  see "Boolean Expressions" (5.1.7)
```

The REPEAT Statement

The REPEAT statement serves to repeatedly execute a statement. The statement is repeated until the specified condition is satisfied. In particular, the statement is executed once before the first check of the condition.

Example :

```
REPEAT
   statement
UNTIL condition;
```

Syntax:

```
 <repeat stmt> ::= REPEAT <stmt>;... UNTIL boolean expr>
```

The WHILE Statement

The WHILE statement allows the conditional repetition of statements. As long as the specified condition is satisfied, the statement is executed. In particular, the condition is checked before the statement is executed for the first time. If the condition is not satisfied, the statement will not be executed at all.

Example :

```
 WHILE condition DO
 statement;
```

Syntax:

```
<while stmt> ::= WHILE <boolean expr> DO <compound>
```

The FOR Statement

The FOR statement executes statements in a loop. The number of times the loop is run through is determined by an initial and a final value. Before the processing loop is executed for the first time, the control variable specified behind the keyword FOR obtains the value specified behind ':=' as initial value. Each time the loop is run through, the value of the control variable increases by 1. The processing loop is terminated as soon as the value of the control variable has reached the final value behind the keyword TO and the statements have been executed. After termination of the loop, the control variable has the final value.

The processing loop is not executed if the initial value is greater than the final value.

When DOWNTO is used instead of TO, the initial value must not be smaller than the final value in order that the processing loop is executed. Correspondingly, the control variable is decreased by 1 for each loop.

Examples:

```
FOR indexvariable := value1 TO value2 DO
statement;

FOR indexvariable := value1 DOWNTO value2 DO
  statement;
```

Syntax:

```
<for stmt>  ::=  FOR <variable> := <expr> <direction> <expr>
                 DO <compound>

<direction>  ::=  TO | DOWNTO
```

The SKIP Statement

SKIP is for dealing with exceptional situations. Skipping can only be done in a forward direction and skip labels are only valid outside control structures. The names of skip labels are formed according to the rules given above.

Example:

```
SKIP label;
...
label : statement;
```

Syntax:

```
<skip stmt> ::= SKIP <name>
```

The RETURN Statement

The RETURN statement terminates the execution of the current routine at once.

Example:

```
IF $KEY = 'F3'
THEN RETURN;
statement;
```

With RETURN(result) the result of the function must be returned from an SQL-PL function or DB function. The value of the function is always NULL without a corresponding expression of the result.

Syntax:

```
 <return stmt>  ::=  RETURN
                   |  RETURN (<expr>)   <-- only in SQL-PL functions
```

The STOP Statement

With the STOP statement it is possible to immediately terminate a stored procedure or an SQL-PL program from any call nesting. A return code as well as a return text can be returned to the calling environment.

Example:

```
SQL ( ... );
IF  $RC < 0
THEN STOP ($RC, $RT)
ELSE STOP (0, 'everything ok');
```

Processing the STOP statement with stored procedures results in the failure of the stored procedure call. The first parameter of the STOP statement is returned to the calling program (precompiled program, SQL-PL, ODBC, JDBC, ...) as SQL return code, the second parameter is returned as SQL return text of the call. If the first parameter cannot be interpreted as a number between –32767 and 32768 or a value predefined by Adabas is selected (especially 0), the stored procedure fails with the runtime error code -503 INVALID STOP CODE IN DB PROCEDURE/TRIGGER. It is therefore recommended to choose the error numbers for stored procedures from the ranges of numbers [-29.999,..29.000] and [29.000,..,-29.999].

Stored procedures terminated without processing a STOP statement or without a runtime error are assumed to be executed successfully; they produce the return code 0 although SQL statements used therein have failed and have produced a return code other than 0. Possible errors should therefore be treated within the stored procedure and, if necessary, be returned with the STOP statement to the calling environment.

DB Procedure

If a DB Procedure fails, all SQL statements within this DB Procedure are reset.

Trigger

If a trigger fails, all SQL statements within this trigger are reset. The calling SQL statement also fails; it is reset as well.

DB Function

If a DB function fails, the embedding SQL statement also fails.

SQL-PL Program

If an SQL-PL procedure or a form is called directly from the operating system (see the "User Manual Unix" or "User Manual Windows"), then control is returned to the operating system by the STOP statement. In this case the absolute value of the return code is provided, if it is less than 126. The return code 127 is returned, if the absolute value of the return code is greater than or equal to 127. In this case it must be taken into account that SQL-PL already uses the error numbers 1 to 8 as return code (see the "User Manual Unix" or "User Manual Windows").

Syntax:

```
 <stop stmt> ::= STOP [(<expr> [<expr>] )]
```

The Statements LTSORT and GTSORT

Vectors can be sorted with the statements GTSORT (in ascending order) and LTSORT (in descending order). The first vector slice specified is sorted accordingly. The other vectors are swapped component for component.

```
LTSORT ( name(1..number) )
      -->  the components of the vector 'name'
           are sorted in ascending order

LTSORT ( name(1..10), firstname(), city() )
      -->  the first ten components of the vector 'name'
           are sorted in descending order; the components
           of the vectors 'firstname' and 'city' are
           swapped accordingly

GTSORT ( name(1..5), firstname() )
      -->  the first five components of the vector 'name'
           are sorted in ascending order; the first five
           components of the vector 'firstname' are
           swapped accordingly
```

When all the components can be represented as numbers, the sorting is done in numeric order (e.g. 2<10). As soon as only one component cannot be represented as a number and is not the NULL value either, all components are treated as strings and sorted accordingly (e.g. 10<2). NULL values are regarded as the largest possible value, i.e. for LTSORT they are put at the beginning of the vector slice to be sorted and for GTSORT they are put at the end.

Syntax:

```
<vectsort stmt>  ::=  LTSORT ( <vector slice>,... )
                   |  GTSORT ( <vector slice>,... )
```

Calling Procedures, Forms, and Functions

The Statements CALL, SWITCH, and SWITCHCALL

With the CALL statement a procedure (or form) can call other procedures (or forms) of the same program:

```
CALL PROC insert;       /* call SQL-PL routine.
CALL FORM mastercard;   /* call form.
```

CALL is a subprocedure call: After executing the called module, the processing of the calling module is continued with the next statement after the CALL. This option can be used in DB Procedures and triggers if the subprocedures comply with the conditions for stored procedures. SWITCH and SWITCHCALL are not allowed in stored procedures (see Section, "Calling Subprocedures and Functions" in Section, "Stored Procedures").

In contrast to CALL, SWITCH...CALL causes a branching to a successor program from where no implicit branching back to the call location takes place.

Examples:

```
SWITCH reservation CALL PROC list;
SWITCH reservation CALL FORM start;
```

As a further possibility for a program branching, there is the SWITCHCALL statement which is a combination of SWITCH and CALL, as the keyword indicates.

SWITCHCALL branches to another program. In contrast to SWITCH, processing is continued after the SWITCHCALL, when the called program has been terminated. All variables of the program maintain their values.

Example:

```
SWITCHCALL db_io
   CALL PROC insert_customer (cno, resdat, rc);
```

In CALL, SWITCH, and SWITCHCALL statements, any expressions can be used instead of the fixed names. To distinguish these expressions from names, they must be preceded by a colon (:). This technique of a dynamic call cannot be used in stored procedures.

Examples:

```
READ name;
CALL PROC :name;

READ num;
appl := 'PART'#
mod  := 'START';
SWITCH :$USER . :appl  CALL PROC :mod;
```

Syntax:

```
<proc call>  ::=  CALL PROC <name expr> [[PARMS] (<param>,...)]

<form call>  ::=  CALL FORM <name expr>
                  [[OPTIONS] (<form calling option>,...)]
                  [PARMS (<param>,...)]

<switch stmt>  ::=  SWITCH [<name expr>.] <name expr> <form call>
                 |  SWITCH [<name expr>.] <name expr> <proc call>
```

```
<switchcall stmt> ::=  SWITCHCALL [<name expr>.] <name expr> <form call>
                     |  SWITCHCALL [<name expr>.] <name expr> <proc call>

<name expr>  ::=  <name> | :<expr>

<param>  ::=  <name>
```

Parameters for CALL, SWITCH, and SWITCHCALL

The modules of different programs do not have any common global variables. If certain values such as customer number and reservation date are to be passed to the successor program, when branching, they have to be passed via module parameters.

Example:

```
SWITCH reservation
  CALL PROC display PARMS (cno, resdat);
```

The parameter transfer for CALL, SWITCH, and SWITCHCALL is only possible when the called module has a declaration of formal parameters in the module header. In the module body, these formal parameters are used as local-dynamic variables of the relevant program.

Example:

```
PROC reservation.display PARMS ( c_no, r_dat );

WRITE CLEAR, 'customer no. :', c_no,
      NL,    'reservation date  :', r_dat;
```

The values of the current parameters are assigned to the formal parameters in the successor program. Arbitrary expressions and all kinds of variables, global as well as local, can be used as current parameters.

The assignment of current parameters to the formal parameters does not take place via the name (which can differ) but via the position in the parameter list.

The numbers of current and formal parameters do not have to agree. If no current parameter is to be assigned to the n-th formal parameter, the n-th current parameter specified in the call:

```
SWITCH ... PARMS ( cno, , counter+1 );
```

Vector slices can also be transferred as current parameters. The precondition for this is that the formal parameter is a vector.

```
Definition  :  PROC x.y PARM ( p() );
Call        :  CALL PROC y PARM ( a(1..5) );
```

In contrast to SWITCH, the parameters for CALL and SWITCHCALL function as input and output parameters. This becomes apparent when the current parameter is a variable and not an expression. After returning from the procedure or form, the variable may have a changed value.

```
A routine has the following statement with respect to the
parameter:

PROC customer.next PARMS (cno);
     ...
     cno := cno + 1;
```

```
    ...

Then after the sequence

 c_number := 4711;
 CALL PROC next (c_number);

 c_number has the value 4712
```

For transferring parameters, the same rules apply for functions as for procedures: All parameters have the effect of input and output parameters. Functions communicate exclusively via parameters.

Syntax of formal parameters:

```
<parm spec list>  ::=  [PARM[S]] (<var decl>,...)

<var decl>  ::= <varname> [<array spec>]

<array spec>  ::= ()
```

Calling Functions

The call of a function is introduced by a percentage sign. Function calls can be used in expressions, but they can also occur as independent statements.

Examples:

```
FUNCTION stdlib.sum PARMS ( s1, s2, s3, s4 )
RETURN ( s1 + s2 + s3 + s4 );

FUNCTION stdlib.list PARM ( reportname );
REPORT CMD ( ttitle ''':reportname''')
```

Examples:

```
sum := %sum (a, b, c, d, e);
%liste ('customerlist');
```

When used in expressions, the function must return a value to the calling environment via the RETURN statement. If no value is returned, the function yields NULL.

Functions cannot call procedures (CALL, SWITCH, and SWITCHCALL statements), but only further functions. The program to which a function is assigned by its name is called a library.

When calling a function, only the module name is specified. SQL-PL looks for the function in the library (program) called STDLIB as the default setting. If another library is to be used, this can be specified in the module options (see Section, "ModuleOptions"). In this way it is possible to administer functions, procedures, and forms in the same program.

Syntax:

```
<function call> ::= %mod_name [(<param>,...)]
```

Calling Stored Procedures

This section covers the following topics:

- DB Procedures
- Triggers
- DB Functions

DB Procedures

SQL-PL programs can also call DB Procedures, apart from other procedures and forms. DB Procedures are special SQL-PL procedures that run in the database kernel (see Sections, "DB Procedures" and "Creating Stored Procedures").

```
CALL DBPROC ins_customer (cno, rdat, rc) WITH COMMIT;

CALL DBPROC Miller.custappl.ins_customer (cno, rdat, rc);
```

A DB Procedure can be called with the CALL DBPROC statement while specifying the name of the procedure. If a DB Procedure belonging to another program or another owner is to be called, the appropriate owner name or program name must be specified.

The names must only be specified as constants. In addition, the called DB Procedure must already exist at the translation point in time.

The current parameters for a DB Procedure call can be SQL-PL variables, expressions or constants.

After the execution of a DB Procedure, a COMMIT is implicitly issued for the SQL return code = 0, but no ROLLBACK for the SQL return code <> 0, if this DB Procedure was called WITH COMMIT. Otherwise, the transaction concept of SQL-PL applies (see Section, "Database Accesses").

The call as SQL statement is equivalent to this DB Procedure call typical for SQL-PL:

```
SQL ( DBPROCEDURE custappl.ins_customer (:cno, :rdat, :rc) WITH COMMIT );

SQL ( DBPROC Miller.custappl.ins_customer (:cno, :rdat, :rc) );
```

Actually the call CALL DBPROC sends an SQL statement to the database kernel which means that both calls are equivalent (see Section, "Calling a DB Procedure").

If TEST DBPROC has been specified as module option, the processing of the procedure called with CALL DBPROC is not performed by the database server. Instead, the call for a DB Procedure is simulated internally so that the test facilities of SQL-PL are available. The called procedure need not be a DB Procedure nor be adapted for use as such, so that WRITE statements and REPORT calls can be used. The parameter definition, however, must be a DB Procedure.

In contrast to calling an SQL-PL module by CALL PROC or SWITCHCALL,

- each argument is checked on the basis of the parameter declarations of the called procedure.

- $RC and $RT are assigned, e.g., a runtime error or the STOP statement.
- subtransactions in DB Procedures must be formulated explicitly.

Syntax:

```
<dbproc call>  ::=  CALL DBPROC [[<owner>.]<progname>.]<dbproc name>
                          [PARMS (<param>,...] [WITH COMMIT]
                  |  SQL ( DBPROC[EDURE] [[<owner>.]<progname>.]<dbproc name>
                         [(<host var>,...)] [WITH COMMIT] )

<host var>  ::=  :<name>
```

Triggers

After defining a trigger in the database, it is executed as soon as the assigned SQL statement has been processed successfully and any conditions defined in addition are satisfied. An explicit execution is not possible. The trigger module can be called with CALL PROC like a common procedure from a program for testing purposes. A STOP statement, if any, results in inmediate abortion of the program. A call similar to CALL DBPROC is not possible.

DB Functions

DB functions can be executed in the database kernel within SQL statements. The module of the DB function can be called like a common function from a program for testing purposes. A STOP statement, if any, results in inmediate abortion of the program. A call similar to CALL DBPROC is not possible.

Embedding SQL

Database Accesses

For accessing database tables, the database language SQL is embedded SQL-PL language.

```
SQL ( SELECT DIRECT name, firstname, city, account
      INTO :cname, :cfname, :ccity, :account
      FROM  customer
      KEY  cno = :cno );

IF $RC = 100 THEN ...
```

In an SQL statement, SQL-PL variables are preceded by a ':' to uniquely distinguish them from column names. In the example, 'cname', 'cfname', 'ccity', 'account', and 'cno' are global variables of the program.

For each SQL embedding, a single DB statement can be specified. All SQL statements for the definition (DDL) and manipulation (DML) of database objects and for the search in database tables are permitted.

Examples: SQL statements

```
SQL ( INSERT customer (cno, name, firstname, city, account)
             VALUES (:cno, :cname, :cfname, :ccity, :account) );

SQL ( UPDATE customer SET city = :ccity, account = :account
             KEY cno = :cno  );

SQL ( DELETE customer KEY cno = :cno  );
```

An SQL statement always returns a numeric code that provides information about the state of the database after processing the statement. This code can be called via the $RC variable (see Section, "SQLErrorHandling").

```
SQL ( SELECT  name, firstname
      FROM  customer
      ORDER BY  name );

IF   $RC = 0
THEN
     SQL ( FETCH FIRST INTO :cname, :cfname );
```

After one of the SQL statements INSERT, UPDATE, DELETE, it can also be found out via the $COUNT variable whether the statement was successful and, if so, how many rows of the statement were affected. $COUNT produces either 0 or the precise number of the inserted, updated or deleted rows.

```
 SQL ( DELETE customer WHERE ccity = 'Washington' );

 WRITE CLEAR, $COUNT, ' rows deleted.';
```

After a SELECT the $COUNT variable returns :

0: if there is no hit, i.e. when the database does not contain any entry satisfying the qualification ($RC=100) or when an error has occurred ($RC <> 0).

>0: if the number of hits is known, that is, if a certain sorting was demanded in the query. In this case, the response time can be fairly long.

NULL: if the precise number is unknown.

After a single SELECT, the $COUNT variable returns either 0 for "not found" or 1 for "found".

The developer of SQL-PL programs must take account of the transaction concept of the database if he wants to program complex database applications. When explicitly or implicitly processing stored procedures in the database kernel, the transaction context of the respective calling application is valid.

When an SQL-PL program is called, SQL-PL implicitly opens a transaction. The SQL statement COMMIT WORK records the modifications of the current transaction in the database, terminates the transaction and opens a successor transaction. ROLLBACK WORK, by contrast, resets the database into the state at the beginning of the transaction.

SQL-PL implicitly uses the ISOLATION LEVEL 1 to synchronize transactions in multi-user operation; i.e., the updated entries are locked for other users and the entry last read cannot be modified by other users until the end of the transaction.

If a certain application requires other locks, it is possible to set explicit locks by means of the LOCK statement (see the "Reference" manual) or to assign another ISOLATION LEVEL (0 to 4) to the user by means of the component XUSER.

SQL-PL implicitly issues a COMMIT WORK before the procedure waits for user input; i.e. when a form is called or in case of READ. In this way the database is prevented from rolling back a transaction itself because a lock has been set for too long. This implicit COMMIT can be suppressed for single SQL-PL procedures by specifying the option AUTOCOMMIT OFF (see Section, "The Option AUTOCOMMIT OFF").

For TEST executions, no COMMIT statements are performed, not even when they come from SQL-PL procedures.

The execution of the program is concluded with COMMIT WORK. If the program has to be interrupted prematurely (runtime error), it is rolled back with ROLLBACK WORK.

The following must be taken into account when accessing the database:

- SELECT statements must stand statically before FETCH statements, preferably in the same SQL-PL module.
- If the comp option ’SQL CHECK’ has been set, the addressed tables must already exist so that the module can be successfully stored. If this option has been disabled, SQL statements are only checked for correct syntax.

  If a table definition is altered, the SQL-PL modules accessing the altered columns must be saved once again with STORE. Only in this way can the program correctly access the altered table. DB Procedures and triggers are removed from the database kernel and must be recreated.
- Tables in stored procedures must be qualified completely, i.e., together with the user name. Statements that influence the transaction (COMMIT, ROLLBACK, ...) are not allowed.

A complete description of the possible SQL statements is contained in the "Reference" manual. Their return codes are described in the "Messages and Codes" manual.

Syntax:

```
<sql stmt>  ::=  SQL [<dbname>] (<sql cmd>)
               |  <dynamic sql stmt>
               |  <dynamic fetch stmt>
               |  <mass fetch stmt>

 <sql cmd>   ::=  see Reference document
```

Dynamic SQL Statements

Dynamic SQL statements are generated when processing the SQL-PL module. They allow for flexible access to database objects which could not have been available when creating the SQL-PL module. Dynamic SQL statements also allow for a check for syntactical correctness of the dynamic SQL statement or a check for the privileges and the assignment between module and database object at the moment when being processed. Errors that might occur can only be found and handled during the processing.

There are three possibilities of formulating dynamic SQL statements in SQL-PL programs and stored procedures.

1. The SQL statement does not contain any SQL variables.

   Example:

   ```
   table  :=  'customer';
   READ upper bound;

   statement  :=  'SELECT * FROM ' & table
                      & ' WHERE ' & 'knr < ' & upper bound;

   SQL EXECUTE IMMEDIATE statement;
   REPORT;
   ```

   program

2. The SQL statement contains SQL variables or position indicators for SQL variables. At execution time a variable or an expression is assigned as current value to each position indicator.

   Example:

   ```
   CALL FORM search_arg ( name, city );


   SQL EXECUTE IMMEDIATE
       'SELECT * FROM customer ' &
       'WHERE name LIKE ? AND city LIKE ?'
       USING name, city;

   REPORT;
   ```

3. The SQL statement generated at runtime contains position indicators for input and output variables. The values or variables to replace them are to be read and checked for validity by the program. First a name is assigned to the SQL statement by means of the SQL PREPARE statement. Then the description of the SQL variables is found out by means of the SQL DESCRIBE statement. Validity checks can be made on the basis of such a description. Finally the SQL statement is executed by means of the SQL EXECUTE satement.

   Example:

   ```
   SQL PREPARE search_1 FROM
       'SELECT * FROM customer ' &
       'WHERE  name LIKE ? AND city LIKE ?';

   SQL DESCRIBE search_1 INTO descrip(1..2);

   CALL FORM arguments ( name, city, descrip (1..2) );
   SQL EXECUTE ssearch_1 USING name, city;

   REPORT;
   ```

   Note:
   In the last example the SQL DESCRIBE statement produces in 'descrip(1)' the value 'IN CHAR BYTE NOT NULL' as description of the variable 'name'.

Syntax:

```
<dynamic sql stmt>  ::=  SQL [<dbname>] EXECUTE IMMEDIATE <expr>
                       |  SQL EXECUTE IMMEDIATE <expr> USING <expr list>
                       |  SQL EXECUTE <name> USING <expr list>
                       |  SQL PREPARE <name> FROM <expr>
                       |  SQL DESCRIBE <name> INTO <vector slice>
```

Dynamic FETCH Statements

To be able to assign variables values that have been selected by means of a dynamic SQL statement, the dynamic FETCH statement is available. The values of the current row of the specified result table are assigned to the variables marked by ':' (local or global variables or vector slices).

If no result table is specified, the FETCH statement refers to the result table last generated. Within the result table, the current row can be determined with FIRST, LAST, NEXT, PREV or POS (:<num expr>). FIRST positions to the first row of the result table, LAST to the last; NEXT to the next row after the current row, PREV to the preceding row. The numeric expression specified with POS determines the position of the row within the order of the result table.

Example:

```
table   := 'customer';
READ upper bound;

statement  := 'SELECT res (name, city) FROM ' & table
              & ' where ' & 'cno < ' & upper bound;

SQL EXECUTE IMMEDIATE statement;

SQL FETCH IMMEDIATE FIRST res INTO cname, ccity;

WHILE $RC = 0 DO
    BEGIN
    WRITE NL, cname, ccity;
    SQL FETCH IMMEDIATE NEXT res INTO cname, ccity;
    END;
```

At translation time it cannot be checked whether the number of variables agrees with the number of select columns.

Syntax:

```
<dynamic fetch stmt>  ::= SQL [<dbname>] FETCH IMMEDIATE INTO <variable>,...
```

Mass Fetch

For a simplified processing of result tables, SQL-PL provides an extended FETCH statement. This statement, which can only be used in SQL-PL, serves to pass several rows of a result table into vectors with one call.

As in the case of the dynamic FETCH statement, it is also possible in the mass fetch statement to determine a named result table as well as the current rows.

Example:

```
SQL ( SELECT name,firstname
      FROM  customer
      WHERE city = 'Washington'
      ORDER BY name );

IF $RC = 0
THEN
      SQL ( FETCH INTO :cname(1..20), :cfname(1..20) );
```

The maximum number of rows to read is determined by the smallest vector slice (here 20 rows). After themass fetch has been executed, the $COUNT variable produces the number of results actually read.

The mass fetch statement can also be used after a dynamic SQL statement.

Syntax:

```
 <mass fetch stmt> ::= SQL [<dbname>] ( FETCH INTO <vector slice>,... )
```

Support of the Adabas Data Type LONG

It is possible to process LONG columns from SQL-PL. The descriptor provided by the DBMS for the OPEN statement (see "Reference" manual) must be specified in subsequent READ, WRITE, and CLOSE statements to identify the LONG column.

Examples: Operations on LONG columns

```
 A table with a LONG column could be defined as follows:


SQL ( CREATE TABLE document ( dno FIXED(2) KEY, dtext LONG ));

Assuming that there is a row in the table with dno=1,
the LONG column can be opened e.g. for writing:

dno := 1;
SQL ( OPEN COLUMN document.dtext
      KEY  dno = :dno
      AS :descr
      FOR UPDATE );

The SQL-PL variables 'position', 'len' and 'contents'
are assigned accordingly. The LONG column can then be
written and read as follows:

SQL ( WRITE COLUMN :descr
      POS :position
      LENGTH :len
      BUFFER :contents );

SQL ( READ  COLUMN :descr
      POS :position
      LENGTH IN :len
      LENGTH OUT :lenout
      BUFFER :contents );
```

```
After terminating the accesses, the LONG column is closed again
in the following way:

SQL (  CLOSE COLUMN :descr );
```

OPEN and READ of a LONG column can also be combined into the faster FREAD. The FREAD statement opens the LONG column in the same way as the OPEN statement and makes the contents of the LONG column available in the specified buffer, starting with the first position.

Example:

```
The LONG column inserted in the above example can now be
processed further with FREAD:

dno := 1;
SQL ( FREAD COLUMN document.dtext
      KEY  dno = :dno
      AS :descr
      LENGTH OUT :lenout
      LENGTH :len
      BUFFER :contents
      FOR UPDATE );

The contents of the LONG column can now be processed and
then stored again with the WRITE statement:

SQL ( WRITE COLUMN :descr
      POS :position
      LENGTH :len
      BUFFER :contents );

After terminating the accesses, the LONG column is closed
again in the following way:

SQL (  CLOSE COLUMN :descr );
```

A complete description of the processing of LONG columns is contained in the "Reference" manual.

Syntax:

For the complete syntax of SQL statements for manipulating LONG columns refer to the "Reference" manual.

Dynamic Opening of LONG Columns

If the LONG column to be opened is only to be determined at runtime, the OPEN statement has to be constructed dynamically. Since, as described above, it is not possible to assign output values of an SQL statement to SQL-PL variables, the LONG descriptor returned by the dynamic OPEN statement must be specially treated.

For this purpose, the statement FETCH LONGDESCR is available which must immediately follow the dynamic OPEN statement. The LONG column descriptor generated with the OPEN statement is thus assigned to an SQL-PL variable and can therefore be used for further LONG statements.

Example:

```
strcol := 'document.dtext';
dno := 1;
com := 'OPEN COLUMN ' & strcol &
       'KEY dno = ' & dno &
       'AS :descr FOR UPDATE';

SQL EXECUTE IMMEDIATE com;
SQL FETCH LONGDESCR INTO :descr;

...
```

Syntax:

```
fetch dynamic descr> ::= SQL FETCH LONGDESCR INTO :<variable>
```

Multi-DB Operation

SQL-PL programs offer the possibility of simultaneously working with up to eight databases and/or user areas. For this purpose a (symbolic) database name has to be specified in SQL statements between the keyword SQL and the statement. Before using this statement for the first time, a user area of a started database must be assigned to this symbolic database name. This is done by means of the CONNECT statement.

The CONNECT statement establishes the specified database connection. The user area to be assigned to the symbolic database name must be defined here. The parameters for user name, password, SERVERDB, and SERVERNODE can be specified either as constants (enclosed in single quotes) or as an arbitrary expression. The specification of a SERVERNODE is only necessary when the database is located at another node than the node of the current database.

SQL-PL programs which work with several database connections can only be compiled when the used symbolic database names are assigned to a user area by means of XUSER. The symbolic database name corresponds to the USERKEY in XUSER. Care must be taken with upper and lower cases.

The symbolic database name can also be specified as a variable in the CONNECT statement.

Examples:

```
CONNECT CUSTOMER_DB AS ( 'ALL', 'START', 'CUSTDB' );
       ==> every user of the program works in the specified user area.

/* or
db := 'customer_db';
CONNECT :db AS ( 'ALL', 'START', 'CUSTDB' );

/* or

REPEAT
WRITE 'Please specify serverdb name and servernode name: ';
READ NL serverdb, servernode, NL;
WRITE 'Please specify user name and password: ', NL;
READ uname, DARK, password;
CONNECT my_db AS ( uname, pw, serverdb, servernode );
   /*  ==>  each user defines the user area in which
   /*       the corresponding SQL statements are to
   /*       be executed.
UNTIL $RC = 0;

SQL my_db ( SELECT ... );
```

```
/* or
SQL my_db EXECUTE IMMEDIATE 'SELECT ...'

REPORT DBNAME = my_db;
/* or
SQL my_db ( FETCH ... );
/* or
SQL my_db FETCH IMMEDIATE ... ;
```

If errors occur during the execution of the CONNECT statement, $RC is assigned appropriately. Any procedures defined for the current program for the handling of SQL errors are not called implicitly.

If the database addressed in the CONNECT statement is not available, this is also reported in $RC and the program is continued. If, however, the attempt is made to establish more than eight database connections, the program is interrupted.

The program is also terminated if no CONNECT was performed beforehand for the symbolic database name used in an SQL statement.

The RELEASE statement cancels the connection to the specified database.

Example:

```
RELEASE my_db;
```

If a database connection is not explicitly cancelled with RELEASE, it is maintained until the database session is terminated or another CONNECT with the same symbolic database name is performed.

Syntax:

```
<connect stmt>  ::=  CONNECT <dbname>
                        AS ( <username>, <password>,<serverdb>[,<servernode>] )
                   |  CONNECT :<var>
                        AS ( <username>, <password>,<serverdb>[,<servernode>] )

<release stmt>  ::=  RELEASE <dbname>
                   |  RELEASE :<var>
```

All SQL statements can relate to the database designated by <dbname> by specifying a <dbname> behind the keyword SQL.

SQL Error Handling

The execution of an SQL statement always returns a numeric code that can be retrieved by the procedure via the variable $RC (synonym: $SQLCODE). The variable $RT (synonym: $SQLERRMC) returns an explanation of the error of up to 80 characters in length.

The Most Important SQL Return Codes ($RC)	
0	command successfully executed
100	no entry found with this qualification
200	key already exists

```
|                                                                              |
| 300       insert/update refused because the new values would violate the     |
|           database integrity                                                 |
|______________________________________________________________________________|
```

These SQL return codes should be handled in an SQL-PL module immediately after the SQL statement.

Example:

```
SQL ( INSERT customer (name,firstname,city,account)
             VALUES (:cname,:cfname,:ccity,:account) );

CASE $RC OF
    200: MESSAGE := 'Customer is already registered';
    300: MESSAGE := 'Customer data erroneous';
    OTHERWISE MESSAGE := $RT;
END;
```

As a rule, negative SQL return codes refer to error situations that have nothing to do with the application itself (syntax errors, operating state of the database, missing catalog definitions).

Procedures for SQL Error Handling

SQL-PL supports the handling of such error situations in programs as follows: Depending on the SQL return code it is examined whether the program currently running has a procedure with the name SQLERROR, SQLEXCEPTION, SQLNOTFOUND, SQLTIMEOUT or SQLWARNING. If so, this procedure is implicitly called with CALL. Otherwise, the current program is continued with the next statement.

```
SQLEXCEPTION  :  100 < $RC < 700
SQLNOTFOUND   :  $RC = 100
SQLTIMEOUT    :  $RC = 700
SQLERROR      :  $RC < 0
SQLWARNING    :  $SQLWARNING  (a warning is set)
```

The procedures for the handling of SQL errors can have a formal parameter. The name of the procedure in which the SQL error occurred is assigned to this parameter by the SQL-PL runtime system.

Example: SQLERROR procedure

```
PROC customer.sqlerror (name)

WRITE  CLEAR, 'SQL error in ' , name ,' :', $RC,
       NL, 'terminate the program? (y/n)';
READ  answer;

IF UPPER(answer) = 'J'
   THEN  STOP     /* program terminated
   ELSE  RETURN;  /* continue after the SQL statement
```

The implicit call of error procedures is not supported in stored procedures.

Syntax:

```
<sqlerror routine> ::= PROC <progname>.SQLERROR ( <modulename> )
                       <lab stmt list>

<sqlexception routine> ::= PROC <progname>.SQLEXCEPTION ( <modulename> )
```

```
                          <lab stmt list>

<sqlnotfound routine> ::= PROC <progname>.SQLNOTFOUND ( <modulename> )
                          <lab stmt list>

<sqltimeout routine> ::= PROC <progname>.SQLTIMEOUT ( <modulename> )
                          <lab stmt list>

 <sqlwarning routine> ::= PROC <progname>.SQLWARNING ( <modulename> )
                          <lab stmt list>
```

Catching Runtime Errors (TRY-CATCH)

The TRYCATCH statement allows runtime errors to be handled without interruption of the current program.

If a runtime error or a STOP statement occurs in the TRY part of the statement, the program branches, with the corresponding error number, to the CATCH part of the statement. If one of the selectors corresponds to the error number, the corresponding statement is executed and the program execution is continued after the CATCH statement. Otherwise, the error remains set. The selectors can be specified as in a CASE statement, but an OTHERWISE branch is not possible.

Example:

```
TRY
     BEGIN
     CALL PROC do_command (...);
     END
CATCH errno OF
     16102 : ERROR := 'There is not enough memory available'
                      'for this command';
     16801 : ERROR := 'command interrupted';
END;
```

Syntax:

```
<try catch stmt>  ::=  TRY <compound>
                       CATCH <variable> OF <case list>
                       END
```

Query Call

Result tables generated by the database can be prepared for output by means of the Adabas Report generator. The command language of this Report generator is embedded SQL-PL language in a way similar to SQL.

The stored commands defined in Query can also be called from an SQL-PL program. The results can then be used in further processing.

The usage of Query commands or the Report generator is not possible in stored procedures.

This section covers the following topics:

- Stored Commands

- Report Formatting
- Further Facilities
- Master/Detail REPORT

Stored Commands

The call of a stored command is marked by the keyword QUERY. As in the command line of Query (cf. the "Query" manual) the call is made by means of the Query keyword RUN, the name of the stored command as well as any necessary parameters.

SQL-PL allows the static or dynamic specification of the call.

In the case of a static specification, the optional keyword CMD is followed by the call enclosed in parentheses. Parameters that stand for values in the stored command can be specified by variables from SQL-PL. Variables must be identified by a ':' prefix. They are replaced by their current values (number or string enclosed in single quotes) before calling the stored command. Parameters standing for names (e.g. of tables or columns) must be explicitly specified. The syntax of the call is checked by the compiler. The usage relations between the SQL-PL module and the stored command (QUERY COMMAND) are maintained if the workbench option USAGE is set.

In the case of a dynamic specification of the call, the necessary keyword EXECUTE is followed by an expression that is evaluated and interpreted as a call command at execution time. Therefore a check cannot be made at compile time. The usage relation between the SQL-PL module and the command is not maintained either.

Stored commands of other users can be called if the user has a corresponding access privilege.

Examples:

```
no := 123;
QUERY CMD ( run customer_list :no 'customer list' );

/* or equivalent

QUERY EXECUTE 'run customer_list 123 ''customer list''';

no := 123;
list := 'customer_list';
QUERY CMD ( run miller.customer_list :no :list );

/* or equivalent

cmd  := 'run miller.customer_list ' &no& ' ''' & list & '''';
QUERY EXECUTE run_cmd;
```

Note:

If the name of the QUERY command contains lower case letters or special characters in Query, the name must be protected by " (double quotes) when calling it from SQL-PL in the same way as when storing it in Query. This also applies to the user name.

```
/* the command was stored by the user MILLER in QUERY with
/* STORE "customer list"

QUERY ( RUN Miller."customer-list" );
/* or
QUERY EXECUTE 'Miller."customer-list"';
```

The total length of the call command in parentheses must not exceed 139 characters after substituting the variables. The total length can only be checked at execution time. If the maximum value is exceeded, an error is reported.

After executing the SQL statement in the stored command, the current database transaction is implicitly terminated (COMMIT). If the module option AUTOCOMMIT OFF is set in the module in which the stored command is called, the database transaction must be terminated explicitly by the application programmer (see Section, "ModuleOptions").

Syntax:

```
<query stmt>  ::=  QUERY <querycmd spec>[<further facilities>]

<querycmd spec> ::=  [ CMD ]  ( <querycmd> )
                   |  EXEC[UTE] <expr>

<further facilities>  ::=  see "Further Facilities" (5.7.3)
```

Report Formatting

The Report generator is called with the keyword REPORT. After the keyword CMD , the Report statements described in the "Query" manual can be specified. The keyword CMD can be omitted in Report statements, if no result table is specified.

```
SQL ( SELECT city,name,cno,account
      FROM customer ORDER BY city,name );

IF ($COUNT > 0) OR ($COUNT IS NULL)
THEN
    REPORT CMD ( RTITLE 'C u s t o m e r  l i s t'
                 SEPARATOR '   '
                 NAME  1 'domicile'
                 WIDTH 1 10
                 GROUP 1
                 NAME  2 'customer'
                 WIDTH 2 12
                 NAME  3 'C.-No.'
                 NAME  4 'acc. bal.'
                 LEAD  4 '$ '
                 SUB SUM 'sum to &COL1 : ' 4
                 SUB AVG 'slice of &COUNT : ' 4
                 TOTAL 'total : ' 4);
```

The individual Report statements can be written without separators in consecutive lines. If there are several statements in the same line, they must be separated from each other by a ';'. Names can be enclosed in single quotation marks. They are replaced in the way they have been specified. Two single quotation marks following each other are represented as one quotation mark. Without enclosing quotation marks, names are converted into upper cases and must not contain a ';' (semicolon), a ''' (single quotation mark) or a ':' (colon), because these are interpreted as end of name/line separation, start of a character string or variable. A complete description of the available Report statements is contained in the "Query"

manual.

Variables with a ':' prefix can be used in Report statements, as in SQL statements. They are replaced by their current values before the REPORT call.

```
kopftext := 'CUSTOMER-LIST ' & DATE(DD.MMM.YY);
REPORT CMD ( RTITLE :header ...
```

As a rule, a REPORT call refers to the result table that was generated last. A result table is implicitly generated with the SELECT statement, with the exception of single row accesses (options DIRECT, FIRST, NEXT, PREV, and LAST).

Result tables can be named explicitly. After REPORT, the name of the result table is specified to which the following formatting refers. Instead of the constant name, an arbitrary expression can be used. To be able to distinguish this expression from the name of a constant, it must be prefixed by a colon (:).

Examples:

```
SQL ( SELECT customer_list (city, name) FROM customer );

REPORT customer_list CMD ( RTITLE ...

cl := 'customer_list';
REPORT :cl CMD ( RTITLE ...
```

If nothing else has been requested, SQL-PL displays the first section of the prepared report on the screen. With the scrolling functions of Report, any section can be displayed on the screen.

With the following Report statements, the report can also be output on the printer or into a file:

```
REPORT CMD ( RTITLE ...

       ... PRINT );                          /* print in addition
or
       ... PRINT ONLY );                     /* print only
or     ... PUT 'customer.rpt' ONLY);         /* file only
```

The representation of NULL values in the report can be set with the NULL statement of Report and, specifically for the user, via the Set function of the SQL-PL workbench.

After leaving the report, $ROWNO and $COLNO can be used to request the position of the cursor. Errors occurring in Report can be requested by using $RC and $RT.

Syntax:

```
<report stmt>  ::=  REPORT [<report result spec>]
                           [<further facilities>]

<report result spec>  ::=  <result table name>  [ CMD (<report cmd; ...) ]
                         |  [ CMD ]  ( <report cmd>;... )

<report cmd>  ::=  see "REPORT Formatting"

<result table name>  ::=  <name expr>

<name expr>  ::=  <name> | :<expr>
```

Further Facilities

To call Report or stored commands containing a REPORT call, various clauses can be specified in addition.

Thus the fields of the header can be set to values by the options PROGNAME, VERSION, MODE and HEADER. For this purpose, an expression (e.g. string or variable) is specified after each keyword and an optional equals sign. The expressions may be truncated to the field lengths that can be represented.

The options, their lengths and values:

Option	Field length	Defaults
PROGNAME	8	SQL-PL
VERSION	8	12 (SQL-PL Version)
MODE	12	REPORT
HEADER	40	Name of the result table

Examples:

```
head := 'customer list from: ' & DATE;
QUERY  ( run Miller."customer-list" )
       PROGNAME = 'list'
       VERSION '12'
       MODE  = 'display'
       HEADER  =  head

head := 'customer list';
REPORT HEADER head;
```

The final results determined by the Report generator for columns of the result table (SUM, AVG, MIN, MAX, COUNT or self-defined arithmetic expressions) can be used further in SQL-PL. To do this, a variable and the corresponding result definition are specified. A result is only non-NULL, if an arbitrary result has been calculated for the specified column when executing Report; i.e. if it is defined interactively either when Report is called or during the execution. A check whether the specified result is defined in the REPORT statement is not made.

Examples:

```
QUERY  CMD ( run customer_list )
       RESULT ( tot_sum = SUM (3) );

REPORT RESULT ( mini = MIN (4) );
```

Options for the execution of Report can also be defined. The option BACKGROUND serves to prepare a result table with the Report generator. Output on the screen, however, is suppressed, until another screen output (e.g. a form) has been made (see Section, "Superimposing Forms (BACKGROUND)").

The interactive modification of the Set parameters can be suppressed with SETOFF while Report is being executed.

SETLOCAL permits a temporary modification of the Set parameters; i.e. after leaving the Report display, the Set parameters are reset to the values before Report was called.

If no option is specified for the modification of the Set parameters, every change of the Set parameters has global effects; i.e. it is valid up to the next modification of the Set parameters. The effects of the modifications on the current program must be considered (in particular, when changing the date and time format or the decimal representation).

With the option NOHEADLINE the output can be modified in such a way that the header is kept vacant and the specification of the product name (Adabas) and of the names of the database and user are suppressed. The specification of this option can be used, e.g., in a master-detail presentation (see Section, "Master/Detail-REPORT") to lay out the detail output more plainly.

Examples:

```
QUERY  CMD ( run customer_list )
       OPTION ( SETOFF );

REPORT OPTIONS ( BACKGROUND, SETLOCAL, NOHEADLINE );
```

In multi-DB operation, too, result tables can be prepared with the Report generator or stored commands be executed. To do this, the symbolic database name, which has previously been assigned to a user area with the CONNECT statement, is specified after the keyword DBNAME= (see Section, "Multi-DB Operation").

Examples:

```
QUERY  CMD ( customer_list )
       DBNAME = staff_db;

REPORT DBNAME = staff_db;
```

Examples:

```
QUERY DBNAME = staff_db
      HEADER 'customer list'
      CMD ( run customer_list :no'customer list' )
      OPTION ( SETOFF )
      RESULT ( tot_sum = SUM (3) );

QUERY CMD ( run customer_list :no'customer list' )
      DBNAME = staff_db
      HEADER 'customer list'
      OPTION ( SETOFF )
      RESULT ( tot_sum = SUM (3) );


REPORT  res
        DBNAME = staff_db
        HEADER 'customer list'
        CMD ( TTITLE 'all customers'
        TOTAL 'total sum : ' 3 )
        OPTIONS ( BACKGROUND, SETLOCAL )
        RESULT ( tot_sum = SUM (3), average = AVG (4) );
```

The default key to leave the Report display is F3. When calling Report, it is possible to specify any keys with any labels to be used to leave the Report display. The keys specified for the call override the keys used by Report. The released key can be requested by using $KEY: The keys HELP, UP, and DOWN are mapped to F10, F11, and F12 as in the case of forms.

Examples:

```
SQL ( SELECT * FROM CUSTOMER );
REPORT
     F1='Help'
     CMD ( RTITLE 'Customer List' )
     HEADER = 'List from ' & date
     F2 = 'continue';

IF $RC <> 0
THEN
   CALL PROC ERROR_ROUTINE ( $RC, $RT );
ELSE
   CASE $KEY OF
     F1: CALL PROC MYHELP (...);
     F2: BEGIN
         WRITE 'Cursor was in line:', $ROWNO, PAUSE;
         ...
         END
    END
```

Syntax:

```
<further facilities>  ::= <further facility>...

<further facility>  ::=  DBNAME   [=] <dbname>
                       |  PROGNAME [=] <expr>
                       |  VERSION  [=] <expr>
                       |  MODE     [=] <expr>
                       |  HEADER   [=] <expr>
                       |  OPTION[S]  ( <query option>,... )
                       |  RESULT ( <result spec>,... )
                       |  <report key spec>

<query option> ::=  BACKGROUND | SETOFF | SETLOCAL | NOHEADLINE

<result spec>  ::=  [:] <variable> = <result spec>

<res spec>     ::=  SUM   ( <columnid> )
                  |  AVG   ( <columnid> )
                  |  COUNT ( <columnid> )
                  |  MIN   ( <columnid> )
                  |  MAX   ( <columnid> )
                  |  VAL1  ( <columnid> )
                  |  VAL2  ( <columnid> )
                  |  VAL3  ( <columnid> )
                  |  VAL4  ( <columnid> )

<columnid> ::=  <natural>

<report key spec> ::=  <basic key> [=] <expr>

<basic key>  ::=  F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8
                |  F9 | F10 | F11 | F12
                |  HELP   | UP     | DOWN
```

Master/Detail REPORT

A frequent application is to represent two tables that are connected by a foreign key and that are in a 1:N relation in master-detail form.

The master row corresponds to a row of the referenced table while the associated detail rows come from the referencing table.

Example:

The tables customer and reservation are defined as follows:

```
CREATE TABLE CUSTOMER
       (  CNO      CHAR(5)   KEY,
          ...                    )

/*
CREATE TABLE RESERVATION
       ( RNO  CHAR(5)  KEY,
         CNO  CHAR(5)  NOT NULL,
        ...
FOREIGN KEY (cno) REFERENCES customer ON DELETE CASCADE)
```

The following example represents the reservations (detail) for each customer row (master).

```
PROC  customer.mast-det



SQL  (  SELECT  result_customer
        ( customer.cno, customer.name, customer.firstname, customer.city )
     FROM   customer);

@pos := 1;
REPEAT
   SQL  (  FETCH POS (:@pos) result_customer
           INTO :cno, :name, :firstname, :city );
   SQL  (  SELECT result_reservation ( * )
           FROM reservation
           WHERE reservation.cno = :cno );
   IF $RC = 0
   THEN
        REPORT result_reservation
           ( WINDOW pos 7 1 size 16 80; WIDTH * )
           OPTIONS ( BACKGROUND, NOHEADLINE )
   ELSE
        MESSAGE := 'no detail rows found';

   CALL FORM display_master ( FRAME, SCREENPOS(1,1),
        ACCEPT( F3='END', F2='DETAIL',F7='PREV', F8='NEXT'));

   CASE $KEY OF
   F2 : REPORT result_reservation
              ( WINDOW pos 7 1 size 16 80; WIDTH * )
              OPTIONS ( NOHEADLINE )
   F3 : RETURN;
   F7 : @pos := @pos - 1;
   F8 : @pos := @pos + 1;
   END;
UNTIL $KEY = F3;
```

With the keys F7 and F8 the previous or the next customer row and the associated reservation rows are fetched simultaneously. The BACKGROUND option of the Report display refreshes the reservation rows, and Report returns control immediately to SQL-PL.

If the user wants to scroll in the report of the reservation rows, the F6 key must be pressed. By this means Report is called in such a way that it maintains the control (without BACKGROUND option).

Apart from this master-detail representation controlled by SQL-PL, it is also possible to allow the master and detail rows to be controlled completely by Report.

This is illustrated by the following example:

```
PROC customer.mast_det_report


SQL ( SELECT  result_customer
      ( customer.cno, customer.name, customer.firstname, customer.city )
      FROM   customer );

REPORT result_customer (
       MASTER
       DETAIL
       SELECT result_reservation ( * )
       FROM reservation
       FOREIGN KEY customer_reservation references customer);
```

This Report call provides the same facilities as the SQL-PL procedure in the example above.

The reference name, which must be specified after the keywords FOREIGN KEY, corresponds to the <referential constraint name> of the referential constraint definition> (cf. "Reference" manual, Section, "<create table statement>").

If the <referential constraint name> has not been specified explicitly with the table definition, Adabas implicitly generates the name from the names of the master and detail tables.

Editor Call

It is possible to call the editor from an SQL-PL procedure or a form. A vector slice or a variable must be specified as argument. The editor is not available in stored procedures.

If a vector slice is specified, each component of the vector corresponds to a row of the editing area.

```
 PROC customer.c_maintain
...
 EDIT ( comment (1..20) );
```

If a variable is specified, the text created with the editor is assigned to this variable. This text can be re-edited at any time. In particular, the text can be stored in a LONG column in the database.

```
PROC customer.c_maintain
...
SQL  ( OPEN COLUMN ... IN WRITE MODE );
SQL  ( READ COLUMN ... BUFFER :text );
EDIT ( text );
SQL  ( WRITE COLUMN ... BUFFER :text );
SQL  ( CLOSE COLUMN ... );
```

With this simple call, the edit area occupies the entire screen. Therefore there are some options to determine the position (POS), size (SIZE) and keys, according to the requirements.

The option MARK positions the cursor in the specified row and column and highlights this row.

The fields of the header can be set to values by the options PROGNAME, VERSION, MODE, and HEADER (or LABEL).

The number of edited lines can be restricted with the option MAXLINES.

With the option PRINT the contents of the specified variables are printed out. The options POS, SIZE, MARK, F1, ..., F9 have no effect here.

In the case of repeated editor calls with the same SQL-PL variable, the option NOINIT causes the position of the editor window (cursor position and first row displayed) to be restored just as it was the last time the editor was left.

The position values can be kept available for two variables.

By means of the option MSG or ERROR a message can be output in the message line when the editor is called. The message of the MSG option is displayed with the attribute for INFO message and the message of the ERROR option is displayed with the attribute for error message (see Set parameters ATTR5 and ATTR6, Section, "User-specific Set Parameters" or "Display Attributes (HIGH,LOW,INV,BLK,UNDERL,ATTR1..ATTR16)").

The following example contains some options:

```
PROC customer.c_maintain
...
EDIT  ( commentary (1..20),
        POS ( 5, 10 ),
        MARK ( 3, 7 ),
        SIZE ( 15, 50 ),
        MSG = 'editing can be done now',
        ERR = 'false input',
        LABEL = 'comment',
        PROGNAME = 'CUSTOMER',
        MODE = 'INPUT',
        F9 = 'ENDE' );
```

With this call, the editor appears on the screen as follows:

column 10

|

line V

```
      light| ****        #                                                  |
           | ====                                                           |
           |  ____________________________________________________________>>> |
           |   PRINT RIGHT ... END ...                                      |
           | editing can be done now                                        |
           | ==>                                                            |
           |________________________________________________________________|
```

The vector ’comment’ can now be edited as desired. With the scrolling keys, the user can write beyond the vector slice specified in the call, (i.e. in this example more than 20 lines).

All the editing statements of the built-in editor are available (see the "User Manual Unix" or "User Manual Windows").

When naming the keys care must be taken that the keys used by the editor itself are not used.

After leaving the editor the following variables are assigned:

- $EDITLINES to the number of edited lines.
- $CMD to the command line input in the editor.
- $KEY to the release key used.

Syntax:

```
<edit call>  ::=  EDIT ( <vector slice> [, <edit option>,...] )
                |  EDIT ( <variable> [, <edit option>,...] )

<edit option> ::=  POS (<expr>,<expr>)
                |  SIZE (<expr>,<expr>)
                |  MSG = <expr>
                |  ERR = <expr>
                |  LABEL = <expr>
                |  MARK (<expr>,<expr>)
                |  HEADER = <expr>
                |  PROGNAME = <expr>
                |  MAXLINES = <expr>
                |  PRINT
                |  NOINIT
                |  <programmable key> = <expr>
```

Line-oriented Input and Output

An SQL-PL program can communicate with the user via terminal screen forms (see Section "Forms",) or via row-oriented READ/WRITE statements.

```
WRITE CLEAR, HI( ’Lotto-Tip ’, DATE(DD.MM.YY) );


WRITE NL(2),  ’Name :’;
READ  name;

WRITE NL, ’Hallo ’, name, COL(20), ’Ihr Tip : ’;
READ  DARK, z1, z2, z3, z4, z5, z6, z7, zz8;
```

After WRITE there is a list of expressions and control options. The values of the expressions are calculated and written on to the screen in the order of their occurrence or the control options are executed.

All control options are optional.

If not something else has beendeclared by means of control options, WRITE separates the output values from each other by blanks and writes only to the end of the current screen line.

The WRITE statement outputs the NULL value in the representation which is set in the Set parameters.

The following control options are available for the WRITE statement:

CLEAR

clears the screen and assumes the left-hand top corner as current position. This statement makes sense if the screen needs to be cleared explicitly (see automatic clearing of the screen when using the NL statement). Since the control options are executed in the order in which they are noted, the CLEAR control option only makes sense at the beginning of a list of expressions.

CLEAR (l, c)

clears a rectangle from the current cursor position l lines downward and c columns to the right.

NOCLEAR

prevents the screen from being cleared when the edge of the screen is reached while executing the NL statement.

,NL(n)

shifts the current position one or n lines forward (n line feeds) to the beginning of the line. If the screen edge is reached while NL(n) is being executed, the next line feed has the effect of the CLEAR statement.

COL(n)

positions the cursor in the current line to the n-th column to the right. The control option can only position forwards. If the current position lies to the right of the desired column, COL does not have any effect.

HI(<expression list>)

Expressions behind the HI control option enclosed in parentheses are highlighted. The subsequent expressions appear with the previously valid intensity.

HIGH

All expressions following the control option HIGH are highlighted.

PAUSE

causes a confirmation to be requested from the user (e.g. pressing the ENTER key).

SIZE(n)

Normally, an expression is output with its current length. The SIZE option causes the following expression to be output in the length specified by SIZE. If the length specified by SIZE is shorter than the current length of the expression, it is truncated. If it is longer, the expression is filled with blanks to the specified length. The length n can also be formulated as a numeric expression.

POS(l,c)

indicates the absolute co-ordinates of the current position. The first parameter specifies the line number, the second (c) the column number. Line and column numbers can also be formulated as numeric expressions. The POS option, however, has an effect only when the following applies:

1<= l <= number of representable screen lines

1<= c <= number of representable characters per line

OPEN, CLOSE

The OPEN control option causes all the following WRITE output not to be displayed on the screen but to be held in background. The terminal screen output with the data gathered so far is explicitly released by the control option CLOSE or PAUSE. The terminal screen output is also automatically released when the edge of the screen is reached by a WRITE statement.

HOLD

has the effect that the user has to press a release key so that a full screen is automatically cleared. In the right-hand lower corner of the screen '...HOLDING' appears. After the release key is pressed, the screen is cleared and the output is continued.

The READ statement processes a list of variables and control options in the given order. Like the WRITE statement, the READ statement, too, separates the input fields from each other by blanks, if nothing else has been declared via control options. A READ statement can contain several variables that are read in the order in which they are written. A READ statement that does not contain any variables does not have any effect.

The following control options are available for the READ statement:

variable

causes an input field to appear at the current position. The user can type in a value and terminate the input with a release key. The value read in is then stored as the value of the variable. If no further control options have been specified, the field stretches to the edge of the screen and is filled with blanks. When filling the field, the characters appear with normal intensity. If the user does not input anything for the variable, it takes on the NULL value.

HIGH

causes the characters entered when filling a field to appear with highlighted intensity.

DARK

causes the characters input not to appear on the screen (no echo).

NL,NL(n)

changes the current position as in the WRITE statement.

COL(n)

changes the current position as in the WRITE statement.

POS(l,c)

changes the current position as in the WRITE statement.

SIZE(n)

restricts the size of the input field to the length specified in SIZE.

CLEAR

clears the screen as in the WRITE statement.

PROMPT'c'

causes the input field to be filled with the character specified with PROMPT (otherwise blanks).

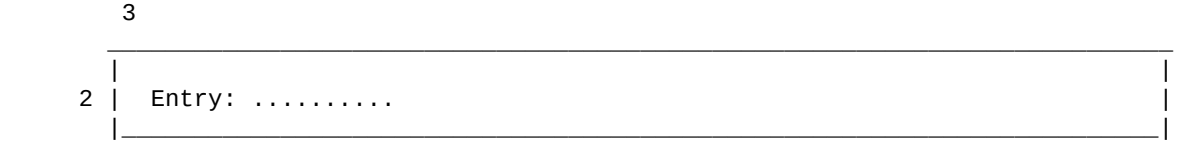
OUTIN(v)

causes the current value of the variable v to be displayed in the input field. If the variable has the value NULL, the field appears either filled with blanks or, if a PROMPT control option has preceded, with the PROMPT character.

Example 1:

```
answer := NULL;
WRITE POS (2,3) , 'input :';
READ  SIZE (10) , PROMPT '.', OUTIN(answer);

WRITE NL, 'today:';
today := date(dd.mm.yyyy);
READ  SIZE (10) , OUTIN(today);
```

Effect:



Since no position was specified in the READ statement, the input field 'answer' is output immediately behind the written text. After inputting a value, SQL-PL continues the processing with the next WRITE and READ statement and waits for the variable value 'today' to be input.

```
    5
  ____________________________________________________________________________
   |                                                                          |
 2 |  Entry : Hello.....                                                      |
   |                                                                          |
   |  Today : 07/27/2002                                                      |
   |__________________________________________________________________________|
```

Syntax:

```
<write stmt>  ::=  WRITE <write expr>,...

<write expr>  ::=  <expr>
                |  HIGH
                |  SIZE (<expr>)
                |  POS (<expr>,<expr>)
                |  PAUSE
                |  CLEAR [(<expr>,<expr>)]
                |  NL [(<natural>)]
                |  COL (<natural>)
                |  HI (<expr>,...)
                |  NOCLEAR
                |  OPEN
                |  CLOSE
                |  HOLD

 <read stmt>  ::=  READ  <read expr>,...

 <read expr>  ::= <variable>
                |  HIGH
                |  DARK
                |  SIZE (<expr>)
                |  POS (<expr>,<expr>)
                |  CLEAR
                |  NL [(<natural>)]
                |  COL (<natural>)
                |  PROMPT '<any char>'
                |  OUTIN (<variable>)
```

Processing Files

SQL-PL allows sequential operating system files to be read and written in programs. These statements are not available in stored procedures. Files must be opened for processing by means of an OPEN control option.

```
OPEN fileid filename openmode

fileid  : internal file identifier
filename  : external file identifier
openmode  : READ or WRITE or APPEND
```

In following READFILE or WRITEFILE statements definite files are referred to by means of the file id. Note: A file that has been opened to be read cannot be written.

```
WRITEFILE fileid   expr:length , ...
READFILE  fileid   variable:length, ...

expr      : any expression
            ( length specification is optional )
variable  : SQL-PL variable or vector component
            ( length specification mandatory )
length    : numeric expression
```

The condition

```
IF EOF (<fileid>)
THEN ...
```

is satisfied when a READFILE statement has been executed although the last record of the file had already been read in.

The CLOSE statement serves to close files explicitly.

```
CLOSE fileid
```

For all files that are still open when the program is terminated, an implicit CLOSE is executed.

The DELETEFILE statement can be used to delete files.

```
DELETEFILE filename
```

Syntax:

```
<open file stmt> ::= OPEN <fileid> <filename> <open spec>

<close file stmt> ::=  CLOSE <fileid>

<write file stmt> ::=  WRITEFILE <fileid> <write file args>

<write file args> ::=  <expr> [: <expr>]  [,<write file args>]

<read file stmt> ::=  READFILE <fileid> <read file args>

<read file args> ::=  <variable> [: <expr>  [,<read file args>] ]

<delete file stmt> ::=  DELETEFILE <filename>

<filename> ::=  <expr>
```

For partial support, SQL-PL provides the WRITETRACE statement which writes exclusively into the protocol file.

WRITETRACE only has an effect when the MODULETRACE or SQLTRACE option is used simultaneously.

Example:

```
WRITETRACE ’Customer: ’, cno, firstname, name;
```

Syntax:

```
<writetrace stmt> ::= WRITETRACE <write file
             args>
```

Calling Operating System Commands

Frequently, one wants to call a command on the operating system level from SQL-PL modules. The EXEC command serves this purpose. In its asynchronous version it can also be used in stored procedures. Take into account that a system environment other than in an SQL-PL program can be valid when processing the command in the database kernel. If the program to be called is not stored in the RUNDIRECTORY of the database kernel, paths for the call should be fully qualified. The corresponding privileges for the call of a program out of the database kernel must be set beforehand.

Normally, the operating system commands are called synchronously. Some operating systems allow an asynchronous command call, in addition.

For a synchronous call, a program result is returned in any SQL-PL variable. For an asynchronous call, there is no such result.

Examples:

```
 _____________________________________________________________________
|                                     |                               |
|                                     |  synchronous call under UNIX  |
|                                     |_______________________________|
|                                                                     |
|   EXEC 'ls -l' RESULT resultvar;                                    |
|_____________________________________________________________________|
```

For the synchronous call, there are two additional options that determine whether user interaction is desired or not. Without an option specification, the command executed synchronously must be confirmed by using the ENTER key. If the option NOHOLD is specified, the screen is cleared by the synchronous command, but no user input is expected. The option QUIET has the effect that the synchronous command is performed without any screen and user interaction.

```
 _____________________________________________________________________
|                                     |                               |
|                                     |  asynchronous call under UNIX |
|                                     |_______________________________|
|                                                                     |
|   EXEC ASYNC 'ls -l > list';                                        |
|_____________________________________________________________________|
```

Note:

1. For an asynchronous command or program call, restrictions specific to the operating system must be observed (see the "User Manual Unix" or "User Manual Windows").

2. Not every operating system allows foreign programs to be called.

Apart from the EXEC command, operating system commands can be issued and foreign programs be called via the command line at any time.

Examples

```
under Unix: EXEC ls -ls > list
 EEXEC vi sqlpl.prot
```

Other syntax formats adapted to the operating system concerned are described in the "User Manual Unix" or "User Manual Windows".

Syntax:

```
<exec command> ::=  EXEC [SYNC] <command> RESULT <variable> [<sync option>]
                  |  EXEC ASYNC  <operating system command>

<command> ::=  any command or
               any program call of the operating system

<sync option> ::=  NOHOLD | QUIET
```

SQL-PL System Functions

Arithmetic Functions

The functions TRUNC, ROUND, AVG, MIN, and MAX

With ROUND and TRUNC numbers can be rounded off or fractional digits be truncated.

The AVG function calculates the average of the specified values. MIN and MAX calculate the minimum or maximum resp.

If string arguments occur in the functions MIN and MAX, all arguments are interpreted as string values and the result is also returned as a string value. For the functions AVG, MIN, and MAX, NULL value arguments are ignored.

```
ROUND (12.1234567 , 5) -->  12.12346

TRUNC (12.1234567 , 5) -->  12.12345

AVG (1,2,3,4,5)        -->  3

MIN (1,2,3,4,5)        -->  1

MAX (1,2,3,4,5)        -->  5

ABS (-1)               -->  1

ABS (1)                -->  1

SQR (2)                -->  4

SQRT (4)               -->  2

PI returns the value of Pi to 18 decimal places

MDS MaxDataSize specifies the maximum string variable length
    that the system can handle

LN (EXP(5))            -->  5  (approx.)

SIGN (100)             -->  1
```

```
SIGN (-PI)             -->  -1

SIGN (0)               -->  0
```

The function SIGN provides the sign of the numeric expression.

Trigonometric functions SIN, COS, ARCTAN

The familiar trigonometric functions. The angle in radians is expected as argument.

Example:

```
SIN ( PI/2 )     -->  1
COS ( PI/2 )     --> -1
ARCTAN ( 1 ) * 4 --> 3.14159265358976
```

Syntax:

```
 <arith function>  ::=  ABS (<expr>)
                     |  SQR (<expr>)
                     |  ROUND (<expr>,<expr>)
                     |  SQRT (<expr>)
                     |  TRUNC (<expr>,<expr>)
                     |  SIN (<expr>)
                     |  COS (<expr>)
                     |  ARCTAN (<expr>)
                     |  EXP (<expr>)
                     |  LN (<expr>)
                     |  INDEX (<vector slice>,<expr>)
                     |  LENGTH(<expr>)
                     |  ORD (<expr>)
                     |  <index function>
                     |  <set function>
                     |  <strpos function>
                     |  <sign function>

 <index function>  ::=  INDEX (<vector slice>, [NOT] <expr>)
                     |  INDEX (<vector slice>, [NOT] NULL)

 <strpos function>  ::= see Section "String Functions"

 <set function>    ::=  MIN (<mixed expr>,...)
                     |  MAX (<mixed expr>,...)
                     |  AVG (<mixed expr>,...)

 <mixed expr>  ::=  <expr> | <vector slice>

 <sign function>  ::=  SIGN (<expr>)
```

String Functions

When implementing interactive applications, the user input and the output values usually have to be converted or prepared. There are two types of string functions:

- functions that have a string as argument and a string as result (<string function>)

- functions that have a string as argument and a numeric value as result (<strpos function>)

The following list of examples illustrates the way string functions work.

```
'.'(12)                                 -->  '............'
 n := 8; '-'(n)                         -->  '--------'
 BLANK(12)                              -->  '           '

 'to' & 'gether'                        -->  'together'
 '(' & $RC & ')'                        -->  e.g. '(0)'

 UPPER ('abc')                          -->  'ABC'
 LOWER ('XYZ')                          --> 'xyz'

 SUBSTR ('ABCDEFGH',3)                  -->  'CDEFGH'
 SUBSTR ('ABCDEFGH',3,2)                --> 'CD'

 TRIM ('  17.25  ')                     -->  '17.25'
 TRIM ('..17.25  ..','.')               -->  '17.25  '
 TRIM ('  17.25  ', ' ', RIGHT )        -->  '  17.25'
 TRIM ('  17.25  ', ' ', LEFT  )        -->  '17.25  '
 TRIM ('  17.25  ', ' ', BOTH  )        -->  '17.25'

 PAD ( 'abc', 7 )                       -->  'abc    '
 PAD ( 'abc', 7, RIGHT )                --> 'abc    '
 PAD ( 'abc', 7, LEFT  )                -->  '    abc'
 PAD ( 'abc', 7, BOTH  )                -->  '  abc  '

 LENGTH ('1234567890123')               -->  13

 $USER provides the 18-digit user name
 $GROUP provides the 18-digit name of the usergroup
 $USERMODE provides the user status (STANDARD, RESOURCE, DBA)
 $SERVERDB provides the name of the database

 firstnames (1..3) := NULL;
 firstnames (2) := 'Harry';
 INDEX (firstnames(1..3),'Harry')       -->  2

 STRPOS ('aabbccbbee','bb')             -->  3
 STRPOS ('aabbccbbee','bb',4)           -->  7
 STRPOS ('aabbccbbee','xx',4)           -->  NULL

 abc  := 'abcdefghijklmnopqrstuvwxyz';
 abc  := abc && UPPER ( abc );
 digits  := '0123456789'
 SPAN ('Miller, 1234', abc )            -->  6
 BREAK ('Miller, 1234', digits, 6 )     -->  8

 CHANGE ('   ',' ','_')                 -->  '___'
 CHANGE ('XX  XX',' ')                  -->  'XXXX'

 t(1..20) := TOKENIZE ( '1,2,,3', ',' );
 t(1)                                                                    -->  '1';
 t(2)                                                                    -->  '2';
 t(3)                                                                    -->  '3';

 t(1..20) := SEPARATE ( '1,2,,3', ',' );
 t(1)                                                                     --> '1';
 t(2)                                                                     -->  '2';
 t(3)                                                                     --> NULL;
 t(4)                                                                      -->  '3';

HEX ('xyz')                                                             -->  '78797A'
CHR (98)                                                                -->  'b'
 x := 98;  CHR(x)                                                      -->  'b'
 ORD ('b')                                                                 -->  98
 x := 'a'; ORD(x)                                                        -->  97
```

In string functions the NULL value is always handled like an empty string. The repeat operator (n) can only be used for single characters. The specified character is repeated as often as is determined by the number defined by the numeric expression.

The function HEX can be applied to any string expression . It provides a string twice as long in hexadecimal notation.

The function CHR supplies the character for the given numeric value that corresponds to the CHAR representation of this value. If something other than a numeric value is specified or if there is no CHAR representation for the numeric value, CHR returns the NULL value as result. The inverse function to CHR is the function ORD. It returns the corresponding numeric value for a character.

The concatenation (&) as well as the functions UPPER, LOWER, SUBSTR, TRIM, LENGTH, STRPOS, SPAN, BREAK, and CHANGE can also be applied to variables and arbitrary string expressions.

With the function TRIM, the specified character is removed from both ends of a string. With an additional argument, this can be restricted to one of the two ends.

With the function PAD, a string is filled with blanks to the specified length. With an additional argument it can be specified whether this should be done on the left or the right or on both sides.

The function INDEX can only be applied to vector slices. From the specified vector slice it supplies the index of the first vector component that has the desired value. If no such vector component is found, INDEX provides NULL as result.

The function STRPOS scans a variable value for the specified string. If it is found, STRPOS returns the starting position as result. Otherwise STRPOS returns NULL. The starting position of the search can be specified.

The function SPAN returns the position of the first character of the first string not contained in the second string. The starting position of the search can be specified.

The function BREAK returns the position of the first character of the first string contained in the second string. The starting position of the search can be specified.

CHANGE replaces the second string within the first string by the third string. By omitting the third string, the second string within the first string is deleted. All arguments can also be specified as variables or string expressions.

The function CHANGE assigns the number of arguments that have been changed to the $variable $ITEMS .

The functions TOKENIZE and SEPARATE fill a vector with the fields of a string. The fields are separated by the characters contained in the second argument. Consecutive separating characters are interpreted by TOKENIZE like one separating character, by SEPARATE, however, as fields with the value NULL. After the call, the system variable $ITEMS contains the number of fields detected.

With the FORMAT function numeric values can be flexibly prepared for output according to a predetermined pattern. The first argument of the FORMAT function can be a number, a variable or an arbitrary arithmetic expression:

```
FORMAT ( 1234,  '9 999' )          -->  '1 234'

FORMAT ( 1234,  '9,999.99 Kg' )    -->  '1,234.00 Kg'

FORMAT ( 12.3,  'DM 999,99' )      -->  'DM  12,30'

FORMAT ( 1.234, '9 Kg 555 g' )     -->  '1 Kg 234 g'

FORMAT ( 12.34, '.9999e-99' )      -->  '.1234e+02'
```

Each '9' in the mask marks the position of a digit. The first point or comma (from the right to the left) is interpreted as position and representation of the decimal sign. If the decimal sign is not to be represented by a point or a comma, the places after the decimal sign must be marked by a '5'.

If the mask does not contain any sign, only floating minus signs are set before the first digit. Otherwise, the '-' (only minus sign) or '+' (sign always) determines the position of the sign in the mask.

```
FORMAT ( -123,  '99 999' )        -->  '  -123'

FORMAT (  123,  '-9 999' )        -->  '   123'

FORMAT (  123,  '+9 999' )        -->  '+  123'

FORMAT (-1234,  '99 999-' )       -->  ' 1 234-'
```

The leading digit can be marked by 0, * or > instead of by 9. With 0, leading zeros are displayed, and with *, places in front of the number are filled with * (cheque protection). With >, the preceding floating text is set in front of the first digit:

```
FORMAT ( 123,   '099 999' )       -->  '000 123'

FORMAT ( 123,   '*99 999' )       -->  '****123'

FORMAT ( 123,   '$>99 999' )      -->  '    $123'
```

If the specified number cannot be prepared according to the pattern, the NULL value is returned by the FORMAT function.

Syntax:

```
<string function>  ::= TRIM ( <expr [ ,'<any char>'] )
                       TRIM (<expr>,'<any char>',<side>)
                     | PAD (<expr>[, <expr>])
                     | PAD (<expr>, <expr>,<side>)
                     | SUBSTR(<expr>,<expr> [,<expr>] )
                     | UPPER (<expr>)
                     | LOWER (<expr>)
                     | FORMAT (<expr>,'<char>...')
                     | HEX (<expr>)
                     | CHR (<expr>)
                     | CHANGE ( <expr>, <expr> <num expr> ] )

<string function>  ::= TOKENIZE (<expr>, <expr>)
                     | SEPARATE (<expr>, <expr>)

<side>  ::=  RIGHT|LEFT|BOTH

<strpos function>  ::=  STRPOS ( <expr>, <expr> [, <num expr> ] )
                     |  SPAN   ( <expr>, <expr> [, <num expr> ] )
                     |  BREAK  ( <expr>, <expr> [, <num expr> ] )
```

Date and Time Functions

The date and time functions belong partly to the string functions, partly to the arithmetic functions and partly to the conversion functions so that they are described here in a separate section.

With the functions DATE and TIME the day's date and the current time of day can be represented in a chosen format or in the format specified by the Set parameters:

```
DATE (YY)                       --> '02'

DATE                            --> date as specified in the SET menu.

DATE (DD.MMM)                   --> '07.Nov'

DATE (MM/DD/YYYY)               --> '11/07/2002'


TIME (HH:MM:SS)                 -->  '14:29:59'

TIME (HH:MM-SS)                 -->  '14:29-59'

TIME (HH:MM)                    -->  '14:29'

TIME                            -->  e.. '14:29:59'
```

It is also possible to convert a given date or time into another format. The input and output formats for date or time are described by a mask.

```
mydate := '20021107';

DATE (YY/MM/DD, mydate, YYYYMMDD)  -->  '02/11/07'


mytime := '11:23:01';

TIME (HHMM, mytime, HH:MM:SS)      -->  '1123'
```

If the description for the input or output format is missing, the entry from the Set parameters is used.

Instead of a self-defined format, the following predefined masks can also be used:

```
   ISO            YYYY-MM-DD   or   HH.MM.SS
   USA            MM/DD/YYYY   or   HH:MM AM  ( PM )
   EUROPE         DD.MM.YYYY   or   HH.MM.SS
   JIS            YYYY-MM-DD   or   HH:MM:SS
   INTERNAL       YYYYMMDD     or   HHHHMMSS
```

Apart from that there are functions that enable date and time arithmetic. The date must always be specified in the format 'YYYYMMDD' and the time in the format 'HHHHMMSS'.

```
ADDDATE ('200291231',1)             -->  '20030101'

SUBDATE ('20011231',31)             -->  '20021130'

DATEDIFF ('20020101','20030101')    -->  365


SUBTIME ('00105523','00000023')     -->  '00105500'
```

```
ADDTIME (’00105523’,’00000037’)      -->  ’00105600’

TIMEDIFF (’00000005’,’00000000’)     -->  5


DAYOFWEEK (’20020105’)               -->  2
        provides value between 1 and 7    (1=Monday)

DAYOFYEAR (’20020101’)               -->  1
        provides value between 1 and 366

WEEKOFYEAR (’20022101’)              -->  1
        provides value between 1 and 53


MAKETIME (10,59,33)                  -->  ’00105933’
```

Syntax:

```
<date function> ::=  <date function>
                  |  <date str function>

<date str function> ::=  DATE [ ([<date mask>]  ,<expr> [,<date mask>])]
                       |  TIME [ ([<time mask>]  ,<expr> [,<time mask>])]
                       |  ADDDATE (<expr>,<expr>)
                       |  SUBDATE (<expr>,<expr>)
                       |  MAKETIME (<expr>,<expr>,<expr>)
                       |  ADDTIME (<expr>,<expr>)
                       |  SUBTIME (<expr>,<expr>)

<date function> ::=  DAYOFWEEK (<expr>)
                  |  WEEKOFYEAR (<expr>)
                  |  DAYOFYEAR (<expr>)
                  |  DATEDIFF (<expr>,<expr>)
                  |  TIMEDIFF (<expr>,<expr>)
```

SET Function

The setting of some user-specific Set parameters can be determined and changed by the function SET. The changes have an effect on the Set parameters of the current application and do not affect the settings in the workbench. Neither the reading nor modifying variant of the Set function can be used in stored procedures.

If SET is called with a parameter, it returns the value specified Set parameter.

Example:

```
 lang := SET ( LANGUAGE );
```

If SET is called with two parameters, it sets the value of the Set parameter (of the first parameter of the SET function) to the value of the second parameter.

```
SET ( DATE, ISO );
SET (NULLVALUE, ’?’ );
SET (DECIMAL, ’//.’ );
SET ( PRESENTATION, ’BLACK’ );
```

All settings are valid immediately after the SET function has been executed.

The following table contains the valid descriptors and values of the individual Set parameters.

Identifier	Value and Description
LANGUAGE	setting of the current language (ENG, DEU)
DATE	EUR, ISO, JIS, USA, INTERNAL or a self-defined date format
TIME	EUR, ISO, JIS, USA, INTERNAL or a self-defined time format
DECIMALREP	decimal point and thousands separator
SEPARATOR	column separator for report
NULLVALUE	NULL value representation
COPIES	number of copies for printout
PRINTFORMAT	name of the print format as defined in the workbench SET menu
SYSEDITOR	name of the system editor
PROTOCOL	name of the SQL-PL protocol file (it must be a valid file name)
PRESENTATION	name of the current SQL-PL attribute presentation

Syntax:

```
<set function>  ::=  SET (<set id>)


<set stmt>  ::=  SET (<set id>, <expr> )

<set id>  ::=  COPIES
             |  DATE
             |  DATETIME
             |  DECIMALREP
             |  NULLVALUE
             |  LANGUAGE
             |  PRESENTATION
             |  PRINTFORMAT
             |  PROTOCOL
             |  SEPARATOR
             |  SYSEDITOR
             |  TIME
```

Time Measuring Functions

To determine runtimes (e.g. of SQL statements), the $variables $SEC and $MICRO can be used. For this purpose, the stop watch is started with the statement INITTIME and the two variables are initialized. GETTIME assigns the time passed since INITTIME to $SEC and $MICRO in seconds and microseconds resp.

The stop watch runs until the next INITTIME or until the termination of the program. With each GETTIME, $SEC and $MICRO are assigned the current values.

Example:

```
PROC timecontrol.test_appl;

READ prog_name;
READ start_module;

INITTIME;
SWITCHCALL :prog_name CALL PROC :start_module;

GETTIME;

WRITE 'execution of program ', prog_name, NL;
WRITE $SEC, $MICRO, PAUSE;
```

Syntax:

```
 <systime func> ::= INITTIME | GETTIME
```

System or $ Variables

SQL-PL makes numerous system values available in system variables. All system variables start with the '$' sign. For this reason, the concept $variable is used as a synonym for system variable.

System variables return either a numeric value, a string or a logical value. In the following, all system variables will be explained. In part there is a further explanation of the system variables provided with the subjects in whose context a $variable is used. All $variables are available in DB Procedures, but not for all variables the usage makes sense, because they refer to the input and output of data. $CURSOR, $ROWNO, $COLNO,$EDITLINES, $SCREENCOLS, $SCREENLNS, $MAXLINES, $MAXCOLS, $KEYLINES, $MSGLINES, $KEY, $ACTION, $FUNCTION(n), $CMD and $TERM are set to NULL.

$CURSOR

specifies the last position of thecursor in the form, that is, the sequential number of the input field or NULL if the cursor was not positioned at any input field (see also Section, "Cursor Control (MARK, $CURSOR)").

```
CALL FORM mastercard;

IF $CURSOR = 5 /* ZIP
THEN  CALL FORM zip_help ...
```

$COUNT

After one of the SQL statements INSERT, UPDATE, DELETE or SHOW, the $COUNT variable can be used to find out whether the statement was successful and, if so, how many rows were affected by the statement. $COUNT returns either 0, the precise number of inserted, updated, deleted or found rows or NULL if the number of rows found is unknown after a SELECT (see Section, "Database Accesses").

$RC $SQLCODE

The $RC variable (synonym: $SQLCODE) always returns a numeric error code after every SQL statement. The value 0 means that the SQL statement has been successfully executed.

Detailed descriptions of $RC are contained in all sections concerning database accesses.

$RT $SQLERRMC

Parallel to $RC, the variable $RT (synonym: $SQLERRMC) returns an explanation of the error with a maximum length of 80 characters.

$SQLWARN

The variable $SQLWARN is a logical expression. If an SQL statement returns warnings, the program branches to the THEN part at IF$SQLWARN. Subsequently, the warnings that have actually been set can be displayed via $SQLWARN(1) to $SQLWARN(15).

(For SQLWARN see the "C/C++ Precompiler" or "Cobol Precompiler" manual.)

Example:

```
IF   $SQLWARN   /* Are there any warnings?
THEN
     FOR i := 1 to 15  DO
        IF   $SQLWARN (i)
        THEN
            WRITE NL,
            'Warning ',i,' is set';
```

$SQLERRPOS

After a syntax error, the position within the SQL statement that led to the error is available in $SQLERRPOS.

$SYSRC

After all file accesses, the variable $SYSRC returns a numeric error code. Even for the call of an operating system command error situations may occur which can be requested via $SYSRC.

$SYSRT

The variable $SYSRT contains an error text belonging to $SYSRC.

$USER, $GROUP, $USERMODE

The variable $USER returns the user name of the user currently using Adabas, $GROUP the group name. If the user is not a member of a group, $GROUP as well as $USER return the name of the user. $USERMODE returns the status (STANDARD, RESOURCE, DBA) of the user.

$SERVERDB

The variable $SERVERDB returns the name of the database with which the user is currently working.

$ROWNO, $COLNO

After a REPORT call, the variables $ROWNO and $COLNO return the row and column of the REPORT display in which the cursor was last positioned. If the cursor was positioned outside the REPORT display, $ROWNO and $COLNO are 0.

$EDITLINES

The variable $EDITLINES returns the number of edit lines after the editor call.

$SCREENCOLS, $SCREENLNS

The variables $SCREENLNS and $SCREENCOLS provide the length and width of the window occupied by the form that was last called.

$MAXLINES, $MAXCOLS

The maximum size of a form is restricted by the physical size of the screen. The variables $MAXLINES and $MAXCOLS return the maximum length and width of the screen.

$KEYLINES, $MSGLINES

With some types of screen, FORM can make use of additional lines on the screen. If the implicit message line is used, the MESSAGE is output in an additional system line, if possible. Likewise, FORM uses a display line provided on some screens for this purpose to output the key assignments. With the variables $KEYLINES and $MSGLINES it is specified whether and how many lines are available for each purpose.

$KEY

The variable $KEY returns the last release key.

$ACTION

The variable $ACTION returns the action of the action bar that was last activated (see Section, "Action Bar with Pulldown Menus and BUTTON Bar"). This can also be the value NULL if the pulldown menu was left with the key CLEAR.

$FUNCTION

The dollar variable $FUNCTION returns the function of the pulldown menu hierarchy that was finally chosen. This can also be the value NULL if the pulldown menu was left with the key CLEAR. If, however, an action of the action bar was chosen that does not have a pulldown menu, $FUNCTION returns the same value as $ACTION.

$FUNCTION1, ... $FUNCTION4

The dollar variables $FUNCTION1 to $FUNCTION4 return the function last chosen from the pulldown menu level designated by its number. In this way it is possible to distinguish the same pulldown menus several times within a pulldown menu hierarchy (examples are contained in Section, "Forms").

$CMD

$CMD returns the command line entered in the editor.

$ITEMS

Returns the number of recognized or modified arguments as the result of the functions TOKENIZE, SEPARATE, and CHANGE.

$SEC, $MICRO

The variables $SEC and $MICRO return the time passed between INITTIME and GETTIME (see also Section, "Time Measuring Functions").

$OS

The variable $OS returns the name of the operating system used. The following text constants are possible as values:

- ’MSDOS’
- ’UNIX’ as well as NULL, if the operating system could not be recognized.

$TERM

The variable $TERM returns, as string, the type of the current terminal as specified by the environment variables $TERM or $DBTERM. $DBTERM is evaluated first. If it does not return a value, $TERM is evaluated. If this does not return a value either, the result is NULL.

Syntax:

```
 <dollar numeric variable>  ::=  $COLNO
                                |  $COUNT
                                |  $CURSOR
                                |  $EDITLINES
                                |  $MAXLINES
                                |  $MAXCOLS
                                |  $MICRO
                                |  $RC
                                |  $ROWNO
                                |  $SCREENCOLS
                                |  $SCREENLNS
                                |  $SEC
                                |  $SYSRC

<dollar string variable> ::=  $ACTION
                             |  $CMD
                             |  $FUNCTION | $FUNCTION1...| $FUNCTION4
                             |  $GROUP
                             |  $KEY
                             |  $OS
                             |  $RT
                             |  $SERVERDB
                             |  $SYSRT
                             |  $TERM
                             |  $USER
                             |  $USERMODE

<dollar boolean variable>  ::=  $SQLWARN
                               |  $SQLWARN [(<expr>)]
```

Module Options

In the header of an SQL-PL module, various options can be defined. They serve various purposes as described in the following sections.

Syntax:

```
<module header>  ::=  <module type> <progname>.<modname>
                      OPTION[S] ( <module option>,... )

<module type>  ::=  DBPROC
                  |  TRIGGER
                  |  DBFUNC
                  |  PROC
                  |  FUNCTION
                  |  FORM
                  |  HELPFORM
                  |  MENU                      /* options cannot be defined

<module option>  ::=  <loop option>
                    |  <autocommit option>
                    |  <sqltrace option>
                    |  <moduletrace option>
                    |  <test dbproc option>
                    |  <lib option>
                    |  <keyswap option>         /* see FORM syntax
                    |  <subproc option>
```

This section covers the following topics:

- The LOOP Option
- The Option AUTOCOMMIT OFF
- The Trace Options MODULETRACE and SQLTRACE
- The TEST DBPROC Option
- The LIB Option
- The KEYSWAP Option
- The SUBPROC Option

The LOOP Option

With the module option LOOP SQL-PL helps the developer to find endless loops in WHILE and REPEAT.

Notation:

```
PROC customer.reservation OPTION ( LOOP 25 )
...
REPEAT ...
```

The option has the effect that all the loops in the SQL-PL procedure 'customer.reservation' are run through 25 times at the most. With the 26th run, the procedure is interrupted with the runtime error 16024 which means that the program suspects an endless loop. If this runtime error occurs in stored procedures, the LOOP option leads to abnormal termination.

Syntax:

```
<loop option> ::= LOOP [=] <natural>
```

The Option AUTOCOMMIT OFF

Normally, an SQL-PL program implicitly terminates the current transaction (COMMIT) before each READ and FORM call. With the module option AUTOCOMMIT OFF, the program developer has the possibility of controlling the transactions explicitly . A further effect of the option AUTOCOMMIT OFF is that the database return code 700 (SESSION INACTIVITY TIMEOUT) has to be handled in the SQL-PL program. If this error occurs, the database connection is established again by SQL-PL and the statements of this transaction must be repeated by the SQL-PL program.

```
PROC customer.read OPTION ( AUTOCOMMIT OFF )
```

Thus transactions can be programmed in which the user is asked questions. There is, however, the danger that other users are blocked by holding database locks for a long time.

The module options are only valid for the execution of the module in which they are contained, not for modules that are called by it. Both options can be specified:

```
 OPTION ( LOOP 25, AUTOCOMMIT OFF )
```

Syntax:

```
<autocommit option> ::= AUTOCOMMIT OFF
```

The Trace Options MODULETRACE and SQLTRACE

To facilitate the error search for the programmer, SQL-PL provides the following translation options:

The SQLTRACE option causes all SQL statements of this module to be recorded in an operating system file.

The SQLTRACEALL option causes all subsequent SQL statements to be recorded in an operating system file up to the end of the program.

The MODULETRACE option causes all subsequent calls of modules and forms to be recorded from the call of the module onward.

The name of the protocol file can be set in the Set menu.

Example:

Mr Miller has chosen the trace options in one of his SQL-PL procedures:

```
PROC customer.accept OPTION ( MODULETRACE, SQLTRACE )
```

After he has started his program and has inserted a data record, the following content is contained in the protocol file:

```
MILLER.CUSTOMER.ACCEPT
  MILLER.CUSTOMER.MASTERCARD

CMD 030102
  INSERT CUSTOMER (cno, firstname, name)
         VALUES (:V001,:V002,:V003)
RC= 0000       ERRORPOS= 0000
IN  : 77777
IN  : Henry
IN  : Newman

CMD 030102
RC= 0000      ERRORPOS= 0000
 <MILLER.CUSTOMER.MASTERCARD
 MILLER.CUSTOMER.START
```

When using the MODULETRACE or SQLTRACE option, the WRITETRACE statement can be used for additional test output that is to be written into the protocol file (see Section, "Processing Files").

Syntax:

```
 <trace option>  ::=  SQLTRACE [ALL]
                   |  MODULETRACE
```

The TEST DBPROC Option

The TEST DBPROC option serves to test DB Procedures before they are created in the database kernel. In this way, procedures called with the statement CALLDBPROC can also make use of the SQL-PL debugging options.

Syntax:

```
 <test dbproc option> ::= TEST DBPROC
```

The LIB Option

The LIB option overrides the current library name. Without explicit specification of a library name via the LIB option, SQL-PL looks for every variable to be called in the library STDLIB.

Example:

```
 PROC customer.start
 OPTION ( LIB public.custlib )
```

This setting is maintained until the program returns to the calling environment or until another declaration is made. At any point in time, only one library can be used. This option can also be used in stored procedures.

Syntax:

```
<lib option> ::= LIB [<username>.] <libname>
```

The KEYSWAP Option

By means of the KEYSWAP option, the key assignments can be swapped. A detailed description is contained in Section, "The KEYSWAP Statement". The KEYSWAP option can only be used in a procedure. It has the same effect as the KEYSWAP statement in a form.

If it is desired to swap the key assignments, it is convenient to do so in the start module of a program. If the start module is a procedure, the KEYSWAP option can be used for this purpose. This option makes no sense in stored procedures.

Syntax:

```
<keyswap option> ::= <keyswap spec>
```

The detailed syntax is contained in Section, "The KEYSWAP Statement".

The SUBPROC Option

An SQL-PL module that starts with the keyword PROC and is defined with the option SUBPROC designates a dependent DB Procedure. A dependent DB Procedure can be called by other DB Procedures by means of CALL PROC. This option is a statement that requires of the compiler to test the statements of this procedure with regard to the restrictions for stored procedures. On principle, each SQL-PL procedure satisfying the restrictions for stored procedures can become a dependent DB Procedure. With this option, the suitability can be checked when translating, not only when creating the procedure.

The parameter list of a dependent DB Procedure corresponds to that of a normal SQL-PL procedure. This has the advantage that DB Procedures can communicate with dependent DB Procedures via vectors.

Syntax:

```
<subproc option> ::= SUBPROC
```

Stored Procedures

Stored Procedures are SQL-PL modules which can be used to extend the facilities of the database. SQL-PL allows three kinds of stored procedures:

- DB Procedures can be called directly by an application process, like an SQL statement.
- Triggers are always bound to a definite table. They are called implicitly by the database after an INSERT, UPDATE or DELETE statement.
- Db functions can be used in SQL statements wherever Adabas functions can be specified.

DB Procedures are used

- to improve the efficiency, since a DB Procedure is processed by the database server which reduces the need for communication.
- to simplify the programming, since complex sequences of SQL statements can be replaced by a single call of a DB Procedure and can be used centrally from several application programs.
- to simplify authorization, since it is not necessary to grant privileges for the addressed database objects in addition to the execute privilege for a DB Procedure.

Triggers are a useful means

- to formulate conditions for which the options of the CONSTRAINT clause do not suffice.
- to link SQL statements along with control statements to the combination of tables, SQL statements, and conditions in the database kernel.
- to keep information in other tables consistent.

DB Functions

- extend the standard functions of Adabas in the database kernel in a customized way.

More detailed information about stored procedures is contained in Section, "General Properties of SQL-PL Modules".

This chapter covers the following topics:

- Creating Stored Procedures
- Calling a DB Procedure
- Calling a Trigger
- Calling a DB Function

Creating Stored Procedures

To ensure that the Adabas catalog is maintained correctly and the required consistency conditions are satisfied, stored procedures must be created by means of the SQL-PL workbench. A stored procedure successfully created has the status ’->DB’ in the module list of the SQL-PL workbench. A stored procedure can call subprocedures and subfunctions. These obtain the status ’+DB’.

Restrictions for stored procedures are described in Section, "Restrictions" of this section.

This section covers the following topics:

- DB Procedures
- Triggers
- DB Functions
- Parameter Declaration
- Calling Subprocedures and Functions
- Calling DB Procedures from Stored Procedures
- Differences between CALL PROC and CALL DBPROC in Stored Procedures
- Setting the Error Number
- Restrictions

DB Procedures

An SQL-PL module is declared to be a DB Procedure by the keyword DBPROC. In the database, it is only activated by the ’ Object/Create in DB ’ menu item (see Section, "The ’Object’ Menu Item" and in Section, "Commmands", the Workbench command "PCREATE"). To ensure that parameters are provided with values from calling applications, the parameters of a DB Procedure must be declared according to mode (IN, OUT, INOUT) and data type (see Section, "Parameter Declaration").

Syntax:

```
<db procedure>  ::=  DBPROC <prog name>.<mod name>
                     [PARMS (<dbproc parm decl>,...)]
                     [OPTIONS (<module option>,...)]
                     [<var section>]
                     <lab stmt list>

<dbproc parm decl>  ::=  <dbproc param mode> <name> <data type>

<dbproc param mode>  ::=  IN | OUT | INOUT
```

Triggers

An SQL-PL module is declared to be a trigger by the keyword TRIGGER. In the database, it is only activated by using the ’Object/Create in DB’ menu item (see Section, "The ’Object’ Menu Item" and in Section, "Commmands, the Workbench command "TCREATE". This can only be done using the

workbench interactively because some specifications on database objects (tables or columns) must be checked against the existing database structures when creating the trigger. The following specifications are made for this command:

- the name of the trigger as database object. (This name is important for the identification of the trigger. It is meaningless for the usage of the trigger.)
- the name of the table to which the trigger is bound.
- the SQL statements after which the trigger is to be called. At present, these are INSERT, UPDATE, and DELETE. The call of the UPDATE trigger can be restricted to the updating of certain columns.
- optionally an additional condition that restricts the call of the trigger to certain rows of the table.

The parameters of a trigger must be declared according to mode (here only IN) and type. For each parameter, a column with the same name and the same type must exist in the table for which the trigger is created. After successfully processing the assigned SQL statement the trigger is called with the corresponding column values. The prefixes "OLD." and "NEW." before the parameter name allow the old or new column value to be addressed in UPDATE triggers.

```
TRIGGER space.inflation (
  IN  OLD.price   FIXED (6,2),
  IN  NEW.price   FIXED (6,2) );

  IF NEW.price > (OLD.price * 1.1)
  THEN
      STOP ( -29200, maximum 10 % price increase' );
```

Additional Predicates in Triggers

Apart from the predicates already familiar, the following additional predicates are available for the formulation of triggers:

- FIRSTCALL is true if the trigger was called by a single command or by the first row of a bulk command.
- LASTCALL is true if the trigger was called by a single command or by the last row of a bulk command.
- INSERTING, UPDATING or DELETING are true if the trigger was called by a corresponding SQL statement.

The usage of these predicates only makes sense within a trigger. In any other case they are not true.

Syntax:

```
<trigger>  ::=  TRIGGER <libname>.<funcname>
                [PARMS (<trigger parm decl>,...)]
                [OPTIONS (<module option>,...)]
                [<var section>]
                <lab stmt list>

<trigger parm decl>  ::=  IN <trigger column spec> <data type>

<trigger column spec>  ::=  <column name>
                          |  NEW.<column name>
```

```
                          |  OLD.<column name>

<trigger predicate>  ::=  FIRSTCALL|LASTCALL|INSERTING
                        |  UPDATING | DELETING
```

DB Functions

With the DB functions, SQL-PL allows the effects of the SQL statements to be extended by user-specific functions. A DB function is defined by a database user with DBA privileges and is available to all the other database users without defining execute privileges. A DB function can be used within an SQL statement wherever standard functions of Adabas can be used: in the SELECT statement either in the SELECT list or in the WHERE qualification; in UPDATE and DELETE statements at the corresponding positions, for the definition of CONSTRAINTs, etc.

An SQL-PL module is declared to be a DB function by the keyword DBFUNC. In the database, it is only activated by using the ’Object/Create in DB’ menu item (see Section, "The ’Object’ Menu Item" and in Section, "Commmands, the Workbench command "FCREATE". When doing so, the DB function is assigned a name which must be unique within the database. This name is used to call SQL statements. The parameters must be declared with the mode IN and their data types. The data type of the return code must be specified. The RETRUN statement passes values from the DB function to the execution environment of the SQL statement. If the DB function was terminated without processing a RETURN statement, its return code is NULL - or - if the data type BOOLEAN was specified - FALSE.

In addition to the restrictions for DB functions described in Section, "Restrictions" of this section DB functions must not contain SQL statements and subprocedure calls. Functions may be called.

Example: Definition of a DB Function

```
DBFUNC convert.first_upper ( IN name CHAR(10)): CHAR(10)

VAR
    result_name;

result_name := UPPER(substr(name,1,1)) & LOWER(substr(2));
RETURN (result_name);
```

Example: Defining the DB Function in the Workbench

```
FCREATE convert.first_upper AS FUPPER
```

Example: Result of the SELECT Statement

```
_____________________________________|_____________________________________
|                                    |                                    |
|   LASTNAME                         |            EXPRESSION1             |
|____________________________________|____________________________________|
|                                    |                                    |
|    ALFONS                          |            Alfons                  |
|    BREITENBACH                     |            Breitenbach             |
|    BAMBERG                         |            Bamberg                 |
|    meier                           |            Meier                   |
|    SCHneider                       |            Schneider               |
|____________________________________|____________________________________|
```

Syntax:

```
<db function>  ::=  DBFUNC <prog name>.<funcname>
                    (<dbfunc parm decl>,...): <data type>
                    [OPTIONS (LIB <prog name>)]
                    [<var section>]
                    <lab stmt list>

 <dbfunc parm decl>  ::=  IN <variable name> <data type>
```

Parameter Declaration

The mode of a parameter is IN (input parameter), OUT (output parameter, for DB Procedures only) or INOUT (input/output parameter, for DB Procedures only).

The type is equivalent to the SQL data types of Adabas: FIXED, FLOAT, CHAR, BOOLEAN, TIME, DATE.

A parameter of the type LONG cannot be declared. NOT NULL, default values, and DOMAIN types are not supported either. These declarations only have a meaning for the transfer of parameters. Within stored procedures, the parameter variables can assume any type.

```
DBPROC hotel.reservation (
 IN  custname  CHAR(40),
 IN  day       DATE,
 OUT hotel     CHAR(20),
 OUT cost      FIXED(6,2) );
```

Syntax:

```
<dbproc parm decl list>  ::=  <parm decl>,...
                            |  <dbproc parm decl>,...

<dbproc parm decl>   ::=  <dir> <name> <data type>


<dir>  ::=  IN | OUT | INOUT

<data type>  ::=  FIXED [ ( <unsigned integer> [, <unsigned integer> ] ) ]
                |  FLOAT [ ( <unsigned integer> ) ]
                |  CHAR  [ ( <unsigned integer> ) ]   [ BYTE ]
                |  DBYTE [ ( <unsigned integer> ) ]
                |  DATE
                |  TIME
```

Calling Subprocedures and Functions

A DB Procedure or a trigger can call SQL-PL procedures and functions as long as these satisfy the restrictions of stored procedures. These modules are called dependent procedures or functions and must exist before creating the stored procedure. When the stored procedure is created in the database kernel these modules are created as well; in the workbench display they have the status ’+DB’.

A dependent procedure is called with the CALL PROC statement; a dependent function is called with %<function name>. To call a function not defined in the stdlib program, the LIB option must be set.

Dependent modules cannot be called using the CALL DBPROC or SQL (DBPROCEDURE ..) statement. Several stored procedures can use the same dependent modules.

Dependent modules need not be stored procedures. Their parameter declaration can contain vectors or variables with LONG values. They have the mode ’INOUT’ in any case.

Calling DB Procedures from Stored Procedures

A DB Procedure can be called from a trigger or DB Procedure using CALL DBPROC or an SQL statement. The call is issued to the database kernel like an SQL statement within the same database session and transaction management.

A stored procedure containing a call of a DB Procedure can only be created in the database kernel when the DB Procedure to be called has been created beforehand and its parameters are compatible with the calling parameters.

DB Procedures can also be called from stored procedures with simple CALL PROC. Then they are executed like dependent procedures.

Differences between CALL PROC and CALL DBPROC in Stored Procedures

The CALL PROC statement issued for the SQL-PL interpreter has the effect that the binary program code is loaded into the kernel and executed within the interpreter call. Processing a STOP statement in a subprocedure therefore leads to the inmediate termination of the stored procedure providing the return code set.

CALL DBPROC, on the other hand, generates an SQL statement that is processed by the database kernel and finally results in another instance of the interpreter. A STOP statement within the called DB Procedure produces the set return code in the calling stored procedure in which it can be processed.

Setting the Error Number

If a DB Procedure is interrupted with a runtime error, a predefined error number and a predefined error text are returned to the calling program. In SQL-PL programs, they are available as $RC or $RT, in precompiler programs as SQLCODE or SQLERRMC.

This behavior can be imitated by the programmer of a stored procedure with the STOP statement. In this case, the first STOP parameter is regarded as an error number, the second parameter as an error text. Although it is also possible to use the predefined error numbers of the database or thetool components, this should be avoided if an analogous error situation has not arisen. Certain values are not permitted as error numbers because these could lead to erroneous behavior in the calling program. These numbers are:

0	COMMAND TERMINATED CORRECTLY
-8	EXECUTION FAILED, PARSE AGAIN
700	TIMEOUT FOR COMMAND INPUT (TRANSACTION ROLLED BACK)

To be sure when setting a return code of your own that the error number does not collide with other error numbers used by Adabas, it is recommended to use positive error numbers in the range of [29000..29999] or negative error numbers in the range of [-29999..-29000].

If a trigger is interrupted with a runtime error or by a STOP statement, the trigger fails and also the SQL statement that started the trigger. The implicitly opened subtransaction is always rolled back.

A DB Procedure only produces a return code not equal to 0 if a runtime error occurred or if the return code was set by the STOP statement.

```
DBPROC hotel.reservation (
IN   custname  CHAR(40),
IN   day       DATE,
OUT  hotel     CHAR(20),
OUT  cost      FIXED(6,2) );

SQL ( SELECT freeroom ( hotelname, curr_date, free, price )
      FROM hotel
      WHERE curr_date = :day
      AND   free  > 0
      ORDER BY priC ASC );

CASE $RC OF
0 :  BEGIN
     SQL ( FETCH FIRST freeroom INTO
           :hotel, :curr_date, :free, :cost );
     free := free - 1;
     SQL ( UPDATE hotel SET free = :free
           WHERE hotelname = :hotel AND curr_date = :curr_date );
     SQL ( CLOSE freeroom );
     END;
100 :
      STOP ( 100, 'no free rooms.' );
OTHERWISE
      STOP ( -300, 'error : ' & $RC );
END;
```

Restrictions

The following statements are not permitted in a stored procedure and lead to a translation error:

- Statements in which global or static-local variables are used

  (The processing of a stored procedure must not depend on previous calls neither of other users in order not to violate the transaction context.)

- Output to the screen, printer or an operating system file

  (A stored procedure is processed in the database kernel; under certain circumstances the output devices are not known or not selectable. In addition, the waiting time for user input or file accesses would lead to excessively long processing times for other users):

  - WRITE, READ
  - CALL FORM, EDIT, REPORT, QUERY

  - WRITEFILE, READFILE, OPEN, CLOSE
  - Module options MODULETRACE, SQLTRACE

- Synchronous processing of operating system commands

  (The stored procedure waits for the termination of the synchronous call and keeps resources of the database in that time. Therefore this statement can lead to excessively long waiting times for other users on the same database.):

  - EXEC [SYNC]

- Reading or setting Set parameters

  (When processing the stored procedure in the database kernel, the Set parameters of the calling user are not longer known.):

  - SET (...)

- Change of the current program

  (The transaction context is decisive for stored procedures The program context cannot be modified. A different program context can be obtained by calling a DB Procedure using CALL DBPROC.):

  - SWITCH, SWITCHCALL

- Termination of a transaction, multi-db operation

  (The stored procedure must not interfere with the session or transaction context of the stored procedure.)

  SQL (COMMIT), SQL (ROLLBACK), SQL (CONNECT ...)

The explicit opening and closing of subtransactions, by contrast, is possible (SQL(SUBTRANS BEGIN), SQL(SUBTRANS COMMIT) and SQL(SUBTRANS ROLLBACK) see the "Reference" manual).

In SQL statements database objects must be completely qualified with the name of the owner.

Dynamic SQL can also be used in stored procedures. But for a stored procedure with dynamic SQL, it is not possible to grant the access rights to the database objects when granting the execute privilege. In this case, the user calling the stored procedure must have all privileges required.

Some $ variables are set to NULL in stored procedures because their usage makes no sense (see Section, "System or $ Variables".)

A DB Procedure or a trigger can only be created when all database objects (DB Procedures, tables, etc.) which are directly or indirectly called by it exist in the database. Depending procedures and functions must exist and be translatable according to the restrictions described here

A DB function is subject to the same restrictions as a FUNCTION. Functions cannot call procedures (CALL, SWITCH, and SWITCHCALL statements), but only other functions (FUNCTION). Without more specifications, all functions called are expected in the standard library (STDLIB). The called functions are expected in another library if the LIB option is set accordingly in the calling DB function.

No SQL statements can be used in a DB function. Adabas returns a translation error if the attempt is made to define a DB function that calls directly or indirectly

If an SQL-PL statement cannot be used in stored procedures, this is noted in the description of the corresponding statement.

Calling a DB Procedure

A DB Procedure is called by the SQL statement DB Procedure (in SQL-PL also by the statement CALLDBPROC). The call out of other application programs is described in the corresponding manuals (ODBC, precompilers, ...). The processing is done in the database kernel by the SQL-PL interpreter.

The statement CALL DBPROC is exclusively used for calling a DB Procedure already stored in the database. If the DB Procedure does not yet exist in the database (status: ’RUN’, not ’->DB’) or if the specified parameters are not compatible with the formal parameters of the DB Procedure, an error is output for the CALL DBPROC statement when storing the SQL-PL module, if the SQL-CHECK option is set.

If the DB Procedure has been called with the option WITH COMMIT, the current database transaction is terminated if the call of the DB Procedure is successful.

If a DB Procedure existing in the database kernel is called from an SQL-PL application program with CALL PROC, it is not executed by the SQL-PL interpreter in the database kernel, but by the SQL-PL interpreter within the running application program. Only in this case the TRY..CATCH statement and the SQLERROR procedures (see Section, "Procedures for SQL Error Handling") have an effect on these procedures.

In DB Procedures, further DB Procedures are called with the statement CALLPROC.

Calling a Trigger

A trigger is executed by the database server after every command for which this trigger is defined. This means that a modification of these column values is no longer possible within the trigger. Before calling the trigger, the parameters are assigned the corresponding column values. If the trigger is interrupted by means of a STOP statement, the preceding SQL statement is also regarded as having failed. If several rows of a table are modified by an SQL statement, the trigger is called for each of these rows after completely processing the SQL statement.

Calling a DB Function

A DB function is executed by the database server within the processing of the SQL statement. Nothing can be said about the order of execution within the SQL statement, because this is determined by the SQL optimizer of the database kernel.

The Debugger

SQL-PL provides various facilities for searching errors in programs:

- tracing SQL statements (SQLTRACE option),
- tracing module and form calls (MODULETRACE option),
- test run of a module with display of the values of variables.

In addition, SQL-PL provides the application programmer with a further comfortable test aid. The integrated DEBUGGER allows detailed control of the program flow.

The debugger can be used to display and modify the contents of variables, to define, remove and display breakpoints, to display the contents of system variables, the call sequence as well as the parameters, to activate and deactivate the single step mode, to interrupt the execution, and to output HELP information.

Only those modules can be processed with the debugger that have been translated with the debug option.

When executing a program with the debugger, it can happen that the space for the contents of the variables is insufficient because internal debug information has to be managed as well. If required, the space can be enlarged using the Set parameter ’Variable Range’.

This chapter covers the following topics:

- Procedure
- Functions of the Debugger

Procedure

The following procedure is necessary for the debugger:

In the command line of the workbench or the editor, the debug option is enabled by entering DEBUG ON. Then the modules to be processed with the debugger must be retranslated with MSTORE or STORE. In the list of modules, DEBUG appears under state. As long as the debug option is enabled, all modules are translated to become capable of debugging. The debug option is disabled by entering DEBUG OFF.

Menus cannot be debugged.

The debugger is implicitly called when a module translated with the debug option is executed and the debug option is on.

For procedures and functions, the debugger appears with the following display:

```
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|________________________________________________________________________________|
|                                                                                |
|  WELCOME SQL-PL Integrated Debugger 001-005                                    |
|_________________________________ _____________________________________>>> |
|                                                                                |
| 0001 PROC welcome.hello                                                        |
| 0002                                                                           |
| **** output := 'hello';                                                        |
| 0004 WRITE issue;                                                              |
| 0005                                                                           |
|  ___________________________________ DB_ONE : MILLER ____________________>>> |
|   1=INFO 2=CLEAR 3=HALT 4=PRINT 5=NEXT 11=RIGHT 12=SPLTJOIN                    |
|   ===>                                                                         |
|________________________________________________________________________________|
```

WELCOME/HELLO/0003

In the lower half of the screen the SQL-PL module is displayed. The current statement is displayed in highlights in the middle line of the editing form. The editing form can be scrolled by means of the keys UP and DOWN. In the command line, the debugger commands described below can be input. Any messages are displayed in the upper half of the screen.

For forms the debugger appears with the following display:

```
 ______________________________________________________________________________
|                                                                              |
|   accountno : 11223344                                                       |
|   name :                                                                     |
|   firstname :                                                                |
|                                                                              |
|                                                                              |
|   balance :                                                                  |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|______________________________________________________________________________|
|                                                                              |
|  CUSTOMER SQL-PL        Integrated Debugger                        013-017   |
|_________________________________ _____________________________________>>> |
|                                                                              |
| 0013 AFTER FIELD                                                             |
| 0014 BEGIN                                                                   |
| **** SQL (SELECT DIRECT name, firstname INTO :name,:f_name                   |
| 0015 FROM customer KEY cno = :cno);                                          |
| 0017 NEXTFIELD account;                                                      |
|  ___________________________________ DB_ONE : MILLER ____________________>>> |
|   1=INFO 2=CLEAR 3=HALT 4=PRINT 5=NEXT 11=RIGHT 12=SPLTJOIN                  |
|   ===>                                                                       |
|______________________________________________________________________________|
```

CUSTOMER/ACCOUNT/0015

As for procedures, the current statement is displayed in highlights in the middle line of the editing form in the lower screen half. In the upper screen half the upper part of the form is visible. This part can be deleted by means of the CLEAR key so that the debugger commands are better to see.

In forms the debugger stops when SQL-PL statements are executed; i.e. within the CONTROL, BEFORE/AFTER FIELD and BEFORE/AFTER GROUP statements.

Functions of the Debugger

The debugger provides the functions described in the following:

This section covers the following topics:

- Displaying and Modifying Variable Contents
- Breakpoints
- Single-step Mode
- Continuing the Execution
- Displaying the Call Sequence
- Displaying System Variables
- Displaying Parameters
- Interrupting the Program
-
- Displaying Help Information

Displaying and Modifying Variable Contents

```
V <variable>
 V <vector slice>
```

displays the current value specified variables, vector components or vector slice.

```
 A <variable> [ <value> ]
 A <vector slice> [ <value> ]
```

The value of the variable, vector component or vector slice is replaced by the specified value. The subsequent execution of the SQL-PL program is done with the modified value. A number or a string specified. Strings must be specified without single quotes. They are accepted unchanged. If no value is specified behind the variable name, this variable is assigned the NULL value.

Examples:

```
v output
     in the above example:
     --> NULL
     --> hello   (after the first statement has been executed)

a output How are you ?
     in the above example:
     the WRITE statement writes hello if this command is input
     before the statement in editor line 003 is executed, and
     after that How are you ?

v @name(1..5)
     displays the values of the first five vector components of
     the local vector variable @name

a vect(5) 10
      assigns the value 10 to the 5th component of the vector 'vect'

a vect(1..10)
      assigns the NULL value to the first ten components of the vector 'vect'
```

Breakpoints

For each module that has been translated with the debug option, two types of breakpoints can be defined. On the one hand, statements can be specified. At the specified point, the execution of the program is interrupted before the given statement is executed. The programmer can thus request, for example, the values of variables at this point.

On the other hand, variables can be defined. The execution of the SQL-PL program is then interrupted for every change of this variable.

```
 B [ <program> / <module> / ] <line>
```

defines a breakpoint. The execution is interrupted before the statement in the editor line <line> (see prefix) is executed. The line concerned must contain a statement (not a blank line, etc.).

If a module name is not specified, the breakpoint is set in the specified line of the current module.

```
b appl/mod/0004  or  b 4
sets a breakpoint in the first example of this chapter
before the WRITE statement is executed.

BV <variable>
```

defines a breakpoint. The execution is interrupted if the value of the specified variable or vector component has been changed.

Global, local-dynamic, local-static variables and single vector components can be specified. Local variables always refer to the module that is currently being processed.

If a breakpoint is defined for a global variable, the execution of the program is interrupted every time this variable is modified. This is also true when this modification is made in a module that has not been translated with the debug option. In such a module, however, the current line cannot be displayed. For this reason, the debugger positions to the start of the module. No further breakpoints can be defined for this module.

```
bv customer_no

bv @i

bv vector_var (5)

R [ <program> <slash> <module> <slash> ] <line>

RV <variable>
```

removes the specified breakpoint.

```
L
```

shows a list of all breakpoints defined for statements.

```
LV
```

shows a list of all breakpoints defined for variables.

Single-step Mode

When the debugger is called, the single-step mode is enabled, i.e. the next statement is executed as soon as the key CONTINUE is pressed. If the single-step mode is disabled, all statements up to the next breakpoint (or up to the end, if there is no breakpoint) are executed, when the key CONTINUE is pressed.

```
S
```

enables or disables the single-step mode.

In the upper half of the screen, the following display appears:

STEP OFF (single-step mode is disabled now) or

STEP ON (single-step mode is enabled now)

Continuing the Execution

```
G
```

continues the execution of the program until the next breakpoint is reached or, if there is no further breakpoint, up to the end of the program. For this purpose, the single-step mode is implicitly disabled.

Displaying the Call Sequence

```
T
```

displays the sequence of SQL-PL modules that have been called (corresponds to MODULETRACE).

Displaying System Variables

```
$
```

displays the values of the system variables:

$USER, $GROUP, $SERVERDB, $SYSRC, $OS, $TERM, $RC, $COUNT, $CURSOR, $EDITLINES, $SEC, $MICRO, $ROWNO, $COLNO, $SCREENLINES, $SCREENCOLS, $MAXLINES, $MAXCOLS, $MSGLINES, $KEYLINES, $CMD, $KEY

```
$<name>
```

displays the value of the specified system variable.

Displaying Parameters

```
X
```

displays the values of the transferred parameters.

Interrupting the Program

```
H
```

or pressing the HALT key causes the program to be interrupted immediately.

By entering ? or pressing the HELP key, the program branches to the HELP mode of the debugger.

Displaying Help Information

By means of the CLEAR key, the upper half of the screen can be cleared again, e.g. after program or HELP information output.

Forms

This chapter covers the following topics:

- General Points on Forms and Menus
- The Form Layout
- Form Processing Statements
- Options for Form Calls
- HELP Forms as Pick Lists
- Action Bar with Pulldown Menus and BUTTON Bar
- Module Options

General Points on Forms and Menus

FORM distinguishes several types of modules:

- forms (FORM)
- HELP forms for displaying information (HELP)
- menus for defining an action bar with pulldown menus (MENU)

The forms of the type FORM and HELP consist of a layout part and a processing part. Menus have neither a layout part nor a processing part. They consist solely of a definition of an action bar with or without pulldown menus.

Forms

A form consists of a layout part and an optional processing part. Before entering or after leaving an input field it is possible to perform different actions in the processing part, such as displaying information when entering the field, performing validity checks with database queries, or calling another form, procedure or function when leaving the field.

By means of NEXTFIELD and NEXTGROUP statements it can explicitly be determined in the processing part in which order the fields are to be processed. Otherwise the user determines the processing sequence by cursor movements.

A form can be designed as genuine input/output form or as controlling module. Any SQL-PL statements can be formulated within the statements CONTROL, BEFORE/AFTER GROUP, and BEFORE/AFTER FIELD. Here can even other modules or REPORT be called and SQL statements be defined.

```
|  FORM customer.mastercard     -> This is the form 'mastercard',           |
|                               -> belonging to the program 'customer'      |
|                                                                           |
|  LAYOUT                       -> The layout drawing starts here.          |
|  ...                          -> Input and output fields and              |
|  ...                          -> any texts are defined here.              |
|  ENDLAYOUT                    -> The layout drawing ends here.            |
|                                                                           |
| ACCEPT (...);                 -> The processing statements are            |
| IGNORE..;                     -> specified here.                          |
|                                                                           |
| GROUP ..                                                                  |
|   FIELD                       -> Definition of what has to happen         |
|       BEFORE FIELD ...        -> when entering or leaving a field         |
|       AFTER FIELD ...         -> or group;                                |
|   FIELD ...;                  -> any SQL-PL statements                    |
| END;                          -> may be used.                             |
|                                                                           |
| AFTER GROUP ..                ->                                          |
|   BEGIN ... END;              ->                                          |
|___________________________________________________________________________|
```

Syntax:

```
<form>  ::=  FORM  <prog name>.<mod name>
                   [OPTIONS (<form option>,...)]
                   [PARMS (<formal parameter>,...)]
                   [<var section>]
                   <form layout>
                   [ <processing stmt>,... ]
```

HELP Forms

HELP forms start with the keyword HELPFORM and consist of a layout part and a processing part as well. Since HELP forms serve to display information, the following restrictions apply:

- No global variables are allowed.
- HELP forms cannot call any further forms, procedures or functions, but further HELP forms.

HELP forms are automatically positioned at runtime in such a way that the input field for which the HELP form was called remains visible. For the HELP forms, too, all call options can be used.

```
 <helpform>  ::=  HELPFORM  <prog name>.<mod name>
                            [OPTIONS (<form option>,...)]
                            [PARMS (<formal parameter>,...)]
                            [<var section>]
                            <form layout>
                            [ <field processing stmt>,... ]
```

Menus

Menus serve to centrally define an action bar with pulldown menus that can be used by all forms of the application. The action bar and its pulldown menus, however, can also be defined in the processing part of the form. It is therefore not mandatory to define a menu.

As a rule the programmer will start with the definition of the menus in the processing part of a form, when developing a menu module. This has advantages for the testing of pulldown menus.

A menu module can also only consist of an action bar without further pulldown menus.

Structure of a Menu

```
MENU customer.pull_down_menu  /* This is the menu                        |
                              /* 'pull_down_menu', belonging to          |
                              /* the program 'customer'.                 |
ACTIONBAR ( ...               /* The menu action bar is defined          |
            ... );            /* here.                                   |
PULLDOWN ... ( ...            /* The individual pulldown menus           |
               ... );         /* are defined here.                       |
...                           /* The menu module must                    |
PULLDOWN ... ( ...            /* contain at least the                    |
               ... );         /* definition of the action bar.           |
                                       |                                  |
```

Syntax:

```
<menu>  ::=  MENU <prog name>.<mod name>
                  [PARMS (<formal parameter>,...)]
             <actionbar>
             [ <pulldown> ]
```

General Properties of Forms and Menus

Forms , like procedures, are modules which are assigned to a program by their names. Within a program, all modules communicate via a common set of global variables (see Section, "Variables")

Each input or output field of a form is assigned a global or a local variable. By calling the form, the variable values in the form fields are displayed on the screen.

Forms can be called like procedures by means of various CALL statements.

1. CALL FORM <module name>
2. SWITCH <prog name> CALL FORM <module name>
3. SWITCHCALL <prog name> CALL FORM <module name>

Menus can only be called from forms with the INCLUDE statement. The INCLUDE statement corresponds to the CALL or SWITCHCALL statement depending on whether the menu belongs to the same program or not.

For communicating by means of global variables or parameters, the same rules apply as for calling SQL-PL procedures.

HELP forms are not explicitly called by means of CALL statements. Instead, they are assigned to one or several fields in the FIELD/HELP statement and are automatically called by pressing the HELP key.

The processing statements within the form definition determine when the form returns control to the calling procedure or form. Then the following $variables can be requested in the calling procedure or form:

$KEY

returns the release key which was used to terminate the form.

$CURSOR

returns the last position of the cursor, i.e. the sequential number of the input field at which the cursor was last positioned.

$ACTION

returns the last selected action of the action bar (if, in the called form, an action bar is defined instead of release keys).

$FUNCTION

returns the function of the pulldown menu hierarchy last selected. This can also be the value NULL, if the pulldown menu has been left with the END key. If, however, an action of the action bar without a pulldown menu has been selected, $FUNCTION returns the same value as $ACTION.

$FUNCTION1, ... $FUNCTION4

return the function last selected on the pulldown menu level designated by its number. In this way, identical pulldown submenus can be distinguished several times within a pulldown menu hierarchy.

The Form Layout

This section covers the following topics:

- Form Fields and Messages (MESSAGE, ERROR)
- Form Fields of Variable Lengths (>>)
- Multi-line Form Fields (")
- Vector Components with Constant Index
- Vector Components with Dynamic Index (FIELD/OFFSET)
- Field Numbers Instead of Variable Names
- Definition of LAYOUT Control Characters

Form Fields and Messages (MESSAGE, ERROR)

FORM distinguishes write-protected output fields, normal input fields (with preassignment and echo) and input fields without echo (e.g. for password input):

```
<today      ->  output field : the current value of the
or <today>   variable 'today' is displayed here; it cannot be overwritten

MESSAGE     ->  predefined global variable for outputting messages
                (MESSAGE, ERROR)

_cno        ->  input field : the current value of the variable 'cno'
                is displayed here;
                it can be overwritten

_(account)  ->  input field without display and echo
```

The first position of the form field is marked by '<' or '_'. For FORM to recognize a form field, the beginning of the line, a blank or a form field must be placed before the field.

Two consecutive fields can be recognized as input fields when instead of '_' another special character that is not valid for a variable name is defined as input field identifier (see Section, "Control Characters for Input an Output Fields (IN,OUT)").

To the right, a form field ends implicitly before the next character or at the end of the line, depending on what is found first. A line is 141 characters long. In the following, other ways of dimensioning a form field are described.

The values of the input variables are displayed in highlights and the values of the output variables and the form background with normal intensity. Here, too, means will be described which can be used to explicitly control the brightness.

The current value of the variable MESSAGE is always displayed with the display attribute for INFO message (ATTR5) set in the Set parameters. MESSAGE is designed for outputting system messages. If the LAYOUT drawing does not define any MESSAGE field, FORM implicitly reserves the bottom screen line for displaying MESSAGE.

The current value of the variable ERROR is always displayed with the display attribute for error message (ATTR6) set in the Set parameters and displayed in the same line as MESSAGE. If ERROR and MESSAGE both have values, the value of the variable ERROR is displayed and the variable MESSAGE is ignored.

It shall be emphasized here that the variables MESSAGE and ERROR exist exactly once at runtime and can be used from procedures, forms and even functions. In the case of functions, it is not important to which library they belong.

The form fields can also be displayed explicitly with another intensity or inversely or blinking. These options are described in Section, "Display Attributes (HIGH,LOW,INV,BLK,UNDERL,ATTR1..ATTR16)".

Form Fields of Variable Lengths (>>)

Forms can also be used to print out standard letters or invoices, that contain information from the database.

If data of varying length is to be inserted into fixed text segments, the subsequent text on the line can be closed up with the variable part by means of the ’>>’ operator.

Example:

```
 ____________________________________________________________________________
|                                                    |                       |
|                                                    |  Form Definition      |
|                                                    |_______________________|
|                                                                            |
|  FORM customer.address                                                     |
|  LAYOUT                                                                    |
|                                                                            |
|  ..item <it_no>>,<it_des>>, is not available.                              |
|                                                                            |
|  ENDLAYOUT                                                                 |
|____________________________________________________________________________|
```

Otherwise space for the maximum length had to be left empty, which would lead to irritating gaps in the text.

Multi-line Form Fields (")

The value of a variable can be spread over several field sections. The individual sections must start directly beneath each other in the same column, but they may be of differing lengths:

```
 ____________________________________________________________________________
|                                                    |                       |
|                                                    |  Form Definition      |
|                                                    |_______________________|
|                                                                            |
|  FORM customer.change                                                      |
|  LAYOUT                                                                    |
|                                                                            |
|  . ADDRESS : _addr_new + <-- 1st. field section                            |
|    _______   _"        + <-- 2nd. field section                            |
|              -"        + ...                                               |
|                                                                            |
|    OLD ADDRESS (1)     OLD ADDRESS (2)                                     |
|    <addr_old_1 +       <addr_old_2 +                                       |
|    <"          +       <"          +                                       |
|    <"          +       <"          +                                       |
|____________________________________________________________________________|
```

When inputting, the values of the individual sections (including all blanks) are concatenated and then assigned to the target variable.

If an output field is concerned, the individual sections are first filled with the variable value from top to bottom and then, at the end, with blanks.

It is recommended to explicitly limit the field sections, because otherwise they would reach up to the end of line of the editing form (141 characters). Although the field sections would not be completely visible, this could lead to unexpected effects.

Vector Components with Constant Index

One frequently wants to maintain several data records with the same structure in one form, e.g. when registering several customers by means of a form. For this purpose SQL-PL provides vector variables which help to comfortably structure the form like a table.

A vector variable can consist of up to 255 components. A vector component is described by its vector name and its index.

```
 ____________________________________________________________________________
|                                                    |                       |
|                                                    |  Form Definition      |
|                                                    |_______________________|
|                                                                            |
|  FORM customer.registration                                                |
|  LAYOUT                                                                    |
|  ...                                                                       |
|  Cust.no.        |  Firstname      |   Name                                |
| __________________|_________________|_____________                          |
|                   |                 |                                       |
|  _cno(1)          |   _firstname(1) |   _name(1)                            |
|  _cno(2)          |   _firstname(2) |   _name(2)                            |
|  _cno(3)          |   _firstname(3) |   _name(3)                            |
| ...                                                                        |
| ENDLAYOUT                                                                  |
|____________________________________________________________________________|
```

The SQL-PL procedure that calls this form could look like the following:

```
 ____________________________________________________________________________
|                                                    |                       |
|                                                    | Routine Definition    |
|                                                    |_______________________|
|                                                                            |
|  PROC customer.entries;                                                    |
|  ...                                                                       |
|  CALL FORM registration;                                                   |
|  FOR i := 1 TO 20 DO                                                       |
|      IF cno ( i ) IS NOT NULL                                              |
|      THEN BEGIN                                                            |
|           SQL( INSERT CUSTOMER (cno,firstname,name)                        |
|                VALUES (:cno(i),:firstn(i),:name(i)) );                     |
|           CASE $RC OF ...                                                  |
|____________________________________________________________________________|
```

To simplify the writing of such a form, vector components with ascending indexes appearing beneath each other can be specified briefly as vector slices.

The same form with vector slices:

```
 ____________________________________________________________________________
|                                                    |                       |
|                                                    |  Form Definition      |
|                                                    |_______________________|
|                                                                            |
```

```
|  FORM customer.registration                                                  |
|  LAYOUT                                                                      |
|  ...                                                                         |
|   Cust.No         | Firstname                | Name                          |
|  ________________|__________________________|_____________                  |
|                   |                          |                               |
|   _cno(1..20)     | _firstname(1..20)        |  _name(1..20)                 |
|   _               | _                        |  _                            |
|   _               | _                        |  _                            |
|  ...                                                                         |
|  ENDLAYOUT                                                                   |
|______________________________________________________________________________|
```

The notation ’cno(1..20)’ with the input fields beneath each other - marked only by ’_’ - is equivalent to the first notation with the vector components ’cno(1)’, ’cno(2)’,... .

For this notation, however, there is the restriction that the fields beneath each other have to have the same length. Otherwise, an error is reported when saving the form.

Apart from that, precisely the number of fields is expected that results from the specified slice of the vector.

Vector Components with Dynamic Index (FIELD/OFFSET)

Example:

```
 ____________________________________________________________________________
|                                                    |                       |
|                                                    |  Form Definition      |
|                                                    |_______________________|
|                                                                            |
|  FORM customer.registration                                                |
|  LAYOUT                                                                    |
|  ...                                                                       |
|   Cno             |  Name                                                  |
|  _________________|_____________________                                   |
|                   |                                                        |
|   _cno(1..20)     |  _name(1..8)                                           |
|          ...      |  ...                                                   |
|   _               |  _                                                     |
|                                                                            |
|  ENDLAYOUT                                                                 |
|                                                                            |
|  FIELD 1:cno(1..8), name(1..8) OFFSET x;                                   |
|____________________________________________________________________________|
```

- If the expression behind OFFSET (here the variable x) has a value greater than 0, the value of the vector component ’name(1+x)’ appears in the form field of the vector component ’name(1)’.
- This rule applies accordingly to all vector components that are addressed in this FIELD statement.
- Instead of the variable x, numeric expressions are also permitted.

Syntax: See Section, "The FIELD/OFFSET Statement".

Field Numbers Instead of Variable Names

When using the vectors in forms, it soon becomes apparent that the notation for vector slices requires quite a lot of space. In other cases, too, the space occupied by a form field can only be kept as small as possible by choosing a very short variable name.

A way out is provided by field numbers which can be placed in the form as representatives for arbitrary variable names, vector components or vector slices.

Example:

```
                                                    |                    |
                                                    |  Form Definition   |
                                                    |____________________|
|
|  FORM customer.registration                                            |
|  LAYOUT                                                                |
|  ...                                                                   |
|   C.no            |  Firstname          |  Name                        |
|  _________________|_____________________|______________                |
|                   |                     |                              |
|   _1              |  _firstname(1..3)   |  _name(1..3)                 |
|   _               |  _                  |  _                           |
|   _               |  _                  |  _                           |
|                                                                        |
|   ...                                                                  |
|  ENDLAYOUT                                                             |
|                                                                        |
|  FIELD 1:cno(1..3) ...                                                 |
|________________________________________________________________________|
```

The assignment of field numbers to the variables represented is done beneath the form layout with the processing statements. The processing statements that can follow the specification ’FIELD 1:cno(1..3)’ are described in Section, "Form Processing Statements".

Example:

```
                                                    |                    |
                                                    |  Form Definition   |
                                                    |____________________|
|
|  FORM customer.display                                                 |
|  LAYOUT                                                                |
|  ...                                                                   |
|    Firstname            |  Name                                        |
|  ______________________|_____________                                  |
|                        |                                               |
|    <1<firstname(1..3)  |  <name(1..3)                                  |
|  < <                   |  <                                            |
|  < <                   |  <                                            |
|                                                                        |
|   ...                                                                  |
|  ENDLAYOUT                                                             |
|________________________________________________________________________|
```

Definition of LAYOUT Control Characters

In the line that starts with the keyword LAYOUT and introduces the layout drawing, special characters can be defined as control characters with various functions.

The definition of control characters can also extend beyond several lines. Then, however, the line has to end before or after the ’=’ sign.

Syntax:

```
<begin layout line>  ::=  LAYOUT  [ <layout char spec> ]

<layout char spec>   ::=  <layout attr char spec>
                        |  <layout inout char spec>
                        |  <layout prompt char spec>
                        |  <layout graphic char spec>
```

This section covers the following topics:

- Display Attributes (HIGH, LOW, INV, BLK, UNDERL, ATTR1..ATTR16)
- Control Characters for Input and Output Fields (IN,OUT)
- Displaying NULL Value Fields (PROMPT)
- Graphic Characters in Forms

Display Attributes (HIGH, LOW, INV, BLK, UNDERL, ATTR1..ATTR16)

The assignment of brightness and color to the 16 display attributes ATTR1 ... Attr16 (HIGH, LOW, INV, BLK, UNDERL) can be set in the Set parameters (see Section, "User-specific Set Parameters").

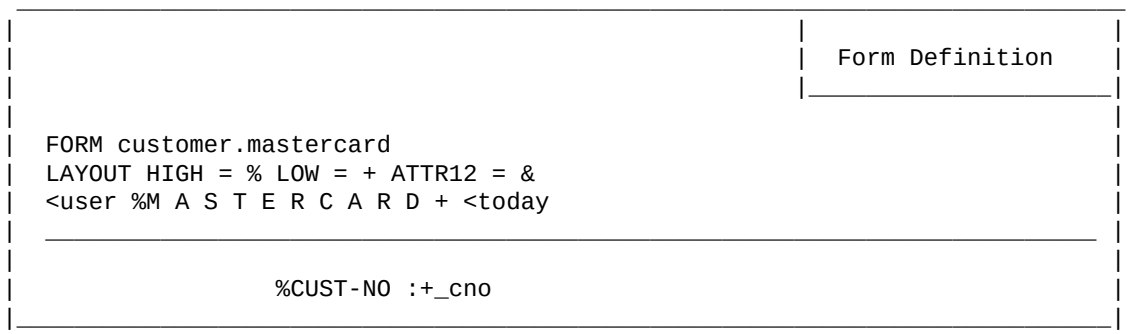
If nothing else is declared, FORM displays the values of the input fields highlighted (HI/ATTR2) and the output fields normal (LOW/ATTR1).

The value of the system variable MESSAGE is displayed with the attribute for info message (ATTR5) and the value of the system variable ERROR with the attribute for error message (ATTR6).

In the LAYOUT line special characters can be declared with HIGH, LOW, INV, BLK, and UNDERL or ATTR1.. ATTR16. These special characters can be used to control the display intensity and colors for each form element.

The control characters are valid from the position where they are located up to the next control character, or, at the most, to the end of the line. They are not displayed.

Example:



It is also possible to explicitly assign an attribute to each field without control characters in the layout by means of the FIELD/ATTR statement (see Sections, "Assigning Field Attributes (FIELD/ATTR)" and "Situation-dependent Display Attributes(SPECIALATTR)").

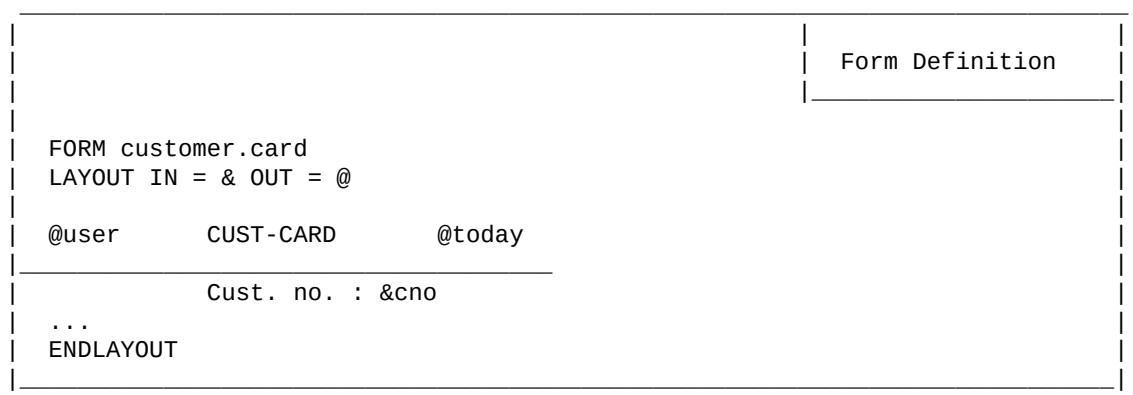
The display attributes can be overridden in each field by specifying the ATTR option in the form call (see Section, "Overriding Display Attributes (ATTR)").

Syntax:

```
<layout attr char spec>  ::=  <attr name> = <spec char>

<attr name>  ::=  LOW | HIGH | INV | BLK | UNDERL
                |  ATTR1 | ... | ATTR16
```

Control Characters for Input and Output Fields (IN,OUT)

Example:



Instead of the characters '_' (underscore) and '<' (less than) that identify the input and output fields, other characters can explicitly be chosen by means of IN and OUT. In this way, e.g. two input fields can be defined in the layout directly one after the other as the underscore is interpreted as part of the name.

Syntax:

```
<layout inout char spec>  ::=  IN  = <spec char>
                            |  OUT = <spec char>
```

Displaying NULL Value Fields (PROMPT)

Before an SQL-PL variable is assigned a value, it has the value NULL. This means that the variable does not have a value. NULL value variables appear as empty fields in forms.

Frequently, input fields are marked in the form by a series of periods or '_' characters (see 'cno' in the example for input and output fields).

After the form is called, the periods or '_' characters that are not part of the input value have to be removed. SQL-PL provides the TRIM function for this purpose that also removes the blanks before and after the value.

cno := TRIM(cno,’.’);

In the forms the procedure is simplified by the PROMPT option. It is specified in the LAYOUT line and declares an arbitrary character to mark NULL value input fields.

```
 ____________________________________________________________________________
|                                                    |                       |
|                                                    |  Form Definition      |
|                                                    |_______________________|
|                                                                            |
|  FORM customer.mastercard                                                  |
|  LAYOUT PROMPT = .                                                         |
|                                                                            |
|              CUST-NO : _cno                                                |
|               ...                                                          |
|                                                                            |
|  ENDLAYOUT                                                                 |
|                                                                            |
|  FIELD cno SIZE 5 INIT NULL;                                               |
|                                                                            |
|____________________________________________________________________________|
```

Input fields whose variables are preset to NULL are filled with the character declared in the PROMPT option. Otherwise, the current variable value is displayed.

If the variable ’cno’ still does not have a value or has been set to NULL, the field in the example form appears as follows:

```
CUST-NO : .....
```

When using the PROMPT option it is a go to explicitly limit the lengths of the input fields (see Sections; "Explicit Field Limiting (FIELD/SIZE)" and "Field Limiting for Multi-line Fields (FIELD/WIDTH)").

After input has been made by the user, the PROMPT characters and blanks before and behind the value are implicitly removed.

Note:
In a form with PROMPT option the following applies:

- An input field is recognized as NULL value if, apart from blanks, it contains only PROMPT characters.
- An input field is recognized as BLANK value if it is empty or contains only blanks (i.e. no PROMPT characters).

In a form without PROMPT option the following holds:

- An input field is recognized as NULL value if it is empty or only contains blanks.
- In this case, an input field cannot be recognized as a BLANK value.

Input fields that are displayed write-protected by the INPUT calling option and have the value NULL are not filled with the PROMPT character.

Syntax:

```
<layout prompt char spec> ::= PROMPT = <spec char>
```

Graphic Characters in Forms

The tool components also run on screens with graphics facilities. The editor and the option FRAME already take this into account. Where possible, several vertical or horizontal lines appear as continuous lines.

Now it can explicitly be determined in FORM whether a sequence of characters (vertical or horizontal) should appear as a continuous line or not.

Example :

```
 _____________________________________________________________________________
|                                                                             |
|                                                                             |
|   FORM box.test                                                             |
|   LAYOUT GRAPHIC=*                                                          |
|   **************                                                            |
|   *              *                                                          |
|   *              *                                                          |
|   *              *                                                          |
|   **************                                                            |
|   ENDLAYOUT                                                                 |
|_____________________________________________________________________________|
```

The rectangle defined in the example is represented on graphics terminals by continuous lines. On terminals without graphics, the horizontal line appears as a sequence of ’-’ (hyphens), the vertical line as a sequence of ’|’ (bars) and the corners as ’*’.

Syntax:

```
 <layout graphic char spec> ::= GRAPHIC = <spec
             char>
```

Form Processing Statements

A form can consist solely of its layout definition. Then the function of the form is merely the display and the subsequent reading in of the fields defined in the form layout.

For all functions beyond this, there is a series of processing statements that will be explained in the next section.

Syntax:

```
<processing stmt>  ::=  <field stmt>
                     |  <group stmt>
                     |  <before group stmt>
                     |  <after group stmt>
                     |  <ignore stmt>
                     |  <mark stmt>
                     |  <accept stmt>
                     |  <keyswap stmt>
                     |  <returnonlast stmt>
                     |  <autopage stmt>
```

```
|  <bottomlines stmt>
|  <headerlines stmt>
|  <special attr stmt>
|  <insertmode stmt>
|  <control stmt>
|  <actionbar stmt>
|  <pulldown stmt>
|  <include stmt>
|  <scrollfield stmt>
|  <keys stmt>
```

This section covers the following topics:

- The FIELD Statement
- Dividing into Field Groups (GROUP)
- BEFORE/AFTER GROUP
- Ignoring the Input Check (IGNORE)
- Initializing the Cursor Position (MARK)
- Key Activation (ACCEPT)
- The KEYSWAP Statement
- Leaving the Form (RETURNONLAST)
- Scrolling Support (AUTOPAGE, PAGE)
- Header Lines and Bottom Lines (HEADERLINES, BOTTOMLINES)
- Situation-dependent Display Attributes (SPECIALATTR)
- Displaying the Input Mode (INSERTMODE)
- Controlling the Dialog Sequence (CONTROL, CASE)
- The PAGE Statement
- PICK/PUT Mechanism in Forms (PICK/PUT/AUTOPUT)
- The NEXTFIELD Statement
- The NEXTGROUP Statement
- The SCROLLFIELD Statement
- The KEYS Statement

The FIELD Statement

The FIELD statement plays a central role among the processing statements. All field-specific activities of the form interpreter are formulated using this statement.

Syntax:

```
<field stmt>  ::=  FIELD <field name>,...[<field proc spec>...]

<field name>  ::=  <variable>
                |  <vector slice>
                |  <field number>:<field name>

<field proc spec>  ::=  <init spec>
                     |  <size spec>
                     |  <check spec>
                     |  <offset spec>
                     |  <display spec>
                     |  <help spec>
                     |  <domain spec>
                     |  <attr spec>
                     |  <noinput spec>
                     |  <autonext spec>
                     |  <before field spec>
                     |  <after field spec>
```

This section covers the following topics:

- Initializing the Fields (FIELD/INIT)
- Explicit Field Limiting (FIELD/SIZE)
- Field Limiting for Multi-line Fields (FIELD/WIDTH)
- Checking the Input (FIELD/CHECK)
- Using DOMAIN Definitions (FIELD/DOMAIN)
- The FIELD/HELP Statement
- Preparing Output Fields (FIELD/DISPLAY)
- The FIELD/OFFSET Statement
- Assigning Field Attributes (FIELD/ATTR)
- NOINPUT Fields
- NOAUTONEXT Fields
- BEFORE/AFTER FIELD

Initializing the Fields (FIELD/INIT)

In a form that consists only of theLAYOUT description, the calling procedure must ensure that the variables for the form fields are correctly preassigned. If a variable is set toNULL, blanks (or, if defined, PROMPT characters) are displayed in the associated form field.

The initialization of the form fields , however, can also be shifted into the form definition. This is useful if the preassignment is usually the same. In the exceptional case it can still be suppressed with the NOINIT option (see Section, "Suppressing the INIT Phase (NOINIT)").

The INIT option of the FIELD statement serves the initialization.

Example:

```
 _____________________________________________________________________________
|                                                     |                       |
|                                                     |  Form Definition      |
|                                                     |_______________________|
|                                                                             |
|  FORM customer.mastercard                                                   |
|  LAYOUT PROMPT = .                                                          |
|  <user    C U S T O M E R M A S T E R C A R D                  <today       |
|                                                                             |
|_____________________________________________________________________________|
|                                                                             |
|              CUST-NO : _cno                                                 |
|                                                                             |
|                                                                             |
|                                                                             |
|_____________________________________________________________________________|
|                                                                             |
|  <MESSAGE                                                                   |
|  ENDLAYOUT                                                                  |
|                                                                             |
|  FIELD user INIT $USER;                                                     |
|  FIELD today INIT DATE(DD.MM.YY);                                           |
|  FIELD cno SIZE 6;                                                          |
|  FIELD MESSAGE INIT 'Please enter';                                         |
|                                                                             |
|_____________________________________________________________________________|
```

The form of the user Miller defined in this way appears on 11/01/97 as follows:

```
 _____________________________________________________________________________
|                                                     |                       |
|                                                     |  Form Definition      |
|                                                     |_______________________|
|                                                                             |
|   Miller    C U S T O M E R M A S T E R C A R D                 11/01/02  |
|                                                                             |
|_____________________________________________________________________________|
|                                                                             |
|              CUST-NO : ......                                               |
|                                                                             |
|                                                                             |
|                                                                             |
|_____________________________________________________________________________|
|                                                                             |
|  Please enter                                                               |
|                                                                             |
|_____________________________________________________________________________|
```

If the current variable values are wanted instead of the preassignment defined by INIT when calling the form, the NOINIT option is to be used.

Syntax:

```
<init spec>  ::=  INIT <assign expr>

<assign expr>  ::=  <expr> | NULL
```

Explicit Field Limiting (FIELD/SIZE)

A form field starts with '<' or '_' and ends on the right with a blank before the next character and at the latest at the end of the line (standard rule):

```
ACCOUNT-BALANCE : _account
           .........       <-- input area
```

Since the control characters themselves are not displayed, they can also be used to limit fields on the right.

Example (LAYOUT LOW=+):

```
 FIRSTNAME : <cfname +
         ............     <--  display area
```

If the field is to be shorter than the variable name, the field length can also be defined with the SIZE option in the FIELD statement:

```
 FIELD function SIZE 1 ...
```

The length of a form field cannot be changed dynamically.

Syntax:

```
 <size spec> ::= SIZE <numeric>
```

Field Limiting for Multi-line Fields (FIELD/WIDTH)

In the case of multi-line fields it is desirable to be able to specify how the total length of the field should be and which length each individual field should have. The total length of the multi-line field is defined with FIELD/SIZE. The width of the individual partial fields is laid down with FIELD/WIDTH.

Syntax:

```
 <width spec> ::= WIDTH <numeric>
```

Checking the Input (FIELD/CHECK)

The values input by the user are only checked when this is explicitly demanded in the form definition.

For each input field, one condition and an associated message can be specified via the CHECK option of the FIELD statements.

If an input value does not satisfy the required condition, the cursor is set to the field concerned and the message behind ELSE in the CHECK option is displayed via MESSAGE. Then the user can correct the input immediately.

If no message text was specified behind ELSE, a substitute message is displayed.

If no fields in the form have been declared with MESSAGE, FORM displays the message in the system line provided for this purpose or in the bottom line of the screen.

The CHECK condition is processed when leaving the field. Only when the CHECK condition is satisfied, the processing is continued with the AFTER FIELD clause or - if this is not available - with the next field.

If no CHECK condition has been infringed, FORM sets the MESSAGE variable to NULL.

There are several other ways of formulating a CHECK condition:

1. The list of fields to be checked only contains simple variables (no vector slices).

   - Behind CHECK a <check cond> can be used without using the field name again. The <check cond> condition can also consist of arbitrary partial conditions joined by AND or OR. All IS, IN, BETWEEN or LIKE conditions and their negations are permitted as partial conditions (see <check cond> in Section, "Boolean Expressions"). The formulated condition applies to all variables of the field list.

   - Behind CHECK, <expr> followed by <check cond> can be used. With <expr>, e.g., a function value of the field can be checked.

     Note:
     'FIELD a, b CHECK UPPER (a) in (....)' checks the condition formulated regarding field a in field a as well as in field b.

   Examples:

Form Definition

```
FORM customer.mastercard
...
ENDLAYOUT

* composite expressions
 FIELD zip SIZE 5
     CHECK BETWEEN 1000 AND 99999
           AND IS NOT NULL OR IS BLANK;

/* regular expressions in the LIKE condition
FIELD cno SIZE 5
    CHECK LIKE '(A-z)(0-9)(0-9)(0-9)(0-9)'
          ELSE 'letter followed by 4 digits';

/* regular expressions in the LIKE condition
FIELD cno SIZE 5
    CHECK LIKE '(A-z)(0-9)(0-9)(0-9)(0-9)'
          ELSE 'letter followed by 4 digits';

 /* composite expression list with simple fields
 FIELD cfname, cname INIT NULL
     CHECK IS NULL AND IS ALPHA
           ELSE 'please enter complete name';
```

2. The fields to be checked contains only vector slices.

   - The predicates ALL, ANY, and ONE must be used for a precise formulation. Behind CHECK and a predicate, a <check cond> can be specified.

```
                                                    |                        |
                                                    |  Form Definition       |
                                                    |________________________|
                                                                             |
   FORM customer.list                                                        |
   LAYOUT                                                                    |
                                                                             |
      _1         _2                                                         |
      _          _                                                          |
      _          _                                                          |
     ...                                                                     |
     ENDLAYOUT                                                               |
                                                                             |
     FIELD 1:cno_old(1..3), 2:cno_new(1..3)                                  |
         CHECK ANY BETWEEN 1000 and 9999 OR IS BLANK                         |
         ELSE 'specify at least one 4 digit number per'                      |
         & ' column';                                                        |
                                                                             |
                                                                             |
```

3. The required condition cannot be formulated with the options described above

   - Behind CHECK an arbitrary <boolean expr> (see Section, "Boolean Expressions") can be specified. A mixture of simple variables and vector slices is permitted as variable list of the fields to be checked. In most cases, however, it will make sense to use this variant only to check a definite field and not a field list.

```
                                                                             |
 FIELD cno CHECK cno > 1000 ...                                              |
           ELSE 'customer numbers start at 1000';                            |
    or                                                                       |
 FIELD cno CHECK ( cno DIV 2 ) * 2 = cno                                     |
           ELSE 'all customer numbers must be even';                         |
                                                                             |
```

Example: Date value check

```
                                                    |                        |
                                                    |   Form Execution       |
                                                    |________________________|
                                                                             |
                        date format         from              to             |
                         ____________________________________                |
                                                                             |
   ..........            mm/dd/yyyy          01/01/2002        06/30/2002     |
                                                                             |
                                                                             |
  ...                                                                        |
  please enter a date with the specified date format and                     |
  within the interval specified                                              |
                                                                             |
                                                                             |
```

```
                                                    |                        |
                                                    |   Form Definition      |
                                                    |________________________|
                                                                             |
```

```
| FORM today.input                                                                |
| LAYOUT LOW=+ PROMPT = .                                                         |
|                       date format         from               to               |
|                      ____________________________________________________      |
|                                                                                 |
|   _today             <datformat          <from_date          <to-date        |
|                                                                                 |
|  ENDLAYOUT                                                                      |
|                                                                                 |
|  FIELD message 'please enter a date with the specified '                        |
|             & 'date format and in the specified interval';                      |
|                                                                                 |
|  FIELD datformat INIT SET  (DATE);                                              |
|  FIELD from_date INIT DATE ( 20020101, yyyymmdd );                              |
|  FIELD to_date INIT DATE ( 20020630, yyyymmdd );                                |
|                                                                                 |
|  FIELD today  SIZE 10                                                           |
|      CHECK                                                                      |
|       DATE (yyyymmdd,today) BETWEEN 20020101 AND 20020630                       |
|          ELSE 'date must be in the specified range'                             |
|            & 'and be noted in the specified format';                            |
|                                                                                 |
|_________________________________________________________________________________|
```

In this example a check is made as to whether a date input (field 'today') is within a certain interval. The test whether a value is in a certain interval is only possible for date values in the internal representation of date values. The function DATE (yyyymmdd, today) converts the variable value 'today' from the format set in the Set parameters into the internal date format.

The subsequent BETWEEN predicate finally checks whether the value is in the specified interval.

Since the DATE function only returns a valid value if the input value has the format expected by the Set parameters, the date format of the date input, too, is automatically checked in this way.

Syntax:

```
<check spec>  ::=  CHECK <check pred>  [ ELSE <msg spec> ]

<check pred>  ::=  <simple var pred>
                |  <vector var pred>
                |  <boolean expr>

 <simple var pred>  ::=  [ <expr> ] <check cond>

 <vector var pred>  ::=  <quant> <check cond>

 <quant>  ::=  ALL | ANY | ONE

 <expr>  ::=  see "Arithmetic Expressions" (5.1.5)

 <boolean expr> ::=  see "Boolean Expressions" (5.1.7)

 <check cond>  ::=  see "Boolean Expressions" (5.1.7)
```

Using DOMAIN Definitions (FIELD/DOMAIN)

If DOMAIN objects are defined, they can be used when defining a form field. The FIELD/DOMAIN specification causes FORM to access the DOMAIN definition in the database and to use it to define SIZE, WIDTH, INIT, and CHECK.

The length specified in the DOMAIN definition under LENGTH has the effect of an explicitly defined FIELD/SIZE specification. The width of the field specified in the DOMAIN definition under COLS has an effect like FIELD/WIDTH. The DEFAULT value of the DOMAIN definition has the same effect as a FIELD/INIT specification with the same value. Apart from that, the RANGE specification of the DOMAIN definition has the same effect as an explicitly defined FIELD/CHECK specification.

Example:

Let a DOMAIN be defined in the database in the following way: (For the definition, the tool component DOMAIN should always be used.)

```
domain name  :  TITLE
length       :  5
data type    :  char
range        :  IN ('Mr','Mrs','Company')
default      :  Company
```

Then...

```
FIELD tit SIZE 5
        INIT 'Company'
        CHECK IN ('Mr','Mrs','Company')
```

is equivalent to...

```
FIELD tit DOMAIN title
```

In some cases it may be desirable that only a part of the attributes declared with the object DOMAIN are used. For this purpose it is possible to use the keywords SIZE, WIDTH, INIT, and CHECK behind the domain name to specify those parts of the DOMAIN definition that should be in effect for this field.

Example:

```
FIELD tit DOMAIN tit ( SIZE, CHECK );
```

has the same effect as:

```
FIELD tit SIZE 5
          CHECK IN ('Mr','Mrs','Company')
```

Syntax:

```
<domain spec>  ::=  DOMAIN <domain name> [ ( <domain spec>,... ) ]

<domain spec>  ::=  SIZE | WIDTH | INIT | CHECK
```

The FIELD/HELP Statement

Calling HELP Forms (FIELD/HELP)

The FIELD/HELP statement provides better support for outputting help information of the program. It is possible to assign a HELP form to each nput field which is shown when the cursor is positioned on the corresponding input field and the key F10 (HELP) is pressed.

The HELP form is automatically positioned on the current form in such a way that the field concerned remains visible as far as possible.

In addition, it is possible to explicitly specify the position and size of the HELP form.

Note:
The following restrictions have to be taken into account for HELP forms:

- In the CONTROL statement, CALL, SWITCH or SWITCHCALL are not permitted.
- Apart from the MESSAGE variable, only local variables can occur in HELP forms.

Help Form Definition

```
HELPFORM customer.helpinfo
LAYOUT HIGH = %


 Customer_no :  5-digit numeric


ENDLAYOUT
FIELD message INIT ’proceed with ENTER’;
```

Form with HELP Call

```
FORM customer.mastercard
LAYOUT

...
                  custno. : _cno
...

ACCEPT( F10=’INFO’, ENTER, F3=’BACK’ );
 FIELD cno
       HELP FORM helpinfo (FRAME);
```

If the key F10 (HELP) is pressed in the form ’mastercard’ while the cursor is on the field ’cno’, the form ’customer.helpinfo’ is displayed.

If the cursor is on a form field for which no HELP option has been defined, the key F10 (HELP)is treated like any other function key.

It is also possible to call a HELPFORM of another program as HELP form (e.g. HELP FORM info.customermaster).

If F1 should be the HELP release key instead of F10, the KEYSWAP (F1 <-> F10) statement has to be used.

Help Messages in the MESSAGE Line (FIELD/MSG)

It is possible to specify a HELP form in the FIELD statement. This form is shown when the key F10 (HELP) is pressed. Often, however, it would suffice to display a brief help text in the system line which could be followed by a more detailed HELP form, if required.

This can be done with the statement FIELDHELPMSG. The statement FIELD a HELP MSG <str expr> causes the specified message to be displayed in the message line when the key F10 (HELP) is pressed. If both types of help information are specified, the message is displayed after pressing the F10 key (HELP), and the form is displayed after pressing the F10 key (HELP) again.

Example:

```
FIELD func
  HELP MSG
     ’The menu items describe the possible actions’
  HELP FORM
     menu_info;
```

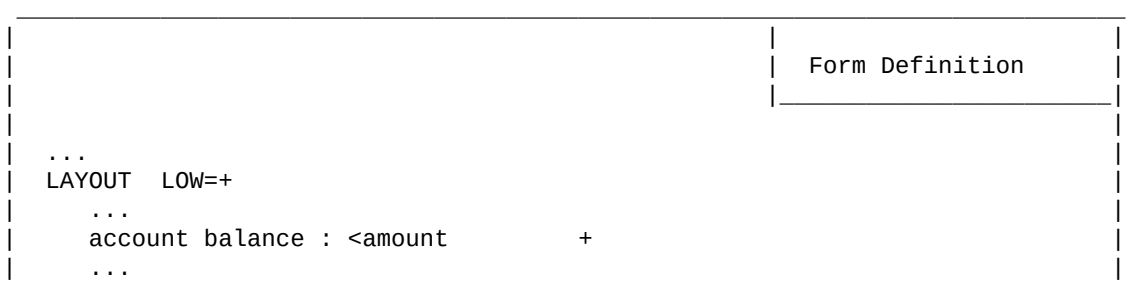
Syntax:

```
<help spec>  ::=  [ HELP MSG <expr> ]
                  HELP FORM [<prog name>.]<module name>
                            [[OPTION[S]] (<form calling options>)]
                            [PARMS (<param>...)]
```

Preparing Output Fields (FIELD/DISPLAY)

The FIELD/DISPLAY statement allows the values of the variables to be formatted for display. Numeric values can be formatted by means of the FORMAT statement and be aligned to the right or left, in addition. Thereby it must be ensured that the FORMAT statement corresponds to the value range of the number to be represented, for otherwise FORMAT returns the NULL value.

Example:



```
| ENDLAYOUT                                                                    |
|                                                                              |
| FIELD amount DISPLAY FORMAT ('+999.999,99');                                 |
| FIELD amount DISPLAY FORMAT ('+999.999,99') RIGHT;                           |
|                                                                              |
|                                                                              |
|______________________________________________________________________________|
```

Alphanumeric values can be prepared in upper (UPPER) or lower (LOWER) cases and aligned to the right (RIGHT) or to the left (LEFT).

Example:

```
 ______________________________________________________________________________
|                                                      |                       |
|                                                      |  Form Definition      |
|                                                      |_______________________|
|                                                                              |
|  ...                                                                         |
|  LAYOUT  LOW=+                                                               |
|     ...                                                                      |
|     Name : <name +                                                           |
|     ...                                                                      |
| ENDLAYOUT                                                                    |
|                                                                              |
| FIELD name DISPLAY UPPER LEFT;                                               |
|                                                                              |
|                                                                              |
|______________________________________________________________________________|
```

Note:
In this version input fields are only output formattedly if they cannot be overwritten at the moment.

Syntax:

```
<display spec>  ::=  DISPLAY <display spec>,...

<display spec>  ::=  FORMAT ( '<char>...' )
                   |  UPPER | LOWER
                   |  RIGHT | LEFT
```

The FIELD/OFFSET Statement

A further option of the FIELD statement is the OFFSET option. Since it is linked very strongly with the fields of the form, it has already been described in Section, "Vector Components with Dynamic Index (FIELD/OFFSET)".

Syntax:

```
<offset spec> ::= OFFSET <expr>
```

Assigning Field Attributes (FIELD/ATTR)

As already described in Section, "Display Attributes (HIGH, LOW, INV, BLK, UNDERL, ATTR1..ATTR16)", a display attribute can be chosen for each form field with which it is to be represented. The attribute character itself takes on the position of a character.

By means of the FIELD/ATTR statement, an attribute can be assigned to each input or output field without an additional character having to be placed before the field in the LAYOUT. The attributes of the text fields, however, can only be modified by means of the control characters.

Example:

Syntax:

```
<attr spec>  ::=  ATTR <attr name>

<attr name>  ::= see "Display Attributes
                     (HIGH, LOW, INV, BLK, UNDERL, ATTR1..ATTR16)" (8.2.7.1)
```

NOINPUT Fields

A frequent application of forms is to choose a certain field in a list of fields by means of the cursor without overwriting the field.

The option NOINPUT ensures that a field defined as input field in the layout cannot be overwritten, but can nevertheless be chosen by the cursor.

Example:

```
LAYOUT

_Z(1..15)
_
_
   ...
   ...
   ...
_

ENDLAYOUT

SPECIALATTR CURSORLINE INV;
FIELD Z(1..15) NOINPUT;
CONTROL CASE $KEY OF
       ENTER : CALL PROC display (Z($CURSOR));
        ...
        ...
```

NOAUTONEXT Fields

Example:

```
 FIELD name NOAUTONEXT
```

This option specifies that the cursor is not automatically positioned on the next input field when the field ’name’ is full.

Syntax:

```
<autonext spec> ::= NOAUTONEXT
```

BEFORE/AFTER FIELD

In addition to the familiar statements INIT, HELP MSG, HELP FORM, CHECK, and DISPLAY the statements BEFORE and AFTER FIELD are available for processing the fields.

BEFORE FIELD and AFTER FIELD introduce a block of SQL-PL statements.

The statements formulated behind BEFORE FIELD are executed when entering the corresponding field.

In practice, mainly guidance texts for the field concerned will be specified here.

Example:

```
FIELD firstname BEFORE FIELD message := ’Firstname of customer’;
```

All other statements permitted in SQL-PL, however, can also be used. The application programmer has all options available, even a REPORT or SQL statement.

For AFTER FIELD, all statements permitted in SQL-PL can be used, as for BEFORE FIELD. The statements formulated behind AFTER FIELD are executed when the field concerned is left and the CHECK condition, if any, has been satisfied.

For AFTER FIELD, also statements such as PICK, PUT, NEXTFIELD etc. defined under <extended compound> can be used, in contrast to BEFORE FIELD.

Syntax:

```
<before field spec>  ::=  BEFORE FIELD <compound>

<after field spec>   ::=  AFTER  FIELD <extended compound>
```

Dividing into Field Groups (GROUP)

In forms, fields can be combined into groups. This makes it easier to assign the fields of a form to different tables. A form without a GROUP statement has the same effect as a form with precisely one GROUP statement in which all the fields of the form are listed.

If a GROUP statement is formulated in a form, FORM expects an assignment to a group for each field. If there are fields that are explicitly listed in GROUP statements, FORM reports an error when storing.

A field group is left when all fields in the group have been processed. Of course, this depends on the individual field statements that have been defined.

Syntax:

```
<group spec>  ::=  GROUP <group name>
                   <field stmt>;...
                   END;
```

BEFORE/AFTER GROUP

Corresponding to the statements BEFORE FIELD, statements BEFORE GROUP and AFTER GROUP can be formulated for the field groups.

Thereby the following restriction applies that a group name may occur at most in one BEFORE GROUP statement and in one AFTER GROUP statement.

Syntax:

```
<before group spec>  ::=  BEFORE GROUP <compound>

<after group spec>  ::=  AFTER GROUP <extended compound>

<extended compound>  ::=  BEGIN <extended stmt>;... END <extended stmt>

<extended stmt>  ::=  <stmt>
                    |  <page stmt>
                    |  <pick stmt>
                    |  <put stmt>
                    |  <nextfield stmt>
                    |  <nextgroup stmt>
```

The BEFORE GROUP statement of an arbitrary field group is executed when the field group is entered by entering one of its fields. The BEFORE GROUP statement of the first field group is executed at the beginning of the form interpretation.

After executing the AFTER FIELD statement of a group field, the AFTER GROUP statement is executed when leaving the field by means of a function key or when branching to a field of another group by means of the cursor key or when leaving the form.

In the case of a form with field groups, pressing a function key does not leave the form but the group of the field, branching to the next group. If nothing else has been defined, the form is left at the end of the last group.

Ignoring the Input Check (IGNORE)

With the IGNORE statement it can be determined that the input check is suppressed in certain situations.

Example:

If the condition behind WHEN applies, the check for the variables listed behind IGNORE is suppressed. Several IGNORE statements with various conditions are permitted.

A variable can occur simultaneously in several FIELD statements with CHECK condition and in several IGNORE statements.

In this example, the following IGNORE variant in a brief form can also be formulated:

```
IGNORE ALL WHEN $KEY = F3
```

If the input check is to be suppressed for all fields except ’cno’ and ’cname’, the following variant is possible:

```
IGNORE ALL EXCEPT cno, cname
         WHEN $KEY = F3
```

Syntax:

```
<ignore spec>  ::= IGNORE  <ignore field spec>  WHEN  <boolean expr>

<ignore field spec>  ::=  <field name>,...
                        |  ALL
                        |  ALL EXCEPT <field name>,...
```

Initializing the Cursor Position (MARK)

When a form is called, the cursor is normally positioned on the first input field of the form. If the form does not have any input fields, FORM positions the cursor in the top left corner of the form. If the cursor is to be positioned to an input field other than the first, the MARK statement within the form can be used to position the cursor, in addition to the call option MARK.

The MARK option specified in the call overrides the MARK statement in the form.

Example:

```
 ENDLAYOUT
...
 MARK ( 3 );
```

Syntax:

```
<mark stmt> ::= MARK (<expr>) | MARK (<variable>)
```

Key Activation (ACCEPT)

When calling a form, FORM accepts the ENTER key (or RETURN key) as default release keys.

The keys F1 to F12 (HELP, UP , DOWN) that can be addressed beyond these must be explicitly activated by means of the ACCEPT statement.

An ACCEPT statement in the form definition can be used to define which release keys are accepted in this form. Moreover, an assignment of the keys can be specified in this way.

When the ACCEPT option is used, the line for the key assignment appears automatically. The key assignment line appears above the message line. When designing the form, this line must therefore be kept empty.

The ACCEPT statement can be located at any place of the form processing part. But there may be one ACCEPT statement only.

```
                                  | Form Definition |
FORM customer.mastercard...
...
 ENDLAYOUT
```

```
| ACCEPT (F3='BACK', F4='ENTRY', F10= INFO');
```

If the user uses a key not contained in the ACCEPT list, FORM refuses it with a default message.

When calling the form, keys defined within the form and their assignment can be overridden (see Section 8.4.3; "Overriding Keys (ACCEPT)").

```
                                                    |                        |
                                                    |       Call             |
                                                    |________________________|
                                                    |                        |
                                                    |                        |
   CALL FORM mastercard OPTIONS( ACCEPT(F4,F10) );  |                        |
                                                    |                        |
____________________________________________________|________________________|
```

If ACCEPT() is specified as call option, the form returns control without expecting a user interaction.

As can be seen from the syntax description and Section, "Overriding Keys (ACCEPT)", further hard keys can be activated.

Syntax:

```
<accept stmt>  ::=  ACCEPT  ( <key spec>,... )

<key spec>   ::=  ENTER
                |  <basic key> [ = <key label>]
                |  <additional hardkey>

<key label>   ::=  <char sequence>  maximum 8 characters

<basic key>  ::=  F1  | F2  | F3  | F4  | F5  | F6
                |  F7  | F8  | F9F10 | F11 | F12 | HELP
                |  UP | DOWN

<additional hardkey>  ::=  CMDKEY | ENDKEY | UPKEY | DOWNKEY
  |
 RIGHTKEY | LEFTKEY
```

The KEYSWAP Statement

The firm assignment between key literals and the keys of the keyboard described in the last section can be changed. This may be useful when the assignment between the keys and the functions of a completed SQL-PL application has to be adapted to a customer’s requirements.

Example:

```
                                  | Form Definition |
...
 ACCEPT ( ENTER, F10='INFO', F1='ENTRY',
```

```
|           F2='SEARCH', F3='END' );
```

Resulting key menu:

```
1=ENTRY 2=SEARCH 3=END 10=INFO
```

With the following KEYSWAP option precisely two functions are swapped and HELP is released by the F1 key.

Resulting key menu:

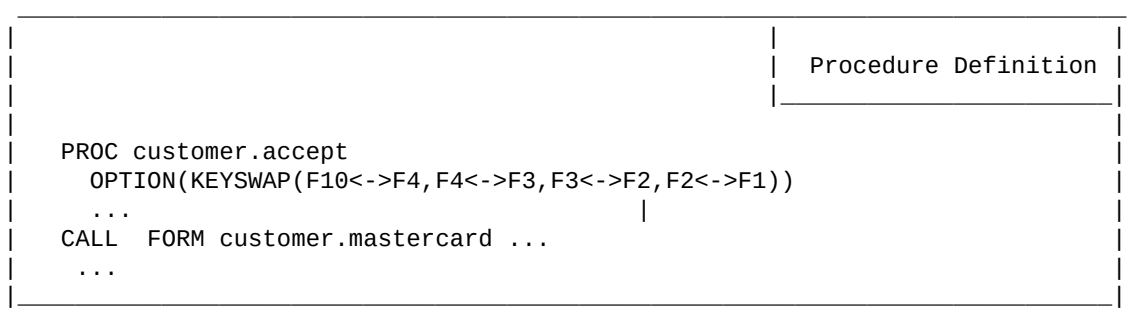
```
1=INFO 2=SEARCH 3=END 10=ENTRY
```

With the following KEYSWAP option the functions are swapped cyclically and HELP is released by the F1 key.

Resulting key menu:

```
1=INFO 2=ENTRY 3=SEARCH 4=END
```

KEYSWAP can be specified behind the form layout or as an option of an SQL-PL procedure behind the procedure name.



The KEYSWAP assignment applies as long as the program runs or until the next KEYSWAP statement.

Syntax:

```
<keyswap stmt> ::= KEYSWAP ( <key pair>,... )

<key pair> ::= <basic key> <swap sign> <basic key>

<basic key> ::= see "Key Activation (ACCEPT)" (8.3.6)

<swap sign> ::= '<->'
```

Leaving the Form (RETURNONLAST)

A form is normally left by pressing RETURN or a function key. The statement RETURNONLAST causes the form to be left when the cursor is on the last field of the form and this field is left by NEXTFIELD, graphics/sqlpl1.gif or graphics/sqlpl2.gif .

If the last field is not a NOAUTONEXT field, then filling it with writing causes the field and thus the form to be left.

Example:

```
...
ENDLAYOUT
RETURNONLAST;
...
```

Syntax:

```
<returnonlast stmt> ::= RETURNONLAST
```

Scrolling Support (AUTOPAGE, PAGE)

Scrolling in forms is necessary when a form is larger than the screen segment in which it is displayed. There are three ways of scrolling in forms:

1. The form is called from an SQL-PL procedure that outputs the corresponding screen segment by means of the FORMPOS option.

```
PROC customer.display
...
CASE $KEY OF
  UP       : firstline := firstline - 15;
  DOWN     : firstline := firstline + 15;
END;

CALL FORM list ( FORMPOS ( firstline, 1 ) );
```

2. One uses the statements PAGE UP and PAGE DOWN, PAGE LEFT and PAGE RIGHT in the CONTROL statement of the form (see Section, "The PAGE Statement").

```
CONTROL
   CASE $KEY OF
     UP    : PAGE UP;      (* one page up  *)
     DOWN  : PAGE DOWN n;  (* n lines down   *)
               ...
END;
```

3. One uses the statement AUTOPAGE which assigns the scrolling functions automatically to the keys and supports scrolling in all directions according to the size of the screen segment.

```
...
ENDLAYOUT
ACCEPT ( ... );
AUTOPAGE;
FIELD ...           ...
...
```

FORM sets definite keys for scrolling with AUTOPAGE. If these are already set by the program, FORM cannot use these keys for scrolling. The function is thus disabled.

For keyboards with hard scrolling keys, these are activated by FORM for scrolling; this is true of all scrolling directions.

For all other keyboards, downward scrolling is released by DOWN (= F12) and upward scrolling by UP (= F11). For left/right scrolling, FORM implicitly distinguishes two states for keyboards without hard scrolling keys:

- If the screen segment is so large that the left or the right side of the form is visible, FORM provides the F9 key for alternate scrolling.
- If the screen segment is so small that horizontally scrolling can be done several times, FORM simultaneously provides the two keys F8 and F9 to scroll to the left or right resp.

If the assignment of the keys does not correspond to the application programmer's taste, it can be changed by means of the KEYSWAP statement.

For keyboards with nine function keys only, there must be hard scrolling keys which FORM will automatically activate for AUTOPAGE. In forms with an explicit MESSAGE variable or forms with an action bar, scrolling support by AUTOPAGE is not possible.

Syntax:

```
<autopage stmt> ::= AUTOPAGE
```

Header Lines and Bottom Lines (HEADERLINES, BOTTOMLINES)

In forms which are to be scrolled, the first or last lines of the form layout can be defined as frame lines. These frame lines are always output at the same position on the screen, when scrolling in the form.

```
...
ENDLAYOUT
HEADERLINES (2);
BOTTOMLINES (1);
...
```

The specification of HEADERLINES or BOTTOMLINES defines the number of header lines or bottom lines which are to be displayed in the first or last lines of the screen window. The remaining area of the form can be scrolled (see Section, "Scrolling Support (AUTOPAGE, PAGE)").

The header and bottom lines are not printed out, when calling the form with PRINT option.

If an action bar is defined in the form, the number of header lines is implicitly set to the line in which the action bar is output, i.e. the header lines comprise the area of the form from the first line up to the action bar inclusive. The explicit definition of header lines overrides the implicit mechanism.

Syntax:

```
<headerlines stmt> ::= HEADERLINES (<expr>)

<bottomlines stmt> ::= BOTTOMLINES (<expr>)
```

Situation-dependent Display Attributes (SPECIALATTR)

Form fields can be assigned a desired display attribute in the form layout or by the FIELD/ATTR statement, and the display attribute of a form field can be overridden when calling the form.

By means of the form processing statement SPECIALATTR, a field can also be displayed with a certain display attribute that depends on the situation.

Example:

The statement SPECIALATTR CHECK causes the input field to be represented with the specified display attribute, if the CHECK condition was not satisfied.

Example:

```
SPECIALATTR INPUT inv;
```

The statement SPECIALATTR INPUT causes the currently active input field to be displayed with the specified display attribute. If a multi-line input field is concerned, then all lines of the field are represented with this attribute.

Example:

```
 SPECIALATTR MSG ATTR16
```

The statement SPECIALATTR MSG causes the automatically displayed message line to appear with the desired display attribute. If the statement SPECIALATTR MSG is not specified, the message line is displayed with the attribute preset by the system (ATTR5).

Example:

```
 SPECIALATTR CURSORLINE ATTR13
 or
 SPECIALATTR CURSORLINE (5,75) INV
```

The statement SPECIALATTR CURSORLINE causes the form line in which the cursor is positioned to be displayed with the defined attribute. The form line appears as a bar when the selected logical attribute is either defined as ’inverse’ or its background color is different from the background of the form.

The start and end position of the bar is determined by the left and right margin of the form, if not otherwise explicitly specified. If the form is output in a window, the bar is restricted by the form’s window size.

The SPECIALATTR statement applies to all fields of the form and can only be specified once for each form. Thus the above examples can be defined in a SPECIALATTR statement as follows:

```
 SPECIALATTR INPUT underl
             CHECK inv
             MSG ATTR12;
```

Syntax:

```
<special attr stmt>  ::=  SPECIALATTR  [ INPUT <attr name> ]
                                       [ CHECK <attr name> ]
                                       [ MSG <attr name> ]
                                       [ CURSORLINE [ (<first pos>,  <last pos>) ]
                                                      <attr name> ]

 <attr name>  ::=  see "Display Attributes
                   (HIGH, LOW, INV, BLK, UNDERL, ATTR1..ATTR16)" (8.2.7.1)
```

Displaying the Input Mode (INSERTMODE)

The statement INSERTMODE can be used to display in which input mode (overwrite or insert) the keyboard is.

If the input mode is set to insert by pressing the INSERT key, the expression defined by LABEL is output (default: 'INSERT') in the attribute ATTR (default: ATTR5) at the position specified by POS. When switching back to the overwrite mode, the label is excluded from the display.

The position POS must be specified in the statement INSERTMODE, the specification of LABEL and ATTR is optional. The INSERT label is output in the specified length up to a maximum of eight characters.

Example:

```
 ENDLAYOUT
   ...
 INSERTMODE ( POS ( 20,3),
                    LABEL ( 'INSERT' ),
                    ATTR  ( ATTR7 ) );
```

Syntax:

```
<insertmode stmt>  ::=  INSERTMODE ( POS (<expr>,<expr>)
                                     [, LABEL ( <expr> ) ]
                                     [, ATTR ( <attr name> ) ] )
```

Controlling the Dialog Sequence (CONTROL, CASE)

In interactive applications the dialog sequence between the user and the program is usually controlled by both dialog partners:

- by the user by selecting certain alternatives in the displayed forms, by pressing function keys or making input ...
- by the program by acting to the user's behavior with its own programmed sequential control.

To program this sequential control, SQL-PL provides the following facilities:

1. The procedure takes over control with its own control structures. It calls the forms and acts to the input.
2. The form assumes the control and calls procedures or other forms depending on the user's input.

3. Combination of a) and b).

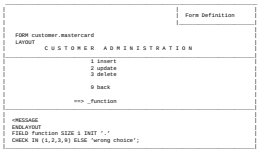
Example:

The form with a function menu calls procedures which in turn control the sequence of the individual functions.

Selection menus are the typical case in which it seems to be useful to control the dialog by the form. It is usual that, after processing the chosen alternative, the user is returned to the selection menu.

SQL-PL supports the programming of such selection menus by the CONTROL statement in the form definition.

Example:



An SQL-PL procedure could call this form in a REPEAT loop and further procedures, depending on the chosen function.

```
PROC   customer.start;
   REPEAT
     CALL FORM mastercard;
     CASE function OF
        1 : CALL PROC   insert;
        2 : CALL PROC   update;
        3 : CALL PROC   delete;
     END
   UNTIL function = 9
```

Precisely the same behavior is achieved by the following CONTROL statement in the form definition:

```
|                                                  |                        |
|                                                  |  Form Definition       |
|                                                  |________________________|
|                                                                           |
|   FORM customer.mastercard                                                |
|   ...                                                                     |
|   ENDLAYOUT                                                               |
|   FIELD function SIZE 1 INIT '.'                                          |
|      CHECK IN (1,2,3,9) ELSE 'Wrong choice';                              |
|                                                                           |
|   CONTROL                                                                 |
|       CASE function OF                                                    |
|          1 : CALL PROC insert;                                            |
|          2 : CALL PROC update;                                            |
|          3 : CALL PROC delete;                                            |
|          9 : RETURN;                                                      |
|       END;                                                                |
|                                                                           |
|___________________________________________________________________________|
```

The CASE statement has the same structure as in procedures. The same statements are permitted, namely, arbitrary SQL-PL statement sequences and, in addition, the special FORM statements PICK/PUT, PAGE UP and PAGE DOWN, PAGE LEFT and PAGE RIGHT. The FORM statements for scrolling have been described in the previous section, the PICK/PUT statements will be described in the next section.

In the following, the various CALL statements are described. It must be noted that after calling another procedure or form at first the INIT phase is automatically performed in the calling form. Only then is the statement following the CALL statement executed.

- * CALL ...
- * SWITCH ...
- * SWITCHCALL ...
- * RETURN
- * STOP ...
- * any SQL-PL statement
- * PICK and PUT statement
- * PAGE statement

In the first case (CALL), the form calls the relevant module like a subprogram. After the module has been executed, the form is redisplayed on the screen with the same options which were previously specified for the call, and the INIT statements are run through once again.

In the second case (SWITCH), the running program is terminated and the interactive dialog with the called program is continued.

In the third case (SWITCHCALL), the form calls the module or another program. The variable values of the calling program are kept. After executing the called program, the form is redisplayed on the screen with the same options which were previously specified for the call, and the INIT statements are run through once again.

In these three cases (CALL, SWITCH, and SWITCHCALL), parameter specified, and for form calls, options (see Sections, "Parameters for CALL, SWITCH, and SWITCHCALL" and "Options for Form Calls").

In the fourth case (RETURN), the form returns control to the environment from which it was called. This is the default behavior, if no CONTROL statement is specified.

In the fifth case (STOP), the program is terminated immediately.

The CONTROL statement can only be specified once at the end of the form processing part.

Syntax:

```
<control spec>  ::=  CONTROL  <control case spec>

<control case spec>  ::=  CASE <expr> OF <control case> [ <else case> ]  END

<else case>  ::=  OTHERWISE <control action>

<control case>  ::=  <value spec> : <control action> [ ; <control case> ]

<control action>  ::=  <extended stmt>
                       see "BEFORE/AFTER GROUP" (7.3.3)
```

The PAGE Statement

As described in Section, "Scrolling Support (AUTOPAGE, PAGE)", there are three different ways of scrolling in forms. One way is the statements PAGE UP and PAGE DOWN, PAGE LEFT and PAGE RIGHT which can only be used within the statements CONTROL, AFTER GROUP and AFTER FIELD.

Example:

```
|                                                                              |
|  ...                                                                         |
|  LAYOUT                                                                      |
|  ...                                                                         |
|  ...                                                                         |
|  ENDLAYOUT                                                                   |
|                                                                              |
|  CONTROL                                                                     |
|     CASE $KEY OF                                                             |
|       UP    : PAGE UP;       (* one page up  *)                             |
|        DOWN  : PAGE DOWN n;  (* n lines down   *)                           |
|        F4    : PAGE LEFT 5;  (* 5 columns to the left  *)                    |
|        F5    : PAGE RIGHT;   (* width of window to the right *)             |
|                ...                                                           |
|      END                                                                     |
|______________________________________________________________________________|
```

Syntax:

```
<page stmt>  ::=  PAGE UP    [<expr>]
                |  PAGE DOWN  [<expr>]
                |  PAGE LEFT  [<expr>]
                |  PAGE RIGHT [<expr>]
```

PICK/PUT Mechanism in Forms (PICK/PUT/AUTOPUT)

The PICK/PUT statements can be used to define HELP forms, for example, from which the example values for input fields can be picked out.

PICK and PUT may only occur in the CONTROL block, in AFTER FIELD and AFTER GROUP statements.

Function of PICK

- without argument:

The value of the field on which the cursor is positioned is stored in the PICK buffer. If the cursor is not placed on an input field, an error message is output.

- with argument:

  The argument is stored in the PICK buffer. Usually, the argument is formulated depending on the cursor position. If the cursor is not placed on an input field, an error message is output.

Example:

```
HELPFORM tpick.test
LAYOUT
_1  <@menu(1)
_   <@menu(2)
_   <@menu(3)
ENDLAYOUT
FIELD  1:@choice(1..3);
FIELD  @menu(1) INIT 'Insert';
FIELD  @menu(2) INIT 'Update';
FIELD  @menu(3) INIT 'Delete';
ACCEPT ( F3='end' );
CONTROL CASE $KEY OF
          ENTER: PICK(@menu($CURSOR));
          ELSE : RETURN;
        END;
```

Function of PUT

- without argument:

  The PICK buffer content is stored as value of the field on which the cursor is positioned. If the cursor is not placed on an input field, an error message is output.

- with argument:

  The PICK buffer content is stored as value of the given variable. This variant is required when an output field is to be set to a picked value.

Using PICK and PUT

- PICK and PUT are used in the CONTROL statement.

Example:

```
FORM tpick.test
LAYOUT
_1           _2
_            _
_            _
_            _
ACCEPT ( enter, f5='PICK' );
FIELD 1:title(1..4) HELP FORM p_title ( frame );
FIELD 2:city(1..4) HELP FORM p_city ( frame );
CONTROL
```

```
|    CASE $KEY of                                                              |
|      F5: PUT;                                                                |
|    END;                                                                      |
|                                                                              |
|______________________________________________________________________________|
```

- PICK and PUT are only allowed in forms.

To simplify the use of PICK in forms, a form can be called with the AUTOPUT option. This has the effect that, after leaving the form, the picked value is automatically assigned to the variable of the field on which the cursor was positioned before calling the form.

Example:

```
 ______________________________________________________________________________
|                                      |                                       |
|                                      |  Using AUTOPUT when calling a form    |
|                                      |  by means of the CONTROL statement    |
|______________________________________|_______________________________________|
|                                                                              |
|  ...                                                                         |
|  ENDLAYOUT                                                                   |
|  ACCEPT ( ENTER, F5='PICK' );                                                |
|                                                                              |
|  CONTROL CASE $KEY OF                                                        |
|       F5 : CALL FORM selection OPTION ( AUTOPUT );                           |
|       ENTER : ...                                                            |
|  END;                                                                        |
|______________________________________________________________________________|
```

If no value has been selected with PICK, the AUTOPUT option does not output an error message, in contrast to the explicitly release statement.

In this way, the PUT statement is no longer necessary within the CONTROL statement and the final user has no longer to press the PUT key in addition. It suffices to choose the value with PICK and to leave the form. Then the value appears in the appropriate field.

Example:

```
 ______________________________________________________________________________
|                                      |                                       |
|                                      | Using AUTOPUT when calling a          |
|                                      | HELP form                             |
|______________________________________|_______________________________________|
|                                                                              |
|  FORM  tpick.m1                                                              |
|  LAYOUT                                                                      |
|  _1           _2                                                             |
|  _            _                                                              |
|  _            _                                                              |
|  _            _                                                              |
|  ACCEPT ( enter, f5='PICK' );                                                |
|  FIELD 1:title(1..4) HELP FORM p_title ( FRAME, AUTOPUT );                   |
|  FIELD 2:city(1..4) HELP FORM p_city ( FRAME, AUTOPUT );                     |
|                                                                              |
|______________________________________________________________________________|
```

The AUTOPUT option can also be used with a variable as argument. This is useful, when the value accepted with PICK is not determined for the calling form but is to be passed from the calling form as a parameter.

Syntax:

```
<pick stmt> ::= PICK [(<expr>)]

<put stmt> ::= PUT [(<simple var>)]
```

The NEXTFIELD Statement

The order in which the fields are executed is determined by the user by cursor movements. The NEXTFIELD statement which can occur in the place of any SQL-PL statement can be used to explicitly alter the processing sequence.

Example:

```
 _____________________________________________________________________________
|                                                                             |
|                                                                             |
|                                                                             |
| LAYOUT                                                                      |
|    Accno       : _cno                                                       |
|    Name        : _name                                                      |
|    Firstname   : _firstname                                                 |
|                                                                             |
|    Account balance : _account                                               |
|  ENDLAYOUT                                                                  |
|  GROUP a                                                                    |
|    FIELD custno                                                             |
|       BEFORE FIELD                                                          |
|         message := 'customer number, if known'                              |
|       AFTER FIELD                                                           |
|          BEGIN                                                              |
|          SQL ( SELECT DIRECT name, firstname INTO :name, :firstname         |
|                FROM customer KEY cno = :cno );                              |
|          NEXTFIELD account;                                                 |
|          END;                                                               |
|    FIELD name ...                                                           |
|    FIELD firstname ...                                                      |
|  END;                                                                       |
|                                                                             |
|_____________________________________________________________________________|
```

The field name used in the NEXTFIELD statement must denote a field in the form. This is especially true of the usage of the FIELD/OFFSET option. In this case not the dynamic index of the variables is important for the NEXTFIELD statement, but only the vector index used in the layout of the form.

The NEXTFIELD statement defines the field which is to be the subsequent field. After executing the AFTER FIELD statement the program explicitly branches to this field specified as the subsequent field - independently of the key that has been pressed.

Syntax:

```
<nextfield stmt> ::= NEXTFIELD <field name>
```

The NEXTGROUP Statement

The order in which the groups are executed is determined by the user by cursor movements. The NEXTGROUP statement allows this sequence to be altered, e.g. according to a condition. As the NEXTFIELD statement, the NEXTGROUP statement can be located anywhere.

The group name used in the NEXTGROUP statement must be defined with the GROUP statement in the same form. Otherwise, FORM reports a translation error when storing.

Syntax:

```
 <nextgroup stmt> ::= NEXTGROUP <group name>
```

The SCROLLFIELD Statement

The SCROLLFIELD statement can be used in any form to move vector slices. The SCROLLFIELD statement has to be defined within the AFTER FIELD statement for every vector slice with OFFSET option which is defined as input field in the layout and which is to be moved by pressing the cursor keys, PAGEUP and PAGEDOWN keys.

The SCROLLFIELD statement acts according to the keys and modifies the value of the OFFSET variable to be specified as argument. Several adjacent vector slices can be moved with one SCROLLFIELD statement, if these vector slices depend on the same OFFSET variable.

Example:

```
 _____________________________________________________________________________
|                                                                             |
|  FORM customer.list_of_names                                                |
|  LAYOUT GRAPHIC=* LOW =+                                                    |
|  _name(1..5)         <firstname (1..5)                                      |
|  _                   <                                                      |
|  _                   <                                                      |
|  _                   <                                                      |
|                                                                             |
|  ENDLAYOUT                                                                  |
|  SPECIALATTR INPUT INV;                                                     |
|  ACCEPT ( ENTER,  UPKEY, DOWNKEY );                                        |
|  FIELD name(1..5) OFFSET x                                                  |
|        AFTER  FIELD                                                         |
|           SCROLLFIELD ( x );                                                |
|  FIELD firstname(1..5)  OFFSET x;                                           |
|_____________________________________________________________________________|
```

In this example the vector ’name’ is moved by means of the SCROLLFIELD statement and, consequently, the output vector ’firstname’ as well.

Example: Using a SCROLLFIELD statement for several input vector slices

```
 _____________________________________________________________________________
|                                                                             |
|  FORM customer.list_of_names                                                |
|  LAYOUT GRAPHIC=* LOW =+                                                    |
|  _name(1..5)         _firstname (1..5)                                      |
|  _                   _                                                      |
```

```
|  _                   _                                                  |
|  _                   _                                                  |
|  _                   _                                                  |
|                                                                         |
|  ENDLAYOUT                                                              |
|  SPECIALATTR INPUT INV;                                                 |
|  ACCEPT ( ENTER,  UPKEY, DOWNKEY );                                     |
|  FIELD name(1..5)                                                       |
|          OFFSET x                                                       |
|          AFTER  FIELD SCROLLFIELD ( x );                                |
|  FIELD firstname(1..5)                                                  |
|          OFFSET x;                                                      |
|          AFTER FIELD SCROLLFIELD ( x );                                 |
|_________________________________________________________________________|
```

Example: Abbreviated notation of the last example

```
 _________________________________________________________________________
|                                                                         |
| ...                                                                     |
|  FIELD name(1..5), firstname(1..5)                                      |
|           OFFSET x                                                      |
|           AFTER FIELD SCROLLFIELD ( x );                                |
|_________________________________________________________________________|
```

When the cursor is placed on the last or first field of the vector slice and the graphics/sqlpl1.gif or graphics/sqlpl3.gif key has been pressed, the content of the vector slice is moved one entry by the SCROLLFIELD statement. The content of the vector slice is moved n entries, if the PAGEUP or PAGEDOWN key is pressed. The number n corresponds to the number of vector slice elements (in the example there are five elements).

The cursor moves to the first (last) field, when it is placed on the last (first) field and the TAB- (BACKTAB-) key is pressed.

As first argument, the SCROLLFIELD statement expects the variable which was specified with the OFFSET option. Without OFFSET option the SCROLLFIELD statement has not effect.

In the last example both input vectors are moved simultaneously. When using the SCROLLFIELD statement in this way, care must be taken that the vector slices have the same number of elements in the layout; otherwise unexpected outputs may occur.

As second - optional - argument of the SCROLLFIELD statement, the maximum number of value specified. If the second argument is not specified, the SCROLLFIELD statement assumes that the end of the list is reached with the 255th vector element.

As third optional argument, the number of lines can be specified which are to be used as scrolling unit for the PAGEUP or PAGEDOWN key. The scrolling unit can also vary, e.g., according to the window size.

If a user wants to position directly to the end or to the beginning within the entries, the keys serving this purpose can be defined in the second parentheses of the SCROLLFIELD statement behind the keywords TOPKEY and BOTTOMKEY.

Example: Using all parameters allowed for SCROLLFIELD

```
 ____________________________________________________________________________
|                                                                            |
| FORM selection.list  PARMS ( auswahl(), max_number, length )               |
| ...                                                                        |
| FIELD selection(1..5)                                                      |
|          OFFSET x                                                          |
|          AFTER  FIELD  SCROLLFIELD ( x, max_number, $screenlns             |
|                                    ( TOPKEY F7, BOTTOMKEY F8 );            |
|____________________________________________________________________________|
```

Syntax:

```
<scrollfield stmt>  ::=  SCROLLFIELD ( <offset var> [, <max count>
                                                    [, <page count> ] ] )
                         [ ( TOPKEY <basic key>, BOTTOMKEY <basic key> ) ]

 <offset var>  ::=  <variable>

 <max count>  ::=  <expr>

 <page count>  ::=  <expr>
```

The KEYS Statement

In FORM particular functions are assigned to a series of keys. For example, the HELP key releases the execution of the FIELD/HELP statement. For the AUTOPAGE statement, too, the scrolling functions are assigned to definite keys.

For ergonomic reasons it is often desired to assign one function to several keys. This can also be helpful in the case of programs which are intended to be used on different keyboards.

The KEYS statement allows one or more keys to be assigned as release keys to the functions HELP, MENU, UP, DOWN, LEFT, and RIGHT. The function MENU designates the key which releases switching between action bar and form.

Example:

```
KEYS ( UP   = F7  |  UPKEY,
       DOWN = F8  |  DOWNKEY,
       HELP = F1  |  HELPKEY )
```

Syntax:

```
<keys stmt>  ::=  KEYS ( <function key spec>,... )

<function key spec> ::=  <key function> = <key> [ | <key> ... ]

<key function>  ::=  HELP | MENU | UP | DOWN | LEFT | RIGHT

<key>  ::=  <basic key>
          |  <additional hardkey>   see "Key Activation (ACCEPT)" (8.3.6)
```

Options for Form Calls

When calling a form, numerous form settings can be overridden by call options. The individual call options are explained in the following sections.

Syntax:

```
<form calling option>  ::=  <noinit option>
                          |  <mark option>
                          |  <accept option>
                          |  <attr option>
                          |  <clear option>
                          |  <screensize option>
                          |  <screenpos option>
                          |  <formpos option>
                          |  <frame option>
                          |  <background option>
                          |  <restore option>
                          |  <input option>
                          |  <noinput option>
                          |  <action option>
                          |  <print option>
```

This section covers the following topics:

- Suppressing the INIT Phase (NOINIT)
- Cursor Control (MARK, $CURSOR)
- Overriding Keys (ACCEPT)
- Overriding Display Attributes (ATTR)
- The Window Options SCREENPOS, SCREENSIZE, and CLEAR
- Form Segments (FORMPOS)
- Automatic Framing by FRAME
- Superimposing Forms (BACKGROUND)
- Restoring the Form Background (RESTORE)
- Form Output via Printer (PRINT)
- Overriding the Active Input Fields (INPUT/NOINPUT)
- Activating the Action Bar (ACTION)

Suppressing the INIT Phase (NOINIT)

The option NOINIT causes the FIELD/INIT statements in the form to become ineffective for this call.

```
 _____________________________________________ _____________________________
|                                             |                             |
|   CALL FORM mastercard                      | form call without           |
|                 OPTIONS(NOINIT);            | initialization from the     |
|                                             | form definition             |
|_____________________________________________|_____________________________|
```

Syntax:

```
<noinit option> ::= NOINIT
```

Cursor Control (MARK, $CURSOR)

When calling a form, the cursor is implicitly positioned on the beginning of the first input field.

The calling module can use the MARK option to position the cursor on the n-th input field - in the sequence from the top left to the bottom right.

Example:

```
 _____________________________________________|_____________________________
|                                             |                             |
|   CALL FORM mastercard                      |  cursor is positioned on the |
|                 OPTIONS(MARK(3))            |  third input field          |
|_____________________________________________|_____________________________|
```

Alternatively, the cursor can be positioned to the field of a certain variable. The variable must be global and it must be defined as an input field in the form.

Example:

```
 _____________________________________________|_____________________________
|                                             |                             |
|   CALL FORM mastercard                      |  cursor is positioned on the |
|                 OPTIONS(MARK( cname ));     |  field cname                |
|_____________________________________________|_____________________________|
```

If the variable of the MARK option does not occur in the called form, its value is interpreted as the number of the input field and the cursor is positioned on the input field with this number.

Thus the number of the input field can be specified as a constant or a variable for MARK. If the variable has the NULL value or if there is no field with this number, the cursor is placed on the first input field.

On the other hand, after a form has been called, the calling procedure can check with the $CURSOR variable on which input field the cursor was last positioned.

```
 _____________________________________________|_____________________________
|                                             |                             |
|                                             |  Form Definition            |
|                                             |_____________________________|
|                                                                           |
|   FORM customer.mastercard                                                |
|   ...                                                                     |
|   ENDLAYOUT                                                               |
|   FIELD function SIZE 1 INIT '.'                                          |
|      CHECK IN (1,2,3,9) ELSE 'Wrong choice';                              |
|                                                                           |
|   CONTROL                                                                 |
|       CASE function OF                                                    |
|          1 : CALL PROC insert;                                            |
|          2 : CALL PROC update;                                            |
|          3 : CALL PROC delete;                                            |
```

```
|          9 : RETURN;                                                    |
|       END;                                                              |
|                                                                         |
|_________________________________________________________________________|
```

```
 ________________________________________________________________________
|                                                    |                   |
|                                                    |   SQL-PL Routine  |
|                                                    |___________________|
|                                                                        |
|   FORM customer.mastercard                                             |
|                                                                        |
|   IF $KEY IN (HELP, F1)                                                |
|      THEN BEGIN                                                        |
|                                                                        |
|   ------> IF $CURSOR = 5 /* ZIP                                        |
|           THEN CALL FORM zip_help ...                                  |
|                                                                        |
|________________________________________________________________________|
```

If the cursor was positioned on an input field, $CURSOR returns the sequential number of the input field in the form. Otherwise, it returns the NULL value.

For MARK as well as for $CURSOR it must be noted that only the input variables count for numbering.

Syntax:

```
<mark option> ::= MARK (<expr>) | MARK (<variable>)
```

Overriding Keys (ACCEPT)

The ACCEPT statement can be used to determine the release keys which are to be accepted by the form.

ACCEPT can be specified in the form definition and as an option when calling the form.

As call option, ACCEPT overrides the definition in the form.

The following example illustrates this:

```
 ________________________________________________________________________
|                                                    |                   |
|                                                    |    Form Definition|
|                                                    |___________________|
|                                                                        |
|   LAYOUT                                                               |
|   ...                                                                  |
|   ENDLAYOUT                                                            |
|                                                                        |
|   ACCEPT ( ENTER, F1='ENTRY',                                          |
|                   F2='SEARCH', F3='END' );                             |
|                                                                        |
|________________________________________________________________________|
```

Call:

```
CALL FORM ...( ACCEPT (ENTER, F1, F2, F3='BACK'))
```

Resulting key menu:

```
1=ENTRY 2=SEARCH 3=BACK
```

HELP, UP, and DOWN activate the keys with the labels PF10 or F10 for HELP, PF11 or F11 for UP, and PF12 or F12 for DOWN, or, if they do not exist, the hard key HELP for HELP and the usual scrolling keys for UP and DOWN .

The ACCEPT option causes the key assignment to be displayed automatically. This key assignment line is always displayed via the message line, i.e. the display area available for the form layout is reduced by this one line.

In addition to the keys already described, which FORM assumes independently of the type of hardware, further keys can be used (see the description of the keyboard in the "User Manual Unix" or "User Manual Windows") that depend on the installation.

As a maximum the following additional key literals can be used: ENDKEY, CMDKEY, LEFTKEY, RIGHTKEY, HELPKEY, UPKEY, and DOWNKEY. These key literals can be used like the other key literals in the ACCEPT statement and for requesting $KEY.

However, the same restrictions apply to them as for the key literal ENTER, since these key literals address hard keys:

- No key labels can be defined for the keys CMDKEY, ENDKEY, RIGHTKEY, LEFTKEY, HELPKEY, UPKEY, DOWNKEY, and ENTER.

- The keys CMDKEY, ENDKEY, RIGHTKEY, LEFTKEY, HELPKEY, UPKEY, DOWNKEY, and ENTER cannot be used in the KEYSWAP statement.

Moreover, it must be noted that the usage of the key literals CMDKEY, ENDKEY, RIGHTKEY, LEFTKEY, HELPKEY, UPKEY, and DOWNKEY considerably restricts the portability of programs.

Example:

```
CALL FORM  ... ( ACCEPT (ENTER, F10='HELPE', F3='END',
                 CMDKEY, LEFTKEY, RIGHTKEY ) );

CASE  $KEY OF
   ENTER: ...
   F10: ...
   ...
   LEFTKEY: ...
   RIGHTKEY: ...
END;
```

The query as to the key last used can be formulated either by means of the key literals or their values. The following table shows the connection between the key literals and their values:

Key Literal	Assigned Value
F1	'F1'

```
|    F2                                  |           'F2'                         |
|    F3                                  |           'F3'                         |
|    F4                                  |           'F4'                         |
|    F5                                  |           'F5'                         |
|    F6                                  |           'F6'                         |
|    F7                                  |           'F7'                         |
|    F8                                  |           'F8'                         |
|    F9                                  |           'F9'                         |
|    F10                                 |           'F10'                        |
|    F11                                 |           'F11'                        |
|    F12                                 |           'F12'                        |
|    HELP                                |           'F10'                        |
|    UP                                  |           'F11'                        |
|    DOWN                                |           'F12'                        |
|    ENTER                               |           'ENTER'                      |
|    LEFTKEY                             |           'LEFTKEY'                    |
|    RIGHTKEY                            |           'RIGHTKEY'                   |
|    HELPKEY                             |           'HELPKEY'                    |
|    UPKEY                               |           'UPKEY'                      |
|    DOWNKEY                             |           'DOWNKEY'                    |
|    CMDKEY                              |           'CMDKEY'                     |
|    ENDKEY                              |           'ENDKEY'                     |
|    CRSLEFT                             |           'CRSLEFT'                    |
|    CRSRIGHT                            |           'CRSRIGHT'                   |
|    CRSUP                               |           'CRSUP'                      |
|    CRSDOWN                             |           'CRSDOWN'                    |
|                                        |           'PREVFLD'                    |
|                                        |           'NEXTFLD'                    |
|________________________________________|________________________________________|
```

Key Literal	Assigned Value

F1	’NEXTFLD’
F2	’F2’
F3	’F3’
F4	’F4’
F5	’F5’
F6	’F6’
F7	’F7’
F8	’F8’
F9	’F9’
F10	’F10’
F11	’F11’
F12	’F12’
HELP	’F10’
UP	’F11’
DOWN	’F12’
ENTER	’ENTER’
LEFTKEY	’LEFTKEY’
RIGHTKEY	’RIGHTKEY’
HELPKEY	’HELPKEY’
UPKEY	’UPKEY’
DOWNKEY	’DOWNKEY’
CMDKEY	’CMDKEY’
ENDKEY	’ENDKEY’
CRSLEFT	’CRSLEFT’
CRSRIGHT	’CRSRIGHT’
CRSUP	’CRSUP’
CRSDOWN	’CRSDOWN’
	’PRECFLD’

When using string constants, care must be taken that they are written in upper cases. The key literals, by contrast, are recognized as keywords so that they are not case significant.

Syntax:

```
<accept option>  ::=  ACCEPT (<key spec>,...)

<key spec>  ::=  ENTER
              |  <basic key> [ = <key label>]
              |  <additional hardkey>

<basic key>  ::=  F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8
               |  F9 | F10 | F11 | F12
               |  HELP   | UP     | DOWN

 <additional hardkey>  ::=  HELPKEY | CMDKEY | ENDKEY | UPKEY
                         |  DOWNKEY | RIGHTKEY | LEFTKEY
```

Overriding Display Attributes (ATTR)

When calling a form, individual display attributes can be overridden as required.

Example:

```
CALL FORM insert ( ATTR (input, INV) );

CALL FORM insert ( ATTR (cno, ATTR16)),
                   ATTR (name, HIGH ),
                   ATTR (today, ATTR5)) );
```

These calls cause the field ’input’ to be represented in the form ’insert’ according to the attribute setting INV or ATTR13 (from the Set parameters) up to the time when the form is called without attribute options or with other attribute options.

With the second call, the field ’cno’ is represented according to the attribute setting INV, the field ’name’ according to HIGH and the field ’today’ according to ATTR5.

Syntax:

```
<attr option>  ::=  ATTR ( <form var>, <attr name> )

<attr name>  ::=   see "Display Attributes
                   HIGH,LOW,INV,BLK,UNDERL, ATTR1..ATTR16)" (8.2.7.1)
```

The Window Options SCREENPOS, SCREENSIZE, and CLEAR

An SQL-PL program can open several windows on the screen and handle various forms within these windows. It is not a matter of user-controlled ’multi-windowing’ but of being able to display forms dynamically from within the program.

The default window in which a form is displayed for the calls described so far is the entire (physical) screen.

The option SCREENPOS (line,column), can be used to define the position on the screen at which the top left-hand corner of the form is to start.

The option SCREENSIZE (length,width) specifies how many lines long and how many columns wide the segment on the screen should be. It can happen that input or output fields can only be partially displayed within the chosen segment or not at all.

```
                                                    |                    |
                                                    |    SQL-PL Routine  |
                                                    |____________________|
   CALL FORM mastercard;

   IF   ($KEY = F1)
         AND ($CURSOR = 1)  /* CNO field
   THEN
         CALL FORM cno_help OPTIONS
              (SCREENPOS(10,15), FRAME );
```

By specifying the FRAME option the displayed form is output in a frame. If the terminal has the facility, the frame in the FRAME option is represented semi-graphically.

To support the application programmer, it suffices to specify only the SCREENPOS option to display the form in the size of its layout.

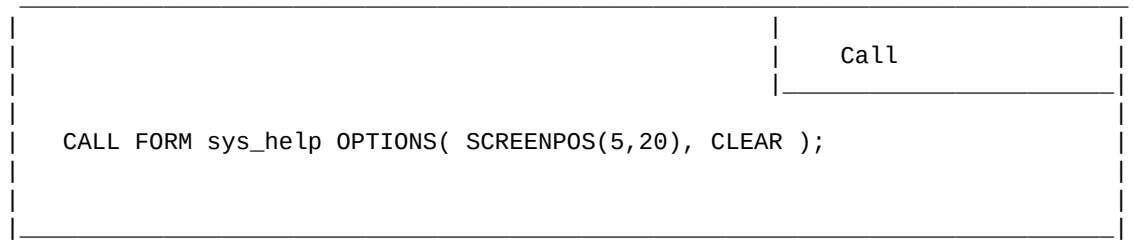
The call for ’cno_help’ leads to:

```
   MILLER      C U S T O M E R    M A S T E R C A R D          11/15/02

         CUST-NO        : 1234

            TITLE       : .....
            FIRSTNAME   : .............
            NAME        : .............

          ANSC     |  cust.no.          |
                   |____________________|
                   |                    |
                   | max 5 digits       |
                   |                    |
                   | starts with C      |
          ACC.     | after that numeric |        ...... $
                   |                    |
___________________|                    |_____________________________________
                   |                    |
                   | Examples:          |
                   | C1, C1234          |
                   |____________________|
```

The displayed form can contain input and output fields. The fields of the underlying form(s) remain(s) on the screen, input is only accepted in the form that was called last.

A window is closed implicitly if it is completely covered by the window of a new form call. In the example above, the displayed form disappears when the underlying form (mastercard) is called.

If a form is to be displayed and the screen is to be cleared before outputting the specified window, the CLEAR option is required:



This can be useful, e.g. at the beginning of a program, when a small form is to be output in the middle of the screen.

Syntax:

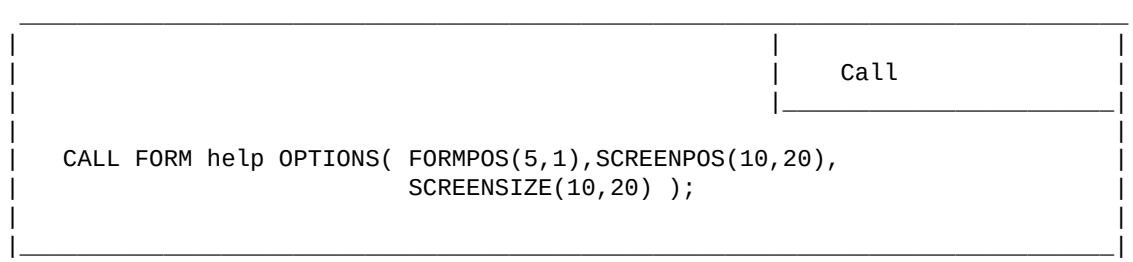
```
<window option>  ::=  SCREENPOS (<expr>,<expr>)
                    |  SCREENSIZE (<expr>,<expr>)
                    |  CLEAR
```

Form Segments (FORMPOS)

SCREENPOS and SCREENSIZE can also be used to define a window that is smaller than the form to be displayed within it.

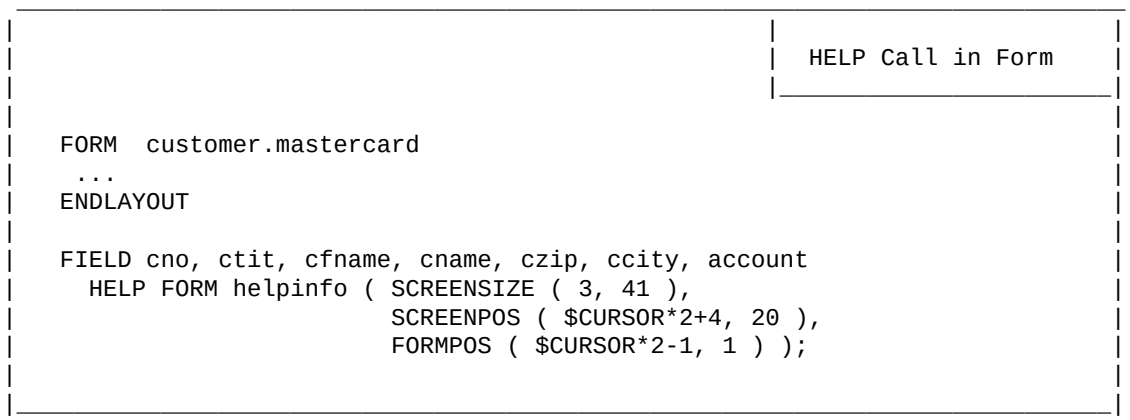
If nothing else has been specified, FORM displays in this case the top left-hand part of the form in the window and truncates it to the right and below.

If this is not desired, that segment of the form that is to be displayed in the window can explicitly be specified with the FORMPOS option :



With all these options, SCREENPOS, SCREENSIZE, and FORMPOS, not only constants but also expressions can be specified.

Example:



In this example, a three-line segment from the HELP form ’helpinfo’ is displayed as help information that always differs according to the position of the cursor.

Syntax:

```
<formpos option> ::= FORMPOS (<expr>,<expr>)
```

Automatic Framing by FRAME

The described options SCREENSIZE and SCREENPOS cause a form to be output in a screen segment. The FRAME option draws a frame line around this screen window. In this way, the displayed form can be emphasized as a window without having to alter the form definition.

In this context the following must be noted:

- The FRAME option enlarges the window four characters in the width and two lines in the length.
- In the case of a form that fills the entire screen, the FRAME option causes the window content to be made smaller by up to four characters on the right margin and up to two lines on the lower margin.
- The displayed form content is shifted one row down and two characters to the right. This means, the SCREENPOS option refers to the position of the top left corner of the frame.
- In a form that fills the entire screen and that has the implicit message line (no MESSAGE field in the form layout) and the implicit key display (ACCEPT option), if applicable, the lower frame line appears above the message and key lines.
- If the screen allows, the frame is displayed using semi-graphic characters.

In addition, a title can be displayed on the upper frame line by means of the FRAME option. The desired title can be specified as string expression or variable.

Example:

```
CALL FORM cno_help OPTIONS
     (SCREENPOS(10,15), FRAME ( ’Help Information’ ));
```

The title appears with the attribute ATTR13 in the middle of the upper frame line.

Syntax:

```
<frame option> ::= FRAME [ (<expr>) ]
```

Superimposing Forms (BACKGROUND)

The window options already described permit forms to be superimposed in such a way that one form after the other appears on the screen. Each form call results in an output made to the screen.

In contrast to this, the BACKGROUND option permits several forms to be combined into one screen output. A form called with the BACKGROUND option does not appear immediately on the screen but is stored as background.

Any number of forms can be superimposed one after the other with the BACKGROUND option. The terminal screen output only takes place when a form is called either without the BACKGROUND option or by a READ or WRITE statement. Of course, the WRITE statement must not contain any OPEN option which corresponds to the BACKGROUND option in the case of forms.

This mechanism can also be used in connection with REPORT output by, e.g., first calling a form with the BACKGROUND option and then a REPORT output which for practical purposes should only cover a part of the screen (see the "Query" manual, Section, "The WINDOW Command").

Syntax:

```
 <background option> ::= BACKGROUND
```

Restoring the Form Background (RESTORE)

A form that occupies only a segment of the screen leaves a blank screen segment when it disappears. The application programmer must ensure that this screen segment is filled again with the previous background.

This is easy if the smaller form is contained completely in the preceding form. A more difficult problem arises for the application programmer when the small form would destroy the background consisting of several forms called one after the other.

The RESTORE option implicitly saves the background so that the form, when disappearing, can restore its background as it was before it appeared.

For HELP forms specified in the FIELD statement, FORM implicitly uses the RESTORE option.

Syntax:

```
 <restore option> ::= RESTORE
```

Form Output via Printer (PRINT)

By means of the PRINT option a form can be output to the printer instead of to the screen:

```
CALL FORM mastercard OPTIONS( PRINT );
```

```
 CALL FORM mastercard OPTIONS( PRINT(CLOSE) );
```

The form is printed out in its full width and length if this is permitted by the printout format set in the Set parameters.

As for screen output, the options SCREENSIZE and FORMPOS are also taken into account for printing, so that even segments of a form can be printed out. The option SCREENPOS, however, does not have any effect on printouts.

For a series of several form calls with PRINT option, the forms are printed out one after the other without page feed. The page feed must be controlled explicitly by the NEWPAGE and CPAGE options.

The option PRINT(CLOSE) starts the printer to print out.

For preparing such a printout, the following control statements are available in addition:

LINEFEED	for generating blank lines before the printout.
LINESPACE	for setting the line spacing.
NEWPAGE	for outputting on a new page.
CPAGE	for performing a page feed depending on the number of empty lines.
PRINTFORMAT	name of the print format for the form print-out (see Section, "User-specific Set Parameters").

Example:

```
CALL FORM form OPTIONS ( PRINT (LINEFEED 2,
                        NEWPAGE, LINESPACE 3))
```

The printout starts on a new page with two blank lines; the lines of the form are printed on every third line.

```
1. CALL FORM form
        OPTIONS ( PRINT ( PRINTFORMAT 'FORMAT1', CLOSE) )

2. format_name := 'ADDRESSFORMAT';
   CALL FORM address
        OPTIONS ( PRINT ( PRINTFORMAT format_name ) );
```

The form in the first example is printed out with the print format called 'FORMAT1', in the second with the print format called 'ADDRESSFORMAT'.

A print format serves to combine a series of print parameters for repeated use. Print formats are defined with the user-specific Set parameters.

Syntax:

```
<print option>  ::=  PRINT [(<print option>,...)]

<print option>  ::=  CLOSE
                   |  CPAGE <natural>
                   |  LINEFEED <natural>
                   |  LINESPACE <natural>
                   |  NEWPAGE
                   |  PRINTFORMAT <expr>
```

Overriding the Active Input Fields (INPUT/NOINPUT)

Form processing in FORM is in general screen-oriented. This means that all visible input fields of the form can be described and processed.

The INPUT option can be used to restrict the processing to individual fields or groups of fields.

The input fields to be activated can either be identified by their sequential field number or by the variable name.

Examples:

1. CALL FORM x OPTIONS (INPUT (1, 2) ...);
2. CALL FORM x OPTIONS (INPUT (cno, cname) ...);
3. CALL FORM x OPTIONS (INPUT (addr(1..4), cno) ...);
4. CALL FORM x OPTIONS (INPUT (cno, 2, addr(1..4)) ...);

Example1:

Only the two first input fields are treated as such. The other input fields remain write-protected.

Example2:

The input fields that should be active are identified by their variable names.

When using the variable names in the INPUT option, the procedure is independent of any re-sorting of the fields in the form layout.

Example3:

Apart from simple variable names, vector components or even vector slices can be specified.

Example4:

The various field arguments of the INPUT option can be mixed, as this example shows.

For situations in which only a few input fields are to be made passive when they are called, there is the NOINPUT option.

The NOINPUT option has precisely the opposite effect of the INPUT option. For the NOINPUT option the same arguments are permitted as for the INPUT option.

```
CALL FORM x OPTIONS ( NOINPUT ( cno, addr(1), 5, ...));
```

If the list behind INPUT is empty, that is INPUT(), then the form has no active input fields in this call.

NOINPUT(), on the other hand, means that all input fields are to be active.

CHECK conditions are only in effect for the input fields active at runtime.

Thus the fields can be identified as for the MARK option with the variable names or the sequential field numbers.

The variable $CURSOR, however, always returns the sequential field number of the input field. For the sequential field numbers, only input fields are counted from the top left to the bottom right.

Syntax:

```
<input option>  ::=  NOINPUT (<input field>,...)
                   |  INPUT (<input field>,...)

<input field>   ::=  <natural>
                   |  <variable>
                   |  <vector slice>
```

Activating the Action Bar (ACTION)

When a form with action bar is called, it is output in such a way that the form is active and the action bar is passive. By specifying the call option ACTION, the specified field of the action bar is active immediately when calling the form.

```
 CALL FORM x OPTIONS ( ACTION ( 5 ));
```

Syntax:

```
<action option> ::= ACTION (<expr>)
```

HELP Forms as Pick Lists

Example: Pick list with pick value assignment within the HELP form

```
 ____________________________________________________________________________
|                                                                            |
|  HELPFORM customer.list_of_names                                           |
|   LAYOUT  GRAPHIC=* LOW =+                                                 |
|   _selection (1..5)                                                        |
|   _                                                                        |
|   _                                                                        |
|   _                                                                        |
|   _                                                                        |
|                                                                            |
|   ENDLAYOUT                                                                |
|   SPECIALATTR INPUT INV;                                                   |
|   ACCEPT ( ENTER,  UPKEY, DOWNKEY );                                       |
|   BEFORE GROUP a                                                           |
|       BEGIN                                                                |
|       SQL ( SELECT name FROM CUSTOMER );                                   |
|       SQL ( FETCH INTO :selection(1..255) );                               |
|       END;                                                                 |
|   GROUP a                                                                  |
|      FIELD selection(1..5) SIZE 15 OFFSET @x NOINPUT                       |
|         AFTER  FIELD                                                       |
|            BEGIN                                                           |
|            SCROLLFIELD ( @x, $COUNT );                                     |
|            IF $KEY = ENTER                                                 |
```

```
|            THEN                                                              |
|                 PICK ( selection($CURSOR+@x) );                              |
|            END;                                                              |
|______________________________________________________________________________|
```

This form simultaneously represents five customer names. The displayed section is moved within the retrieved list of names by means of the cursor and scroll keys. When the ENTER key is pressed, that name is picked out at which the cursor is placed.

Example: Calling the pick list

```
 ______________________________________________________________________________
|                                                                              |
|    FORM customer.mastercard                                                  |
|    LAYOUT  PROMPT =. low=+                                                   |
|    Name:            _name             +                                      |
|    Firstname:       _firstname        +                                      |
|    ...                                                                       |
|                                                                              |
|    BUTTON                                                                    |
|    ENDLAYOUT                                                                 |
|    BUTTON  ( 'Help':RELEASEKEY HELP, 'End' );                                |
|    KEYSWAP ( F1<->HELP );                                                    |
|    ...                                                                       |
|    FIELD name                                                                |
|    HELP FORM list_of_names ( AUTOPUT , FRAME );                              |
|                                                                              |
|                                                                              |
|______________________________________________________________________________|
```

If the key F1 is pressed while the cursor is placed on the field 'name' the HELP form 'list_of_names' appears. The AUTOPUT option has the effect that the value picked out of the HELP form is automatically passed to the field 'name'.

The vector slice can be moved directly to the end (top) of the list, when the key defined as BOTTOMKEY (TOPKEY) is pressed.

```
 ______________________________________________________________________________
|                                                                              |
| FIELD selection(1..5) SIZE 15 OFFSET @x NOINPUT                              |
|     AFTER  FIELD                                                             |
|       BEGIN                                                                  |
|        SCROLLFIELD ( @x, @max_cnt, 5 )                                       |
|                    ( TOPKEY F7, BOTTOMKEY F8 );                              |
|        IF $KEY = ENTER                                                       |
|        THEN                                                                  |
|              PICK ( selection($CURSOR+@x) );                                 |
|        END;                                                                  |
|______________________________________________________________________________|
```

Action Bar with Pulldown Menus and BUTTON Bar

Apart from controlling forms by means of function keys, there is the possibility of defining action bars and pulldown menus. This increases the number of functions that can be chosen in an application.

Then the control flow within the SQL-PL program no longer depends on the limited number of activated keys but on the chosen menu items.

Pulldown menus consist of an action bar (ACTIONBAR) and the pulldown menus defined for the menu items.

There are two ways of defining pulldown menus:

- within a form
- as separate menu module.

The separate menu module has the advantage of being capable of being used - by means of the INCLUDE statement - for all forms of the application.

This section covers the following topics:

- Defining a Separate MENU Module
- Defining the Action Bar within a Form
- Defining an Action Bar and a Pulldown Menu
- Examples of Action Bars with Pulldown Menus
- Using an Action Bar with Pulldown Menus
- The BUTTON Bar

Defining a Separate MENU Module

If the same action bar and the associated pulldown menus are to be used in various forms of an SQL-PL program, a separate MENU module can be defined that is linked to a form with the statement INCLUDE MENU.

In the MENU module, the action bar and the pulldown menus are defined one after the other.

Example: Definition of a MENU module

```
MENU general.pd_menu

 ACTIONBAR ('LIST',
            'PROCESS' : PULLDOWN process,
            'DELETE' );

 PULLDOWN process ( 'START',
 'ENTER' : PULLDOWN entry );

 PULLDOWN entry ( 'NEW',
                  'OLD' );
```

In a form containing the INCLUDE MENU statement the keyword ACTIONBAR must be specified within the layout part, as for the definition of the action bar within a form.

Example: Statement INCLUDE MENU

```
FORM general.form1
LAYOUT
ACTIONBAR
...

ENDLAYOUT

INCLUDE MENU pd_menu;
 ...
```

Note:
A menu module cannot be tested with the TEST function. The menu is only displayed when the form that contains the INCLUDE statement is tested.

If the MENU module is part of another program, its program name must be specified for the INCLUDE statement. In this case, the called MENU module cannot access the global variables of the calling module. Even if the MENU module belongs to the same program, the usage of the program name has the effect that the global variables of the called MENU module are different from those in the calling module (see global variables for the SWITCHCALL call).

Syntax:

```
<menu>  ::=  MENU <prog name>.<mod name>
             [PARMS (<formal parameter>,...)]
             <actionbar>
             [ <pulldown> ]

<include menu stmt>  ::=  INCLUDE MENU  [<prog name>.]<mod name>
                        |  [PARMS (<formal parameter>,...)]
```

Defining the Action Bar within a Form

For smaller applications or for testing a menu, it is useful to define the menu directly in the form.

For this purpose, the statements for defining the menu (ACTIONBAR, PULLDOWN statement) are formulated in the processing part of the form.

Syntax:

```
<form>  ::=  FORM  <prog name>.<mod name>
             [OPTIONS (<form option>,...)]
             [PARMS (<formal parameter>,...)]
             [<var section>]
             <form layout>
             [ ...
              <actionbar>
             [ <pulldown> ]
               ... ]
```

Defining an Action Bar and a Pulldown Menu

A menu consists of an action bar and the pulldown menus associated with it. A menu can also simply consist of the action bar.

An action bar is a horizontal menu, whereas a pulldown menu is a vertical menu. Definitions concerning a menu item can be specified in the action bar as well as in a pulldown menu.

A pulldown menu is always displayed in a frame, whereas the action bar may be shown with or without a frame. The position of a pulldown menu is automatically determined by SQL-PL. The action bar, by contrast, can be positioned by the application programmer on any line of the form layout.

Syntax:

```
<actionbar stmt>  ::=  ACTIONBAR [WITH FRAME] ( <menupoint def>,... )
                    |  ACTIONBAR [WITH FRAME] ( <menupoint group>;... )

<pulldown stmt>   ::=  PULLDOWN <name> ( <menupoint def>,... )
                    |  PULLDOWN <name> ( <menupoint group>;... )

<menupoint group>  ::=  <menupoint def>,...

<menupoint def>   ::=  <function label> [ : <action clause> ]

<function label>  ::=  <variable>
                    |  <string>
                    |  <langdep literal>

<action clause>   ::=  [<comment>] [<activate cond>] [<action>]

<comment>  ::=  COMMENT <expr>

<activate cond> ::=  WHEN <boolean expr>

<action>  ::=  <pulldown call>
             |  <releasekey spec>

<pulldown call>   ::=  PULLDOWN <name>

<releasekey spec>  ::=  RELEASEKEY <key literal>

<key literal>  ::=  <basic key>
                  |  <additional hardkey>
```

This section covers the following topics:

- Defining the Action Bar (ACTIONBAR)
- Defining a Pulldown Menu (PULLDOWN)
- Defining Guidance Texts (COMMENT Clause)
- Dynamical Deactivation of a Label (WHEN Clause)
- Optical Grouping of Menu Items

Defining the Action Bar (ACTIONBAR)

An action bar consists of up to eleven fields that can be output in a line of the form one after the other. When defining the action bar, the labels of the fields are specified behind the keyword ACTIONBAR. Apart from that, the position of the action bar in the form is defined by specifying the keyword ACTIONBAR in the layout. The layout line in which the action bar is to be output must not contain any other fields.

The dollar variable $ACTION, which returns the activated field of the action bar, corresponds to the fields of the action bar.

Example:

```
LAYOUT
ACTIONBAR

  ...

ENDLAYOUT

ACTIONBAR ( 'LIST','PROCESS','DELETE' );
```

Valid labels are string constants, language-dependent literals, variables, and key literals.

The labels can be up to 16 characters long. If there is no subsequent pulldown menu, the label may have 18 characters.

It must be noted that the dollar variables $ACTION, $FUNCTION1, ... $FUNCTION4 and $FUNCTION return the value truncated to 16 (or 18) characters if longer labels are used.

For each label FORM automatically finds out a choice letter which, combined with the CTRL key, serves to select a function. By placing a '&' sign before a letter, the user can determine which letter within a label is to be taken as choice letter. Note that thereby the label can only be up to 17 characters long.

If an action bar is defined in the form, the number of header lines is implicitly set to the line in which the action bar is output, i.e. the header lines comprise the area of the form from the first line to the action bar (see Section, "Header Lines and Bottom Lines (HEADERLINES, BOTTOMLINES)", statement "HEADERLINES").

Example:

```
ACTIONBAR WITH FRAME ( !LIST(s),
                       !PROC(s),
                       !DATA(s) );
```

The option WITH FRAME causes the action bar to be output in a frame. Care must be taken that the lines before and after the keyword ACTIONBAR in the layout is left empty, since otherwise they are overwritten by the frame lines.

Syntax:

```
<actionbar stmt>  ::=  ACTIONBAR WITH FRAME ( <menupoint def>,... )
                     |  ACTIONBAR WITH FRAME ( <menupoint group>;... )
```

Defining a Pulldown Menu (PULLDOWN)

For each field of the action bar, a pulldown menu can be defined. To do this, a reference to the pulldown menu is given behind the label of the field in the ACTIONBAR definition and the pulldown menu is defined analogously to the action bar. In the same way, a further pulldown menu can be defined for each field of a pulldown menu.

Example:

```
_____________________________________________________________________________
|                                                                           |
| LAYOUT                                                                    |
| ACTIONBAR                                                                 |
|                                                                           |
|  ...                                                                      |
|                                                                           |
| ENDLAYOUT                                                                 |
|                                                                           |
| ACTIONBAR ('LIST',                                                        |
|            'PROCESSING' : PULLDOWN proc,                                  |
|            'DELETE' );                                                    |
|                                                                           |
| PULLDOWN proc ( 'START','ENTER' );                                        |
|___________________________________________________________________________|
```

In a pulldown menu, up to 20 labels can be specified which are output one beneath the other under the calling field of the action bar.

The pulldown menus are positioned downward in the first level and to the right downward and overlapping in further levels. Up to five levels are possible.

If a field of a pulldown menu is activated, the corresponding label is returned in $FUNCTION. Thereby the labels of the pulldown menu beneath a field of the action bar must be unique.

$FUNCTION1, ... $FUNCTION4 return the function last chosen of the pulldown menu level designated by its number. In this way it is possible to distinguish the same pulldown submenu several times within a pulldown menu hierarchy.

The same conditions apply to the labels of the pulldown menu and the action bar.

Defining Guidance Texts (COMMENT Clause)

For each label of the action bar or a pulldown menu, a brief comment can be defined behind the keyword COMMENT. The brief comment appears in the message line as soon as the associated label is activated.

Example:

```
ACTIONBAR ('LIST':  COMMENT 'generate a list of all objects',
           'PROCESS':  COMMENT 'further processing functions',
                       PULLDOWN proc,
           'DELETE':   COMMENT 'delete object');


PULLDOWN proc ( START' :     COMMENT !progstart,
                'ENTER' : COMMENT !enter_object );
```

Dynamical Deactivation of a Label (WHEN Clause)

A pulldown menu represents the functions that can be chosen from the form. Depending on various conditions, it may be desirable to deactivate certain parts of the functions, but to display them nevertheless.

This can be done with the WHEN condition that can be specified behind every label of a pulldown menu. Depending on this condition, the corresponding label is displayed with the display attribute 'menu item passive' (ATTR12), and the associated function cannot be chosen.

Example:

```
PULLDOWN enter
   ( 'back'  :            WHEN  level > 1,
     'trigger funct.'  :  WHEN  modtype ='TRIGGER'
                             PULLDOWN trigger_funcs );
```

If, in the example, the variable 'level' has a value less than or equal to 1, the label 'back' is deactivated. If the variable 'modtype' does not have the value 'TRIGGER', the label 'trigger functions' is deactivated and the following pulldown menu cannot be called.

Optical Grouping of Menu Items

To be able to group the menu items of a pulldown menu according to logical criteria, a semi-colon can be specified in the list of menu items instead of a comma. The semi-colon causes a separating line to be output between the menu items separated in this way.

Example:

```
PULLDOWN proc ( 'Insert',
                'Delete';
                'Import',
                'Export' );
```

In this example two groups of menu items are displayed.

Examples of Action Bars with Pulldown Menus

Examples

The fields of the action bar and of the pulldown menus behind which no further pulldown menus are defined correspond to actions that are to be performed. If one of the fields is activated, the dollar variables $ACTION, $FUNCTION1, ... $FUNCTION4 and $FUNCTION are set to the corresponding labels. These can be used to define the desired action in the CONTROL block of the form.

Example:

```
LAYOUT
ACTIONBAR

 ...

ENDLAYOUT

ACTIONBAR ( 'process'   : PULLDOWN proc,
            ...     );

PULLDOWN proc ( 'module'   : PULLDOWN fctn,
                ...
                'trigger' : PULLDOWN fctn );

PULLDOWN fctn ( 'display',
                 ...
                'print' );

CONTROL CASE $ACTION OF
        'process' :
               CASE $FUNCTION1 OF
               'module' :
                     CASE $FUNCTION OF
                          'display'  : CALL PROC mod_no;
                            ...
                          'print'   : CALL PROC mod_print;
                      END;
                ...
                'trigger' :  ...
                END;
          ...
          END;
```

In the following example labels are specified that are not string constants. These allow the above procedure to be used in the same way.

Example:

```
ACTIONBAR ( !ADM(s),
            !PROCESS(s): PULLDOWN process );

PULLDOWN process ( 'UPDATE','INSERT' );

CONTROL CASE $ACTION OF
```

```
|         !ADM(s)  :  ...                                                    |
|         !PROCESS(s)  :  CASE $FUNCTION OF                                  |
|                              'PROCESS'  : CALL FORM start;                 |
|                              'INSERT'   : CALL FORM insert;                |
|                         END;                                               |
|         END;                                                               |
|____________________________________________________________________________|
```

Example:

```
_____________________________________________________________________________
|                                                                           |
| ACTIONBAR (!UP(s)    : RELEASEKEY F4,                                     |
|            !DOWN(s)  : RELEASEKEY F5 );                                   |
|                                                                           |
| CONTROL CASE $KEY OF                                                      |
|         F4  : ...                                                         |
|         F5  : ...                                                         |
|         END;                                                              |
|___________________________________________________________________________|
```

Example:

```
_____________________________________________________________________________
|                                                                           |
|  FORM menutest.ff1 OPTION (FIELD )                                        |
|  LAYOUT                                                                   |
|  ACTIONBAR                                                                |
|                                                                           |
|  Custno        : _cno                                                     |
|  Customername  : _cname                                                   |
|  City          : _city                                                    |
|  ...                                                                      |
|  Text          : _text                                                    |
|  ENDLAYOUT                                                                |
|                                                                           |
|  FIELD cno SIZE 20                                                        |
|     BEFORE FIELD MESSAGE:='Please enter customer number'                  |
|     AFTER FIELD                                                           |
|     IF $FUNCTION='SEARCH'                                                 |
|     THEN                                                                  |
|          BEGIN                                                            |
|          SQL ( SELECT DIRECT name, city INTO :cname,:city..);             |
|          IF $RC=0                                                         |
|          THEN NEXTFIELD text                                              |
|          ELSE BEGIN                                                       |
|               MESSAGE := 'This customer name is not known';               |
|               NEXTFIELD cno;                                              |
|               END;                                                        |
|          END                                                              |
|      ELSE  NEXTFIELD text;                                                |
|  FIELD cname ...                                                          |
|  ACTIONBAR ( 'LIST', 'PROCESS' : PULLDOWN process, 'DELETE' );            |
|  PULLDOWN proc ( 'START','ENTER','SEARCH' );                              |
|___________________________________________________________________________|
```

In this example, the fields 'customer name' and 'customer address' are retrieved from a table if the user has chosen the function 'SEARCH' in the pulldown menu 'PROCESS' before, during or after entering the customer number. In this case the two selected fields are skipped; otherwise, the user has to fill them in.

Syntax: See Section, "Defining an Action Bar and a Pulldown Menu".

Using an Action Bar with Pulldown Menus

When calling the form, the action bar is not activated and the form can be processed in the usual way.

The action bar can be activated in the following ways:

- function key F12

  The function key F12 activates the action bar and positions the cursor to the first field.

- CTRL / <char>

  Simultaneously pressing the CTRL key and the letter highlighted as choice letter directly selects the corresponding action. If there is a pulldown menu for the action, it is pulled down; otherwise, $ACTION returns the chosen action.

If the action bar is activated, the corresponding action can be started by the key for the letter enhanced in the label, e.g. the pulldown menu of the field with the label ’PROCESS’ is pulled down with the key p. The appropriate letter is automatically determined by the system and represented with the attribute ’select char’ (ATTR8) or ’select char active’ (ATTR9) (see Section, "User-specific Set Parameters").

In an active pulldown menu, the functions can be chosen and started analogously.

If the action bar is activated, the cursor can be positioned on the fields to choose a field of the action bar with the cursor keys. If a pulldown menu is defined behind a field of the action bar, the menu can be pulled down by positioning on the field and then pressing the ENTER key.

If the program is positioned in a pulldown menu on the first level, the pulldown menus to the right or left can be chosen directly by means of the NEXTFIELD and PREVFIELD keys or graphics/sqlpl2.gif and graphics/sqlpl42.gif .

Fields of the action bar or the pulldown menus behind which a further pulldown menu is defined are identified by ’..’.

The action bar is displayed with the attribute ’menu items’ (ATTR10), the active field with the attribute ’menu item active’ (ATTR11) and the passive fields with the attribute ’menu item passive’ (ATTR12).

If an action of a field of the action bar or a pulldown menu is started, e.g. by positioning the cursor to this field and pressing the ENTER key, control is returned to the associated form and the dollar variables $ACTION, $FUNCTION or also $KEY are assigned the labels of the chosen fields.

If one wants to return to the form without starting an action, this is done with the key F12 . Control is now returned to the associated form and the dollar variables $ACTION, $FUNCTION and $KEY have the NULL value.

Pressing the key BACKSPACE closes the pulldown menus step by step. When doing so, the NULL value is assigned to the associated dollar variables.

The BUTTON Bar

Alternatively to the action bar a series of release fields (BUTTON bar) can be defined at the bottom form margin.

The BUTTON bar is defined in the same way as an action bar. A list of labels is specified behind the keyword BUTTON, and the position of the BUTTON bar is defined in the layout part of the form by means of the keyword BUTTON.

The differences to the action bar are:

1. It is not possible to define pulldown menus in the BUTTON bar.
2. The position of the BUTTON bar and the spaces between the buttons are determined by the system.
3. When BUTTON WITH FRAME is defined, each of the labels will be provided with a frame of its own.
4. The BUTTON bar is assigned to the bottom lines (BOTTOMLINES) of the form, i.e. the bottom lines of the form begin at least with the line in which the BUTTON bar is output.

Example:

```
LAYOUT
  ...
        BUTTON
  ENDLAYOUT
  ...
  BUTTON ( 'Help', 'Start', 'Exit' );
```

Syntax:

```
<button stmt>  ::=   BUTTON [WITH FRAME] (<buttonpoint def>,...)

<buttonpoint def>  ::=  <function label> [ : <button action> ]

<button action>   ::=  [<comment>] [<activate cond>] [<releasekey spec>]
```

Module Options

This section covers the following topics:

- The LIB Option
- The WRAP Option

The LIB Option

The LIB option specifies the name of the function library to which the SQL-PL functions belong that are used in the form.

Syntax:

```
<form lib option> ::= LIB [<username>.]<libname>
```

The WRAP Option

The WRAP option causes multi-line fields of a form to be represented as continuation fields on terminals that allow such representation. A continuation field can be recognized by the fact that the following characters within the continuation field are pushed over the limits of the partial fields when characters are inserted in a partial field of the continuation field.

Content can only be shifted if there are blanks at the end of the field content. For fields that are preset to PROMPT characters, the content is not shifted.

When using the WRAP option, care must be taken that no further input fields are defined in the lines occupied by the continuation field, since these automatically disable the WRAP function.

Syntax:

```
 <form wrap option> ::= WRAP
```

Language-dependent Programs

For supporting the development of multi-lingual programs, there is the concept of language-dependent literals. A language-dependent literal is represented by its name, which starts with an ’!’ (exclamation mark). When storing the module, the names of literals are replaced by their values in the language currently set (’DEU’, ’ENG’, ’FRA’). These literals are not case significant.

The literal names and their values can be easily inserted into the underlying system table by means of DOMAIN. Like the SQL-PL programs, the literal names are also user-specific. Only the language-dependent literals of an author can be part of his program.

The name of the literal is replaced by one of four literals of differing lengths. The desired size of the literal is chosen by specifying S, M, L and XL behind the literal name. S, M, L and XL have the meanings: short (length 8), medium (length 12), large (length 18) and extra large (length 80).

Example:

```
|                                                                              |
|  !tab_name(s)   --> ’table’                                                  |
|  !tab_name(m)   --> ’table name’                                             |
|  !tab_name(l)   --> ’name of a table’                                        |
|  !tab_name(xl)  --> ’This is the name of a table’                            |
|                                                                              |
|  !col_name(s)   --> ’column’                                                 |
|  !col_name(m)   --> ’column name’                                            |
|  !col_name(l)   --> ’name of a column’                                       |
|  !col_name(xl)  --> ’This is the name of a column’                           |
|______________________________________________________________________________|
```

If a literal name cannot be found in the system table at the translation point in time, the literal name itself is displayed as value.

Example of the case that no entries exist for the literal !TAB_NAME:

```
|                                                                              |
| !tab_name(s)             --> ’TAB_NAME’                                      |
| !column_name(m)          --> ’COLUMN_NAME’                                   |
| !customer_directories(l) --> ’CUSTOMER_DIRECTORY’                            |
|______________________________________________________________________________|
```

The way of writing the literal name is converted to upper case and is accepted with a maximum length of 18 characters.

In this way, SQL-PL programs can be written without previously defining the literals to be used. The workbench command ’LIT’ can then be used to find out the literals still undefined and to define them ad hoc.

This chapter covers the following topics:

- Language-dependent Literals in Procedures

- Language-dependent Literals in Forms

Language-dependent Literals in Procedures

A language-dependent literal can be used instead of a string in an SQL-PL procedure or function, as described in the previous section.

Example:

```
message := !No_Entry(XL);
EDIT ( txt, F3=!BACK(s) );
```

Syntax:

```
<langdep literal>  ::= !<name> (<literal size>)

<literal size>  ::= S | M | L | XL
```

Language-dependent Literals in Forms

In contrast to procedures, there are two ways of using literals in forms.

In the FORM layout, the literals are used as a substitute for text fields. In contrast to the notation described up to now, literals are written within the form layout without specifying the length. Since for a form field, the length is always known, the FORM compiler accepts the largest of the four values that fits into the form field as literal value.

Example:

```
                                          Layout Definition
LAYOUT prompt=. low=+

 !tab_name  :   _tabname          +
 !col_name    : _colname          +

 ENDLAYOUT
```

If one proceeds from the above example with the literals defined there, the form would look like the following:

```
                                          Executed Form
Table : ..................
Column name : ..................
```

Outside the form layout, literals can also be used in all the other statements. Here, however, they are used with a length specification, as in SQL-PL procedures.

```
FIELD message INIT !entry_msg(xl);

ACCEPT ( F10=!HELP(s), F3=!END(s) );
```

Syntax:

```
 <langdep literal>  ::=  !<name>      <-- only in the form layout
                      |  !<name> (<literal size>)

  <literal size>  ::=  S | M | L | XL
```

Calling SQL-PL from Precompiled Programs

The SQL-PL applications can be invoked from within a program by means of the following calls.

This chapter covers the following topics:

- EXEC SQLPL
- EXEC SQL PROC

EXEC SQLPL

```
EXEC SQLPL <sqlpl definition> <end-symbol>


<end-symbol>   see "General Rules" in the
                   "C/C++ Precompiler" or "Cobol Precompiler" manual



 <sqlpl definition>  ::=  <application name>
                          [ VARS    ( <sqlpl var clause>  ) ]
                          [ PARM    ( <parameter clause> ) ]
                          [ OPTIONS ( <form option>, ... ) ]

 <application name>  ::=  [<user name>.]<appl name>.<mod name>
                        |  :<host variable>

 <parameter clause>   ::=  <parameter>,...

 <parameter>  ::=  :<host variable> [ :<indicator variable> ]

 <sqlpl var clause>  ::=  <form var> = <parameter> , ...

 <form var>  ::=  <name> | <name>(parameterspec)

 <form vect slice>  ::=  <name>(parameterspec..parameterspec)

 <form option>  ::=  PROMPT = <char>
                   |  BACKGROUND
                   |  RESTORE
                   |  NOINIT
                   |  [NO]CLEAR
                   |  INPUT     ( <field spec> , ... )
                   |  NOINPUT    ( <field spec> , ... )
                   |  PRINT      [ ( <print optionlist> ) ]
                   |  MARK       ( <field spec> )
                   |  SCREENPOS  ( <parameterspec>,
                                   <parameterspec> )
                   |  FORMPOS    ( <parameterspec>,
                                   <parameterspec> )
                   |  SCREENSIZE ( <parameterspec>
                                   [,<parameterspec>])
                   |  ACCEPT     ( <accept spec> , ...  )
                   |  ATTR       ( <form var> [, <attr spec>] )
                   |  FRAME      [ ( <string const> |
                                     :<host variable> ) ]
```

```
                                  - maximum 76 bytes -
                  |  ACTION ( <parameterspec> )

 <field spec>  ::=  <form var> | <field seqno>
 <field seqno>   ::=  1..255   sequence number of the field
 <attr spec>  ::=  <attr symbol>
 <attr symbol>  ::=  LOW | HIGH | INV | BLK | UNDERL
                  |  ATTR1 | ATTR2 ... | ATTR16
 <parameterspec>  ::=  unsigned integer | <parameter>
 <parameter>  ::=  :<host variable> [ :<indicator variable> ]
 <print optionlist>  ::=  <print option> [, <print optionlist>   ]
 <print option>  ::=  CLOSE
                   |  NEWPAGE     <parameterspec>
                   |  CPAGE       <parameterspec>
                   |  LINEFEED    <parameterspec>
                   |  LINESPACE   <parameterspec>
                   |  PRINTFORMAT <printformat name>

 <printformat name>  ::=  <string const> | :<host variable>
                          up to 18 bytes

 <accept spec>  ::=  <key literal>
                   |  <key literal> = '<key label>'

 <key literal>  ::=  F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9
                   |  F10 | F11 | F12
                   |  HELP (=F10) | UP(=F11) | DOWN(=F12) | ENTER
                   |  UPKEY | DOWNKEY
                   |  LEFTKEY | RIGHTKEY
                   |  ENDKEY | CMDKEY | HELPKEY

 <key label>  ::=  8 characters
```

An SQL-PL module is invoked by EXEC SQLPL.

The parameters of the SQL-PL module are provided with values via PARM. Host variables are assigned to the global variables used in an SQL-PL form by means of VARS. All the host variables are to be declared in the DECLARESECTION. It is not possible to change the host variable values by means of an SQL-PL module. The OPTIONS specification has only an effect if the called SQL-PL module is a form.

The program must ensure that the types of parameters and host variables are compatible with each other. The precompiler only differentiates between numeric and alphanumeric data. In the case of discrepancies the runtime system will return a corresponding error message.

As the result SQL-PL writes the return code (SQLCODE), the return text (SQLERRMC), the last used key (SQLPFKEY), and the sequence number of the field where the cursor was positioned last (SQLCURSOR) into the SQLCA.

The last used key is indicated as a numeric value. Thereby the following assignment is valid:

Basic Function Keys		Additional Release Keys	
Key	SQLPFKEY	Key	SQLPFKEY
F1	1	UPKEY	14
F2	2	DOWNKEY	15
F3	3	LEFTKEY	16

F4	4	RIGHTKEY	17
F5	5	ENDKEY	18
F6	6	CMDKEY	19
F7	7	HELPKEY	20
F8	8		
F9	9		
F10, HELP	10		
F11, UP	11		
F12, DOWN	12		
ENTER	13		

EXEC SQL PROC

```
EXEC SQL PROC <db-procedure>

<db-procedure>  ::=  <db-procedure-name>
                     [ ( <paramterlist> ) ]
<parameterlist> ::=  <parameter> , ...
<parameter>     ::=  :<host variable>
                     [ :<indicator variable> ]
```

This SQL statement calls DB Procedures which have been stored in the database by means of SQL-PL.

Examples

Various examples are provided with the installation. These examples are stored in the directory $DBROOT/demo/eng/SQL-PL. The help information, too, offers numerous examples. These can be read directly into the editor (GET command) and be executed at once.

Restrictions

In the following the restrictions are listed which the SQL-PL program developers must take into account.

- 512 g l o b a l variables
- 255 components per vector variable
- 16 K per string
- 63modules with global or static-local variables per program
- any number of modules without global variables per program
- 255 formal parameters per module
- 255 variables in an SQL statement
- 4 KByte long SQL statements
- 255 fields per form
- 20 arguments for the form call option ATTR
- 20 variable arguments for the form call option INPUT/NOINPUT
- 255 field number arguments for the form call option INPUT