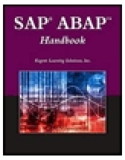


SAP ABAP Handbook

by Kogent Learning Solutions, Inc.

Chapter 9: Modularization Techniques

Overview

Modularization techniques enhance the readability and understandability of large ABAP programs. At times, it becomes difficult to enhance and debug the source code of a lengthy ABAP program. The ABAP language simplifies the debugging process by offering various modularization techniques, such as subroutines, function modules, and source code modules. Modularization techniques help to remove redundancy that occurs when an ABAP program contains the same or similar blocks of statements or the same function is used multiple times. Modularization techniques also improve the structure of an ABAP program and make it easy to read, maintain, and update.

In this chapter, you learn about three modularization techniques: subroutines, function modules, and source code modules. We start the chapter by discussing subroutines, which are modularization units in a program. In addition, you learn how to pass the values to the parameters of these subroutines and terminate them. Next, we discuss function modules that encapsulate the processing logic of a program into the form of a function. Finally, you learn about source code modules, which are collections of ABAP statements that are not required to be included in an ABAP program.

Now, let's begin our discussion with subroutines and learn how to work with them.

Working with Subroutines

A subroutine is a mini-program that consists of a sequence of statements. Subroutines are used to prevent redundancy of the statements in an ABAP program. A subroutine is defined with the help of the `FORM` statement that signifies the start of the subroutine. Within a subroutine, you can define variables, execute ABAP statements to calculate results, and display the calculated result on the screen. You can end the subroutine with the help of the `ENDFORM` statement. The length of the name of a subroutine is limited to 30 characters. Subroutines defined in an ABAP program can be called either in the same ABAP program or from any other ABAP program with the `PERFORM` statement (you learn about `PERFORM` statements later in this chapter).

Note Subroutines cannot be nested. As a best practice, specify the subroutine definition at the end of an ABAP program.

The following syntax shows how to define a subroutine by using the `FORM` statement:

```
FORM sub_name [USING        prm1 TYPE type
                            prm2 LIKE field
                            . . . .
                            VALUE(prm3) TYPE type
                            VALUE (prm4) LIKE field
                            . . . . . ]
[ CHANGING { {VALUE(prm1)}|{prm1 [{TYPE type}|{LIKE field}]}
                 {VALUE(prm2)}|{prm2 [ {TYPE type}|{LIKE
                 field}]}
                 . . . .
} ]
. . . . . .
ENDFORM.
```

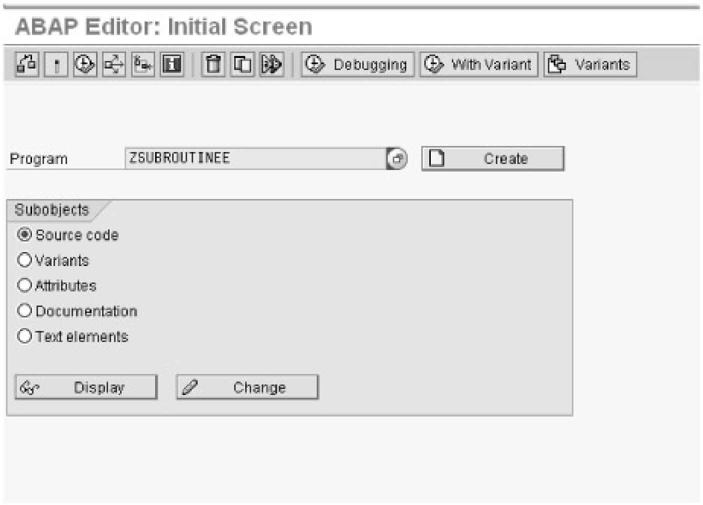
In this syntax, the `sub_name` expression denotes the name of the subroutine, while the `USING` and `CHANGING` clauses define the parameter interface.

Programs for subroutines are written in the ABAP Editor tool of ABAP Workbench. Perform the following steps to create a subroutine program in ABAP Editor:

1. Select `SAP menu > Tools > ABAP Workbench > Development > SE38-ABAP Editor` to start the initial screen of ABAP Editor.

Note You can also simply enter the `SE38` transaction in the command field and click the `Enter` (✓) icon or press the ENTER key to start the initial screen of ABAP Editor.

The initial screen of ABAP Editor appears, as shown in Figure 9.1:



ABAP Editor: Initial Screen

Program ZSUBROUTINEE

Subobjects

● Source code
○ Variants
○ Attributes
○ Documentation
○ Text elements

© SAP AG. All rights reserved.

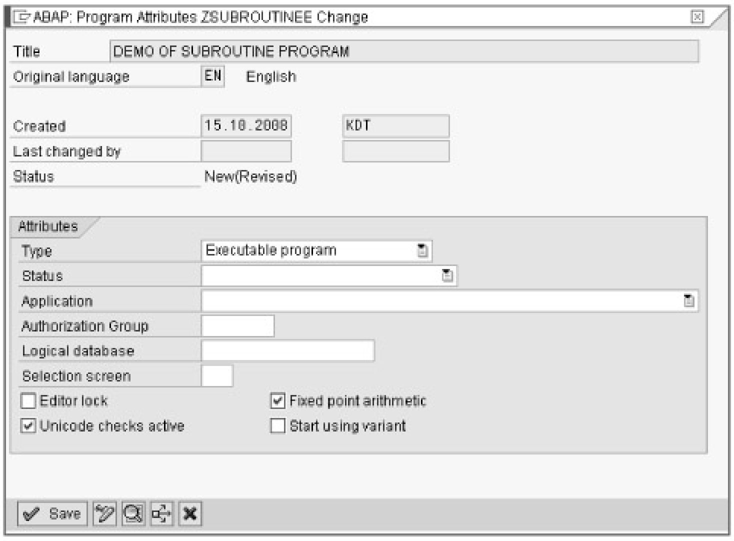
Figure 9.1: Displaying the initial screen of ABAP editor

2. Enter the name of the program in the `Program` field as "ZSUBROUTINEE".

   Note You will notice that the `Source code` radio button under the Subobjects group box is already selected by default. You can select any of the radio buttons depending on the purpose. To learn more about these radio buttons, please refer to Chapter 6.

3. Click the `Create` button.

The `ABAP: Program Attributes ZSUBROUTINE1 Change` dialog box appears, as shown in Figure 9.2:



ABAP: Program Attributes ZSUBROUTINEE Change

Title DEMO OF SUBROUTINE PROGRAM
Original language EN English

Created 15.10.2008 KDT
Last changed by
Status New(Revised)

Attributes

Type Executable program
Status
Application
Authorization Group
Logical database
Selection screen

□ Editor lock ☑ Fixed point arithmetic
☑ Unicode checks active □ Start using variant

© SAP AG. All rights reserved.

Figure 9.2: Displaying the ABAP—program attributes dialog box

4. In the `ABAP Program Attributes` dialog box, enter the title of the program as "DEMO OF SUBROUTINE PROGRAM", select the Type of the program as `Executable program`, and select the `Unicode checks active` check box to make the program compatible.

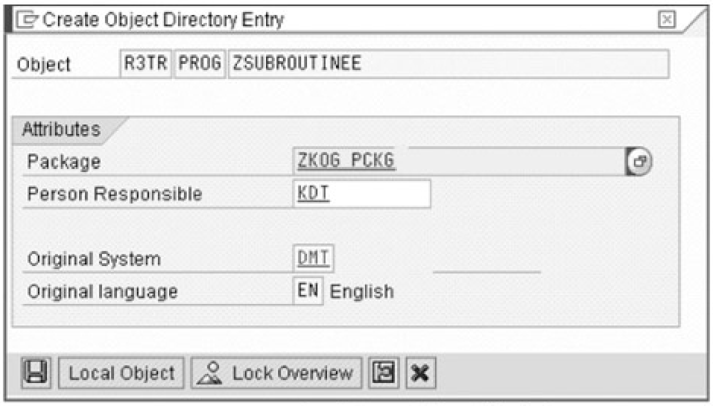
5. Now, click the `Save` button. The `Create Object Directory Entry` dialog box appears, as shown in Figure 9.3:

6. Enter the package name in the `Package` field as ZKOG_PCKG, as shown in Figure 9.3.

7. Click the `Save` (💾) icon. The `Prompt for local Workbench request` dialog box appears, as shown in

Figure 9.4:

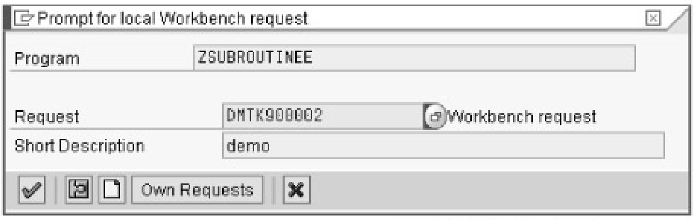
8. Click the `Continue` (✓) icon. The `ABAP Editor: Change Report ZSUBROUTINE` screen appears, as shown in Figure 9.5:



Object	R3TR	PROG	ZSUBROUTINEE
Package	ZKOG_PCKG		
Person Responsible	KDT		
Original System	DMT		
Original language	EN	English	

© SAP AG. All rights reserved.

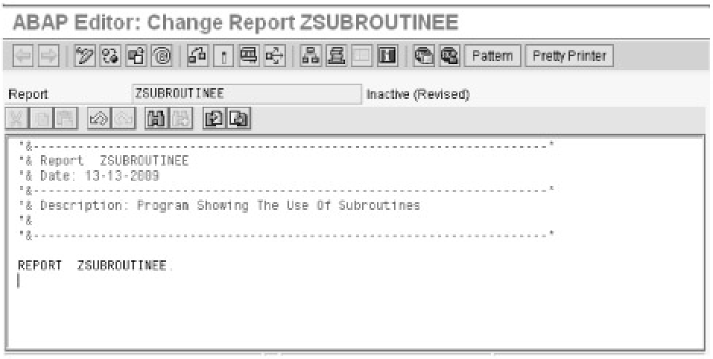
Figure 9.3: Entering the package name



Program	ZSUBROUTINEE
Request	DMTK900002
Short Description	demo

© SAP AG. All rights reserved.

Figure 9.4: Saving the program in a package



Report ZSUBROUTINEE Inactive (Revised)

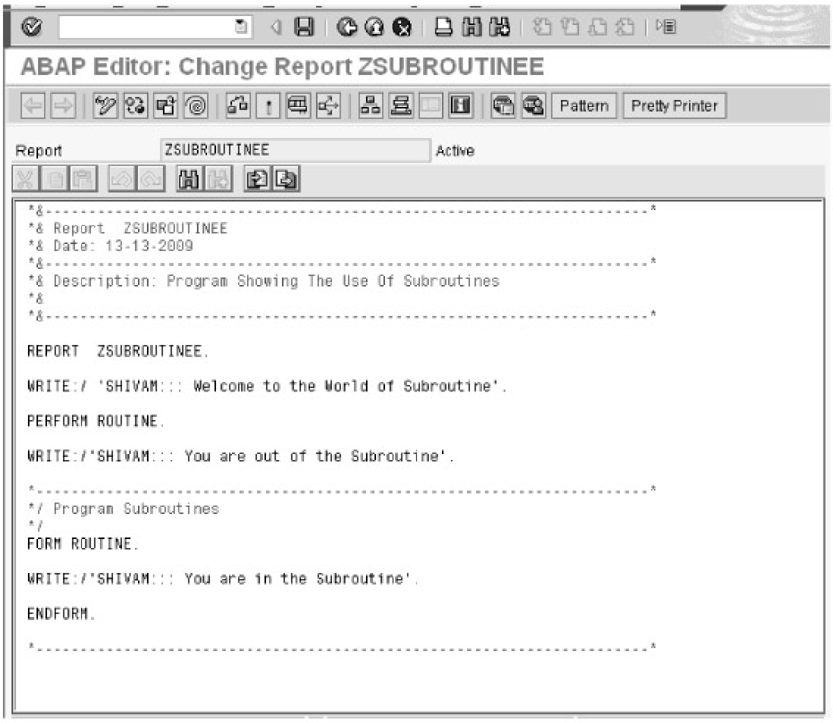
```
*&---------------------------------------------------------------------*
*& Report  ZSUBROUTINEE
*& Date: 13-13-2009
*&---------------------------------------------------------------------*
*& Description: Program Showing The Use Of Subroutines
*&
*&---------------------------------------------------------------------*

REPORT  ZSUBROUTINEE.
```

© SAP AG. All rights reserved.

Figure 9.5: Displaying the ABAP editor screen

In Figure 9.5, notice the generated program name, i.e., ZSUBROUITNEE appears with the `REPORT` statement. In the `ABAP Editor: Change Report ZSUBROUTINE` screen, we have added some comments regarding the date and description of the program.

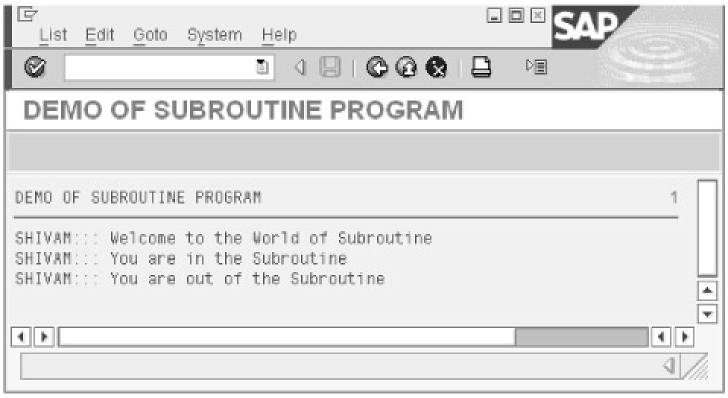
9. Enter the code of an ABAP program. For instance, we have inserted the code for creating a subroutine in Figure 9.6:
10. Click the `Save` (💾) icon, click the `Check` icon, and then click the `Activate` icon.
11. Click the `Direct Processing` icon to see the output of the program. Figure 9.7 shows the output of the program ZSUBROUTINEE:



```
*&---------------------------------------------------------------------*
*& Report  ZSUBROUTINEE
*& Date: 13-13-2009
*&---------------------------------------------------------------------*
*& Description: Program Showing The Use Of Subroutines
*&
*&---------------------------------------------------------------------*

REPORT  ZSUBROUTINEE.

WRITE:/ 'SHIVAM::: Welcome to the World of Subroutine'.

PERFORM ROUTINE.

WRITE:/'SHIVAM::: You are out of the Subroutine'.

*---------------------------------------------------------------------*
*/ Program Subroutines
*/
FORM ROUTINE.

WRITE:/'SHIVAM::: You are in the Subroutine'.

ENDFORM.

*---------------------------------------------------------------------*
```

© SAP AG. All rights reserved.

Figure 9.6: The screen showing the subroutine program



```
DEMO OF SUBROUTINE PROGRAM                                    1

SHIVAM::: Welcome to the World of Subroutine
SHIVAM::: You are in the Subroutine
SHIVAM::: You are out of the Subroutine
```

© SAP AG. All rights reserved.

Figure 9.7: Displaying the output of the subroutine program

In the following sections, we learn to:

- Work with formal and actual parameters
- Handle data in subroutines
- Make internal and external calls
- Pass parameters to subroutines
- Terminate subroutines

Now, let's discuss each in detail.

Working with Formal and Actual Parameters

In subroutines, parameters are defined by using the `FORM` and `PERFORM` statements. When you define the parameters by using the `FORM` statement, the parameters are said to be formal parameters. The parameters defined by using the

PERFORM statement are called actual parameters. USING and CHANGING are additional clauses in the FORM statement, which are used to include or change the type or field of formal parameters by using TYPE and LIKE clauses, respectively. Each clause, whether it is USING or CHANGING, is followed by one or more formal parameters. Formal parameters behave as dynamic local data inside a subroutine. Formal parameters hide the global data objects that have the same name as the formal parameters.

Whenever the subroutine is called, all the formal parameters must be populated with the values of the actual parameters. At the end of the subroutine, the values in the formal parameters are passed back to the corresponding actual parameters. The syntax to define formal parameters is:

```
FORM sub_name USING prml[{TYPE type}|{LIKE field}]
                       prm2[{TYPE type}|{LIKE field}]
                       . . . .
            CHANGING      prm1[{TYPE type}|{LIKE field}]
                       prm2[{TYPE type}|{LIKE field}]
                       . . . .
```

In this syntax, sub_name represents the name of a subroutine, prm1 and prm2 represent formal parameters, and field represents the name of a predefined field. You do not need to allocate memory space to formal parameters because only the address of the actual parameters is transferred to the formal parameters. If the value of the formal parameter is changed, the content of the actual parameter in the calling program also changes. The USING clause is used for the documentation of those parameters that do not change in the subroutine (also known as input parameters), and the CHANGING clause is used for the documentation of those parameters that change in the subroutine (also known as output parameters). It is possible that the values of the actual parameters change during the processing of the subroutine. This change in the values of the actual parameters can be prevented with the help of different clauses, such as USING VALUE and CHANGING VALUE, used in the FORM statement. The values of the actual parameters can be prevented from changing in the following ways:

- **Passing values using input parameters**—Input parameters are used to pass data to subroutines. They are specified after the USING VALUE clause in the FORM statement. The following syntax shows how to define input parameters:

  ```
  FORM sub_name USING VALUE(prm1) [{TYPE type}|{LIKE field}]
                    VALUE(prm2) [{TYPE type}|{LIKE field}]
                        . . . . . .
  ```

- **Passing values using output parameters**—Output parameters are used to pass data from subroutines. The output parameters are specified after the CHANGING VALUE clause in the FORM statement. The following syntax shows how to define output parameters:

  ```
  FORM subr CHANGING VALUE(prm1) [{TYPE type}|{LIKE field}]
                    VALUE(prm2) [{TYPE type}|{LIKE field}]
                        . . . . . .
  ```

Handling Data in Subroutines

A subroutine handles the data of a program by adopting any of the following methods:

- Using global data of a program
- Using data types and data objects in subroutines

Using Global Data of a Program

A subroutine defined in a program can access all the global data of that program. The values stored in global data can be modified by executing a subroutine. To avoid the change in the values of global data, you must use the parameter interface. With the help of parameter interface, a subroutine can perform complex operations on its own data and pass the data back to the actual parameters without affecting the global data of the associated program.

The values in the global objects in a subroutine can be prevented from changing through the use of the LOCAL statement. The following is the syntax of the LOCAL statement:

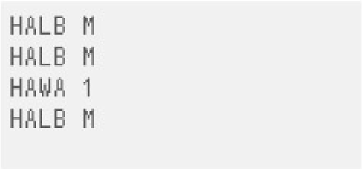
```
LOCAL f.
```

The LOCAL statement should be used in between the FORM and ENDFORM statements. Note that you cannot declare the table work area already defined by the TABLES statement with another TABLES statement inside a subroutine. If you want to use the table work area locally and preserve the contents of the table work area outside the subroutine, you must use the LOCAL statement. Listing 9.1 shows an example of the LOCAL statement:

Listing 9.1: Using the LOCAL statement in subroutines

```
REPORT ZMYSUBROUTINE1.
*------------------------------------------------*
*/ Tables Used in the Program
*/
TABLES MARA.
*------------------------------------------------*

PERFORM routine1.

WRITE: / MARA-MTART, MARA-MBRSH.

PERFORM routine2.

WRITE: / MARA-MTART, MARA-MBRSH.
*------------------------------------------------*
*/ Definition of Subroutine
*/
FORM routine1.

      MARA-MTART = 'HALB'.
      MARA-MBRSH = 'M'.
      WRITE: / MARA-MTART, MARA-MBRSH.

ENDFORM.
*------------------------------------------------*
*/ Definition of Subroutine
*/
FORM routine2.

      LOCAL MARA.
      MARA-MTART = 'HAWA'.
      MARA-MBRSH = '1'.
      WRITE: / MARA-MTART,
      MARA-MBRSH.

ENDFORM.
*------------------------------------------------*
```

In Listing 9.1, notice that when the first subroutine, routine1, is called, the values of the MTART (MATERIAL TYPE) and MBRSH (INDUSTRY SECTOR) fields are displayed. When routine2 is called, the LOCAL statement is used. The LOCAL statement is used to preserve the values of the global data objects. In our case, the global data object is a table named MARA. Any changes made to the fields of this table (MARA) in the routine2 definition are not reflected outside the definition of routine2. Figure 9.8 shows the effect of the LOCAL statement in a subroutine:



```
HALB M
HALB M
HAWA 1
HALB M
```

© SAP AG. All rights reserved.

Figure 9.8: Displaying the effect of the LOCAL statement

Using Data Types and Data Objects in a Subroutine

Data declaration is done in procedures such as subroutines, function modules, and source code modules, to create local data types and data objects that are visible only within that procedure. The following are the two types of local data types and data objects:

- Dynamic data types and data objects

- Static data types and data objects

Dynamic Data Types and Data Objects

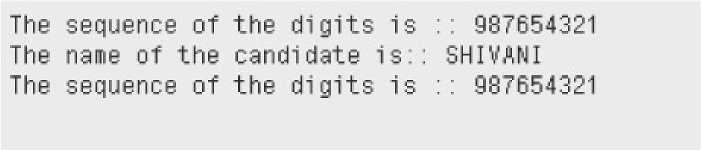
Dynamic data types and data objects are declared in a subroutine with the help of the TYPES and DATA statements. Dynamic data types and data objects exist only when a subroutine is running, and they are deleted when the subroutines end and recreated each time the subroutine is called.

Every subroutine has its own local namespace. A user cannot address the global data type or the data object from within the subroutine whose names are used to declare the names of the local data type or data object. Local data types or data objects hide the global data types declared or data objects that have the same name. The global data types and the data objects can be prevented from being hidden by assigning different names to the local data types and the data objects. Listing 9.2 shows how to use the dynamic data types and data objects:

Listing 9.2: Using dynamic data types and data objects in a subroutine

```
Report ZMYSUBROUTINE2.
*-------------------------------------------------*
*/ Defining Data
*/

TYPES numbers(10) TYPE c.
DATA digits TYPE numbers.
*-------------------------------------------------*
digits = '987654321'.
WRITE:/ 'The sequence of the digits is ::', digits.

PERFORM subroutine.

WRITE:/ 'The sequence of the digits is ::', digits.
*-------------------------------------------------*
*/ Definition of Subroutine
*/

FORM subroutine.

TYPES alphabet(7) TYPE c.
DATA name TYPE alphabet.

name = 'SHIVANI'.
WRITE:/ 'The name of the candidate is::', name.
ENDFORM.
*-------------------------------------------------*
```

In Listing 9.2, a data type named numbers is created with the help of the TYPES statement. A global data object named digits with the data type as numbers is declared. A value has been assigned to the digits data object and is displayed on the output screen. The subroutine named subroutine is called; a local data type, alphabet, and a local data object, name, with the type alphabet is declared inside the subroutine. These local data types and the data objects hide the global data type and data object. Note that the global definitions are valid again after the subroutine has been executed. Figure 9.9 shows the output of Listing 9.2:



```
The sequence of the digits is :: 987654321
The name of the candidate is:: SHIVANI
The sequence of the digits is :: 987654321
```

© SAP AG. All rights reserved.

Figure 9.9: Displaying the dynamic data types and data objects

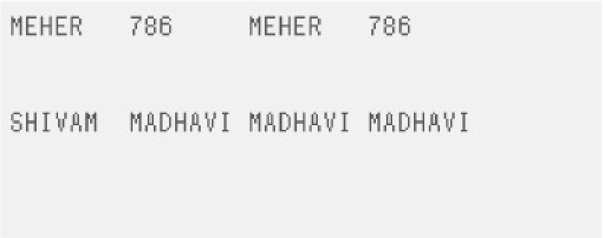
Static Data Types and Data Objects

The data types and data objects defined in a subroutine are known as either static data types and static data objects or local data types and local data objects. Use the STATICS statement if you want to ensure that the values in the local data objects do not change after exiting the subroutine. This statement declares a data object that is globally defined, but is only locally visible from the subroutine in which it is defined. Listing 9.3 shows the use of the STATICS statement in a subroutine:

Listing 9.3: Using the STATICS statement in subroutines

```
Report ZMYSUBROUTINE3.

PERFORM subroutine1.
PERFORM subroutine1.
SKIP 2.
PERFORM subroutine2.
PERFORM subroutine2.
*----------------------------------------------*
*/ Definition of Subroutine
*/
FORM subroutine1.

TYPES alphabets(7) TYPE c.
DATA name TYPE alphabets value 'MEHER'.
WRITE name.
name = '786'.
WRITE name.

ENDFORM.

*----------------------------------------------*
*/ Definition of Subroutine
*/
FORM subroutine2.

TYPES alphabets(7) TYPE c.
STATICS name TYPE alphabets VALUE 'SHIVAM'.
WRITE name.
name = 'MADHAVI'.
WRITE name.

ENDFORM.
*----------------------------------------------*
```

Listing 9.3 shows that two similar subroutines, subroutine1 and subrouitne2, are defined. In subroutine2, the STATICS statement is used instead of the DATA statement to declare a data object, name. Each time subroutine1 is called, the name data object is initialized, but subroutine1 maintains the original value for subroutine2. The VALUE clause in the STATICS statement works only when subroutine2 is called for the first time. Figure 9.10 shows the output of Listing 9.3:



```
MEHER   786     MEHER   786


SHIVAM  MADHAVI MADHAVI MADHAVI
```

© SAP AG. All rights reserved.

Figure 9.10: Displaying the static data types and data objects

Using Local Field Symbols

When a field symbol is defined within a subroutine, it is called a local field symbol. This is because a local field symbol does not have its scope and cannot be used outside the subroutine definition. A local field symbol is defined in a subroutine by using the FIELD-SYMBOLS statement. The following are some rules that apply to local field symbols:

- Addressing the local field symbols outside a subroutine is not possible.
- Whenever a subroutine is called, no field is assigned to a local field symbol.
- The names of the local field symbols can be the same as the names of the global field symbols (field symbols defined outside the subroutine).

Within a subroutine, local copies of the global data objects can be created on the local stack. The following syntax creates local copies of the global data object by using local field symbols and the ASSIGN statement:

```
ASSIGN LOCAL COPY OF field TO <fs>.
```

In this syntax, the ASSIGN statement helps an SAP system place the copy of the specified global data object, represented by the field expression, on the local stack.

In a subroutine, you can access and change the copy of the global data object without changing the original value of the global data object. You can access the copy of the global data object by writing the field symbol in the <fs> expression. The LOCAL COPY OF clause can be used with all the variants of the ASSIGN statement except ASSIGN COMPONENT. Table 9.1 shows different syntaxes of the ASSIGN statement:

Table 9.1: Variations of the ASSIGN statement

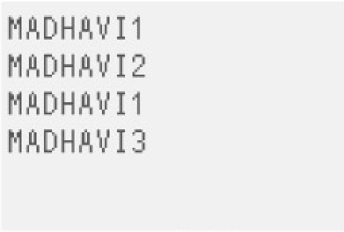
Variation	Description
ASSIGN LOCAL COPY OF INITIAL field TO <fs>	Creates an initialized copy of the field global data object on the stack without transporting the field content.
ASSIGN LOCAL COPY OF INITIAL LINE OF itab TO <fs>	Creates an initial copy of the lines of a global internal table, itab, on the stack.
ASSIGN LOCAL COPY OF INITIAL LINE OF (field) TO <fs>	Creates an initial copy of the lines of a global internal table, itab, on the stack. The internal table is specified dynamically, similar to the contents of the global data object field.

Listing 9.4 shows how to use the ASSIGN statement to declare local field symbols in a program:

Listing 9.4: Using the ASSIGN statement to declare local field symbols

```
Report: ZMYSUBROUTINE4.
*------------------------------------------------*
*/ Defining the Data
*/
DATA name(10) TYPE c VALUE 'MADHAVI1'.
*------------------------------------------------*

PERFORM assignment.
Write / name.

*------------------------------------------------*
*/ Definition of Subroutine
*/
FORM assignment.

      FIELD-SYMBOLS <fs> TYPE ANY.
      ASSIGN LOCAL COPY OF name TO <fs>.
      Write / <fs>.
      <fs> = 'MADHAVI2'.
      Write / <fs>.
      ASSIGN name to <fs>.
      Write / <fs>.
      <fs> = 'MADHAVI3'.

ENDFORM.
*------------------------------------------------*
```

In Listing 9.4, the name data object is assigned to the <fs> local field symbol in the assignment subroutine. A copy of the name `data object` is placed on the local data stack. This local copy can only be read and changed by addressing the field symbol represented by the `<fs>` expression in the `ASSIGN` statement. The global data object is not affected by the operations on the local copy. If the data object is assigned to the field symbol without using the `LOCAL COPY OF` clause, the field symbol points directly to the global data object. Now, if you perform any operations on the global data object, it affects the global field. Figure 9.11 shows the output of Listing 9.4:



```
MADHAVI1
MADHAVI2
MADHAVI1
MADHAVI3
```

© SAP AG. All rights reserved.

Figure 9.11: Displaying the output of the local field symbol

Making Internal and External Calls

The subroutines created in ABAP can be called only by using the `PERFORM` statement. The subroutines created in ABAP programs are of two types, internal subroutines and external subroutines. Internal subroutines are defined in the same program in which they are called, and external subroutines are defined in one program and called in another program. The `PERFORM` statement is used to call the internal as well as external subroutines. Therefore, depending on the type of the subroutines, the call can be divided into the following two categories:

- Internal calls
- External calls

Internal Calls

In this type of subroutine call, the definition of the subroutine as well as the calling procedure exists in the same ABAP program. The call is made with the help of the `PERFORM` statement. The following syntax is used for the `PERFORM` statement:

```
PERFORM sub_name   [USING      prm1 prm2 . . . . . ]
                   [CHANGING      prm1 prm2. . . . ]
```

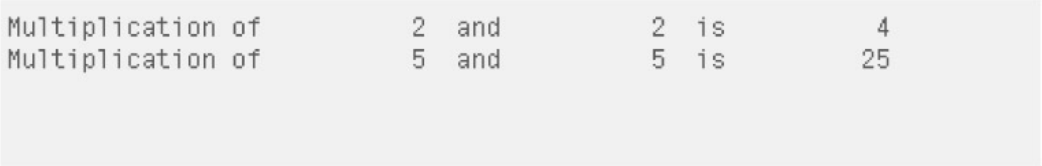
In this syntax, the `sub_name` expression represents the name of the subroutine, and `prm1` and `prm2` represent the actual parameters. All the global data of the calling program can be accessed by the internal subroutine. The clauses used in the `PERFORM` statement are the same as those used with the `FORM` statement. Listing 9.5 shows how to use the `PERFORM` statement while making an internal call in a program:

Listing 9.5: Using the PERFORM statement in an internal call

```
Report ZMYSUBROUTINE5.
*-----------------------------------------------*
*/ Variables Used
Data: number1     type i,
          number2    type i,
          number3    type i.
*-----------------------------------------------*

number1 = 2.
number2 = 2.

PERFORM multiply.

number1 = 5.
number2 = 5.

PERFORM multiply.
*-----------------------------------------------*
```

```
*/ Definition of Subroutine
*/
FORM multiply.
  number3 = number1 * number2.
PERFORM output.
ENDFORM.
*-----------------------------------------------*
*/ Definition of Subroutine
*/
FORM output.

  WRITE: / 'Multiplication of', number1, 'and', number2, 'is',
  number3.
ENDFORM.
*-----------------------------------------------*
```

In Listing 9.5, the two subroutines, named `multiply` and `output`, are defined at the end of the program. The `multiply` subroutine is called by the program, which in turn calls the `output` subroutine. The subroutines have access to the `number1`, `number2`, and `multiply` global data objects. Figure 9.12 shows the multiplication of the two numbers:



```
Multiplication of          2  and          2  is          4
Multiplication of          5  and          5  is         25
```

© SAP AG. All rights reserved.

Figure 9.12: Displaying the multiplication of number1 and number2

External Calls

In the external call of a subroutine, a subroutine is defined in one program and is called in another program. Sometimes, you might also have a separate ABAP program, which consists of only subroutines. Note that the name of the ABAP program that needs to be called must be known. The following syntax is used to make external calls:

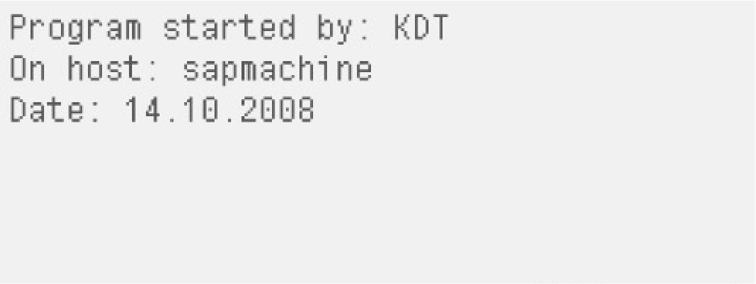
```
PERFORM sub_name (prog) [USING    prm1 prm2 . . . ]
              [CHANGING prm1 prm2 . . . ] [IF FOUND].
```

In this syntax, the name of the ABAP program is represented by the `(prog)` expression, which is defined statically. The `IF FOUND` clause is used to prevent the runtime error, which may occur if the ABAP program `(prog)` does not contain a subroutine named `sub_name`. If the mentioned subroutine, `sub_name`, is not found, the SAP system ignores the `PERFORM` statement. Listing 9.6 shows the use of the `PERFORM` statement in an external call:

Listing 9.6: Using the PERFORM statement in an external call

```
Report ZMYSUBROUTINE7.
PERFORM systemdata.
*--------------------------------------------------*
*/ Definition of Subroutine
*/
FORM systemdata.
      WRITE: / 'Program started by:', sy-uname,
              / 'On host:', sy-host,
              / 'Date:', sy-datum.

ENDFORM.
*--------------------------------------------------*
```

In Listing 9.6, the information related to an SAP system, such as the system user name, host information, and the system date on which the program is created, is displayed with the help of a subroutine named `systemdata`. Figure 9.13 shows the output of Listing 9.6:



```
Program started by: KDT
On host: sapmachine
Date: 14.10.2008
```

© SAP AG. All rights reserved.

Figure 9.13: Displaying the output of system data

Now, the `systemdata` subroutine can also be called from some other program. Listing 9.7 shows how an external call is made:

Listing 9.7: Calling a subroutine externally

```
REPORT ZEXTERNALSUB.
PERFORM systemdata (ZSUBROUTINE8) IF FOUND.
```

In case of an external call, we give the name of the subroutine to be called, i.e., `systemdata`, and the name of the ABAP program, ZSUBROUTINE8, which contains the definition of the called subroutine. The output of Listing 9.7 appears, as shown in Figure 9.13.

In addition to the external call of the subroutine, you can specify the name of the program in which the subroutine occurs dynamically at runtime, as shown in the following syntax:

```
PERFORM sub_name [IN PROGRAM (fprog)] [USING     prm1
prm2 . . . ]
                      [CHANGING prm1 prm2 . . . ]
                         [IF FOUND].
```

In this syntax, the `sub_name` expression represents the name of the subroutine and the `(fprog)` expression shows the name of the external program containing the subroutine.

Apart from the internal and external call of the subroutine, a user can also call a subroutine from a list of subroutines. The following syntax shows how to call a subroutine from a list of subroutines:

```
PERFORM <idx> OF <subr1> <subr2> . . . . . <subrn>.
```

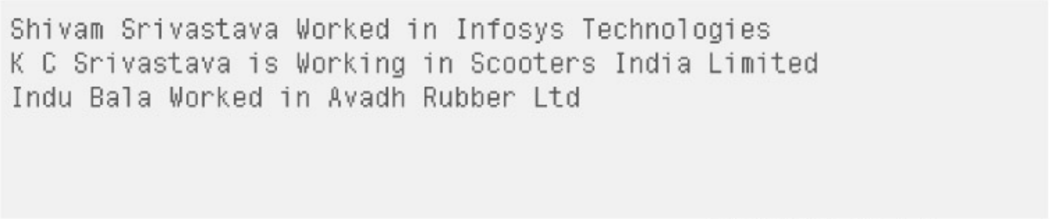
In this syntax, the SAP system calls a subroutine from the subroutine list represented by the `<subr1>`, `<subr2>`, and `<subrn>` expressions. The position of the called subroutine in the subroutine list is denoted by the `<idx>` expression. The `<idx>` expression can be a literal or a variable. Listing 9.8 shows how to call subroutines from a list:

Listing 9.8: Calling subroutines from a subroutine list

```
Report ZMYSUBROUTINE8.
DO 3 TIMES.
PERFORM SY-INDEX OF first second third.
ENDDO.
*---------------------------------------------------*
*/ Definition of Subroutine
*/
FORM first.
WRITE / 'Shivam Srivastava Worked in Infosys Technologies'.
ENDFORM.
*---------------------------------------------------*

*/ Definition of Subroutine
*/

FORM second.
WRITE / 'K C Srivastava is Working in Scooters India Ltd'.
ENDFORM.
```

```
*---------------------------------------------------*
*/ Definition of Subroutine
*/
FORM third.
WRITE / 'Indu Bala Srivastava Worked in Avadh Rubber Ltd'.
ENDFORM.
*---------------------------------------------------*
```

In Listing 9.8, the three subroutines, named `first`, `second`, and `third`, are called consecutively from the subroutine list. Figure 9.14 shows the output of Listing 9.8:



```
Shivam Srivastava Worked in Infosys Technologies
K C Srivastava is Working in Scooters India Limited
Indu Bala Worked in Avadh Rubber Ltd
```

Figure 9.14: Calling a subroutine from a list

Passing Parameters to Subroutines

If a subroutine contains a parameter interface, you must provide values to all the formal parameters. The sequence of the actual parameters is essential whenever the values are passed. Consequently, the value of the first actual parameter in the list of parameters is passed to the first formal parameter, the value of the second actual parameter is passed to the second formal parameter, and so on.

The technical attributes of the actual parameters must be compatible with the type specified for the corresponding formal parameter. Actual parameters can be any data objects or field symbols of the calling program. The following are the three methods to pass parameters to a subroutine:

- Passing by reference
- Passing by value
- Passing by value and result

Table 9.2 shows different kinds of additions that can be used in the `FORM` statement to identify the type of method used in an ABAP program:

Table 9.2: Additions used in the FORM statement

Addition	Name of the Method	Description
USING `prm` OR CHANGING `prm`	Pass by reference	Shows that the parameters are passed by reference. The pass by reference method is very effective because it passes a pointer to the original memory location of the passed variable.
USINGVALUE (`prm`)	Pass by value	Shows that the parameters are passed by value. This method ensures that no change is made to the memory of the passed variable. A new memory location is allocated to the passed variable within a subroutine, which becomes freed as soon as the subroutine ends.
CHANGING VALUE (v1)	Pass by value and result	Shows that the parameters are passed by value and result. This method is similar to the pass by value method, but the content of the new memory location is copied back into the original memory.

Note The following points must be kept in mind to avoid errors while passing parameters:

- The `PERFORM` and `FORM` statements must contain the same number of parameters.
- The way in which the `PERFORM` and `FORM` statements are coded may differ.
- You cannot use the `VALUE` clause on the `PERFORM` statement.

- The `USING` clause must always be used before the `CHANGING` clause.
- You can use the `USING` and `CHANGING` clause only once in a statement.

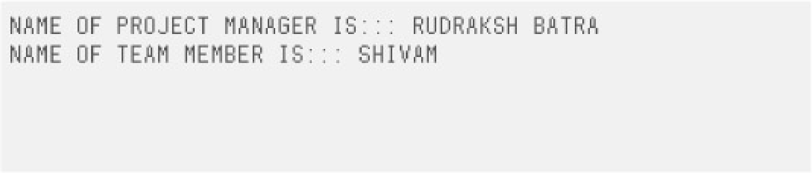
The Pass by Reference Method

In the pass by reference method of passing parameters, no new memory location is allocated for the value. Instead, a pointer to the original memory location is passed. If the value of the variable is changed within a subroutine, the original memory location is changed immediately. Listing 9.9 shows how to use this method:

Listing 9.9: The pass by reference method

```
Report ZMYSUBROUTINE9.
*--------------------------------------------------*
*/ Defining Variable
*/
DATA: pmname(35) VALUE 'RUDRAKSH BATRA'.
*--------------------------------------------------*

WRITE: 'NAME OF PROJECT MANAGER IS:::', pmname.

PERFORM routine USING pmname.
WRITE:/ pmname1.

*--------------------------------------------------*
*/ Subroutine Definition
*/
FORM routine USING pmname1.

pmname1 = 'NAME OF TEAM MEMBER IS::: SHIVAM'.
ENDFORM.
*--------------------------------------------------*
```

In Listing 9.9, a memory location is allocated to the `pmname` variable. Now, let's assume that the assigned memory location is 2000. When a subroutine named `routine` is called, the `USING VALUE` clause on the `FORM` statement causes the `pmname` variable to be passed by reference. Therefore, `pmname1` acts as a pointer to the memory location 2000. The assignment to the variable `pmname1`, changes the contents of the memory location 2000 to "NAME OF TEAM MEMBER IS::: SHIVAM", which previously was "RUDRAKSH BATRA". Figure 9.15 shows the output of Listing 9.9:



```
NAME OF PROJECT MANAGER IS::: RUDRAKSH BATRA
NAME OF TEAM MEMBER IS::: SHIVAM
```

© SAP AG. All rights reserved.

Figure 9.15: Displaying the output of the pass by reference method

Pass by Value Method

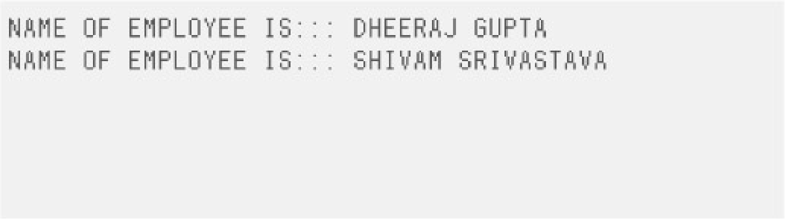
In the pass by value method of passing parameters, a new memory location is allocated for the value being passed. A new memory location is allocated when the subroutine is called and is released when the execution of the subroutine is completed. Therefore, any references made to the parameter are actually the references made to the unique memory area allocated, which are valid only within the subroutine. The original value of the variable remains unchanged even if you change the value of the parameter. Listing 9.10 demonstrates the pass by value method:

Listing 9.10: The pass by value method

```
Report ZMYSUBROUTINE10.
*----------------------------------------------------*
*/ Defining Variable
```

```
DATA: kogemp(36) value 'SHIVAM SRIVASTAVA'.
*----------------------------------------------------*

PERFORM routine USING kogemp.
WRITE:/ 'NAME OF EMPLOYEE IS:::',kogemp.

*----------------------------------------------------*
*/ Definition of Subroutine
*/
FORM routine USING VALUE(kogemp1).
kogemp1 = 'NAME OF EMPLOYEE IS::: DHEERAJ GUPTA'.
WRITE:/ kogemp1.
ENDFORM.
*----------------------------------------------------*
```

In Listing 9.10, you see that a memory location is allocated to the `kogemp` variable. When the `routine` subroutine is called, the `USING VALUE (kogemp1)` clause on the `FORM` statement causes the `kogemp` variable to be passed by value. Therefore, the `kogemp1` variable refers to a new memory location that is independent of the memory location used by the `kogemp` variable. Moreover, the content of the memory location for the `kogemp` variable remains unchanged. Figure 9.16 shows the output of Listing 9.10:



```
NAME OF EMPLOYEE IS::: DHEERAJ GUPTA
NAME OF EMPLOYEE IS::: SHIVAM SRIVASTAVA
```

© SAP AG. All rights reserved.

Figure 9.16: Displaying the output of the pass by value method

The Pass by Value and Result Method

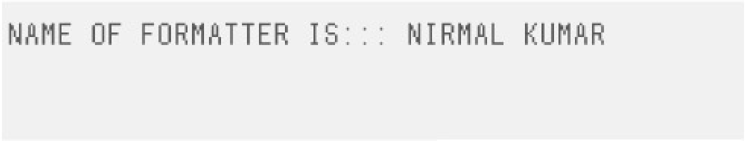
In the pass by value and result method, a new memory location is allocated to the variable passed. In addition to this, an independent copy of the variable is maintained at the new memory location. The value at this new memory location is freed when the subroutine ends. When the `ENDFORM` statement is executed, the value in the new memory location is copied to the original memory location. Listing 9.11 shows how to use this method:

Listing 9.11: The pass by value and result method

```
Report ZOURSUBROUTINE.
*--------------------------------------------------*
*/ Defining Variable
*/
DATA: kogemp(40) value 'Satendra Pal Chopra'.
*--------------------------------------------------*

PERFORM Routine USING kogemp.
WRITE:/ 'HEAD OF ACHEIVERS TEAM IS:::', kogemp.
*--------------------------------------------------*
*/ Definition of Subroutine
FORM Routine CHANGING value(kogemp1).
Kogemp1 = 'NAME OF FORMATTER IS::: NIRMAL KUMAR'.
ENDFORM.
*--------------------------------------------------*
```

In Listing 9.11, a memory location is allocated to the `kogemp` variable. When the `Routine` subroutine is called, the `CHANGING VALUE (kogemp1)` clause on the `FORM` statement causes the `kogemp` variable to be passed by value and result. Therefore, the `kogemp1` variable refers to a new memory location that is independent of `kogemp`. In Listing 9.11, the value of `kogemp` is copied automatically to the memory location of `kogemp1`. In addition, the content of the memory location for `kogemp1` is changed. The (`Kogemp1 = 'NAME OF FORMATTER IS::: NIRMAL KUMAR'`) line of Listing

9.11 copies the value of the `kogemp1` variable back to the `kogemp` variable, and the value in the `kogemp1` variable is now freed. Figure 9.17 shows the output of Listing 9.11:



```
NAME OF FORMATTER IS::: NIRMAL KUMAR
```

© SAP AG. All rights reserved.

Figure 9.17: Displaying the output of the pass by value and result method

Terminating Subroutines by Using *EXIT* and *CHECK* Statements

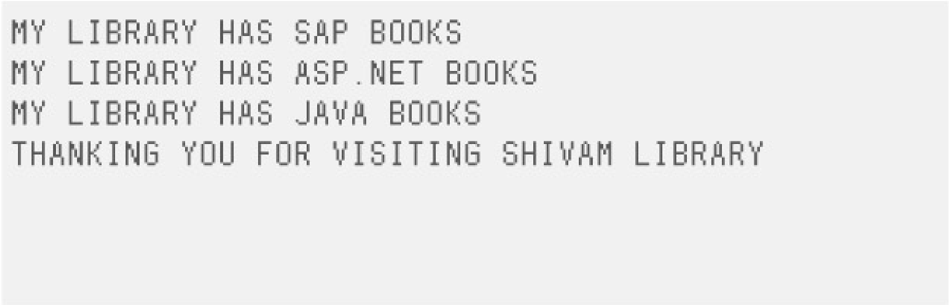
Under normal conditions, the subroutine ends when the ABAP program encounters the `ENDFORM` statement. However, the subroutine can also be terminated by using the `EXIT` or `CHECK` statements at any time. If a subroutine is ended with the `EXIT` or `CHECK` statement, the current values of the output parameters (`CHANGING` parameters passed by value) are returned to the corresponding actual parameter.

The `EXIT` statement is used to terminate a subroutine without any condition. After exiting from the subroutine, the SAP system continues the processing of the ABAP program, executing the statements after the `PERFORM` statement. Listing 9.12 shows how to use the `EXIT` statement in a subroutine:

Listing 9.12: The EXIT statement in a subroutine

```
Report ZMYLIBRARY.
PERFORM LIBRARY.
WRITE / 'THANKING YOU FOR VISITING SHIVAM LIBRARY'.
*---------------------------------------------------*
*/ Definition of Subroutine
*/
FORM LIBRARY.
WRITE / 'MY LIBRARY HAS SAP BOOKS '.
WRITE / 'MY LIBRARY HAS ASP.NET BOOKS'.
WRITE / 'MY LIBRARY HAS JAVA BOOKS'.
EXIT.
WRITE 'MY LIBRARY does not have novels '.
ENDFORM.
*---------------------------------------------------*
```

In Listing 9.12, a subroutine named `LIBRARY` is defined. This subroutine is ended when the third `WRITE` statement is executed. Figure 9.18 shows the output of Listing 9.12:



```
MY LIBRARY HAS SAP BOOKS
MY LIBRARY HAS ASP.NET BOOKS
MY LIBRARY HAS JAVA BOOKS
THANKING YOU FOR VISITING SHIVAM LIBRARY
```

© SAP AG. All rights reserved.

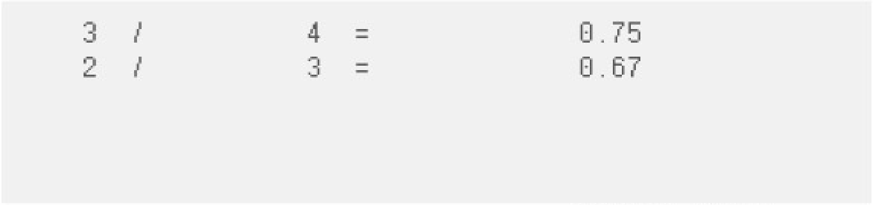
Figure 9.18: Showing the function of the EXIT statement

The `CHECK` statement is used to terminate a subroutine conditionally. The subroutine is ended when the condition specified in the `CHECK` statement evaluates to false. When the subroutine ends, the program continues to execute the statements following the `PERFORM` statement. Listing 9.13 shows how to use the `CHECK` statement in a program:

Listing 9.13: The CHECK statement in a subroutine

```
Report ZMYCHECK.
*---------------------------------------------------*
*/ Defining Variable
*/
DATA: Number1 TYPE I,
       Number2 TYPE I,
       R TYPE P DECIMALS 2.
*---------------------------------------------------*
Number1 = 3.
Number2 = 4.
PERFORM DIVISION USING Number1 Number2 CHANGING R.
Number1 = 5.
Number2 = 0.
PERFORM DIVISION USING Number1 Number2 CHANGING R.
Number1 = 2.
Number2 = 3.

PERFORM DIVISION USING Number1 Number2 CHANGING R.
*---------------------------------------------------*
*/ Definition of Subroutine
*/
FORM DIVISION USING Number1 Number2 CHANGING R.
CHECK Number2 NE 0.
R = Number1 / Number2.
WRITE: / Number1, '/', Number2, '=', R.
ENDFORM.
*---------------------------------------------------*
```

In Listing 9.13, the SAP system terminates the execution of the DIVISION subroutine during the second call of the DIVISION subroutine. It is because during the second call, the value of NUMBER2 is zero, and the condition checked by the CHECK statement evaluates to false. Figure 9.19 shows the output of Listing 9.13:



```
    3  /            4  =                 0.75
    2  /            3  =                 0.67
```

© SAP AG. All rights reserved.

Figure 9.19: Showing the functioning of the CHECK statement

Note The EXIT and CHECK statements function differently in loops and subroutines.

Function Modules

Function modules are ABAP routines that are administered in a central function library. Function modules are assigned to a function pool, also called function group. A function group is a container where all the function modules are connected logically to each other. Function modules apply across applications and are available throughout the SAP system. In other words, function modules are used in remote communication between SAP R/3 system s or between an SAP R/3 system and a non-SAP R/3 system. Function modules also support exception handling and help update databases.

Function groups and function modules are not defined in the source code of a program. Instead, function modules are created with the help of an ABAP Workbench tool known as Function Builder. Moreover, Function Builder can also be used to test function modules without including them in an ABAP program.

Function modules encapsulate program code and provide an interface for data exchange, similar to subroutines. Function modules must belong to a pool called a function group. Some of the features of function modules are:

- A fixed interface is provided by a function module to exchange data between the function module and an ABAP program. Due to this interface, you can pass input and output parameters easily to and from the function module.

- No data can be exchanged between the calling program and function module by using the shared memory area because function modules always use their own memory.
- A function module can be called in an ABAP program by using the `CALL FUNCTION` statement.

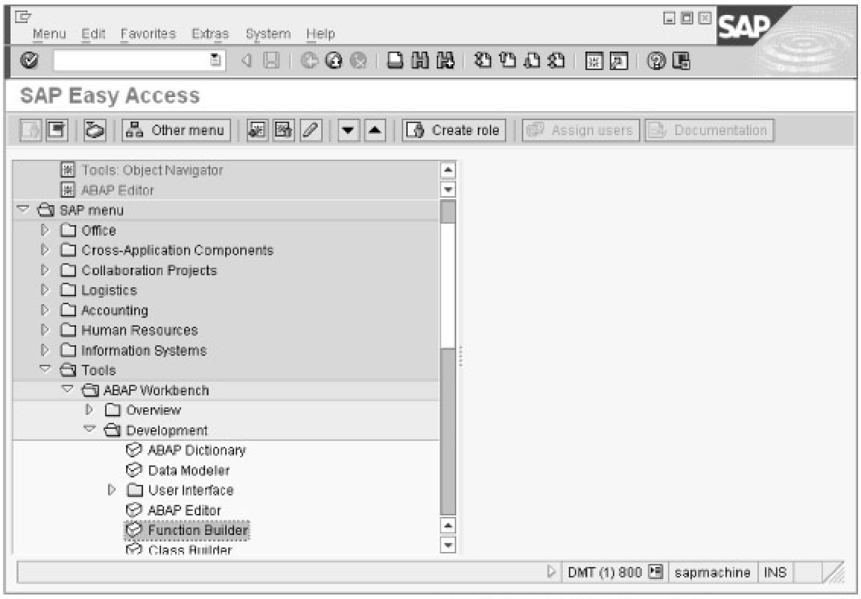
Use of function modules within an ABAP program can be determined by the function module interface parameters. Table 9.3 describes the interface parameters used for function modules:

Table 9.3: Interface parameters of function modules

Interface Parameter	Description
Import	Transfers the values of the parameters from the calling program to the function module. The contents of the import parameter cannot be overwritten at runtime.
Export	Transfers the values of the parameters from the function module back to the calling program.
Changing	Acts as both the import and export parameters simultaneously. The original value of a changing parameter is transferred from the calling program to the function module. The initial value of the changing parameter can be changed by the function module and can be returned back to the calling program.
Tables	Specifies the names of the internal tables that can be imported and exported. The content of the internal tables is transferred from the calling program to the function module. The content of the internal tables can also be altered by the function module and returned to the calling program. Remember that table parameters are passed by reference.
Exceptions	Show the error situations that occur within a function module. These parameters are primarily used by the calling program to identify any error in the function module.

Creating Function Modules

As learned earlier, Function Builder is a tool of ABAP Workbench, which is used to create, test, and administer function modules in an integrated environment. The initial screen of the Function Builder can be started from the initial screen, i.e., `SAP Easy Access`, by selecting `SAP Menu > Tools > Development > Function Builder`, as shown in Figure 9.20:

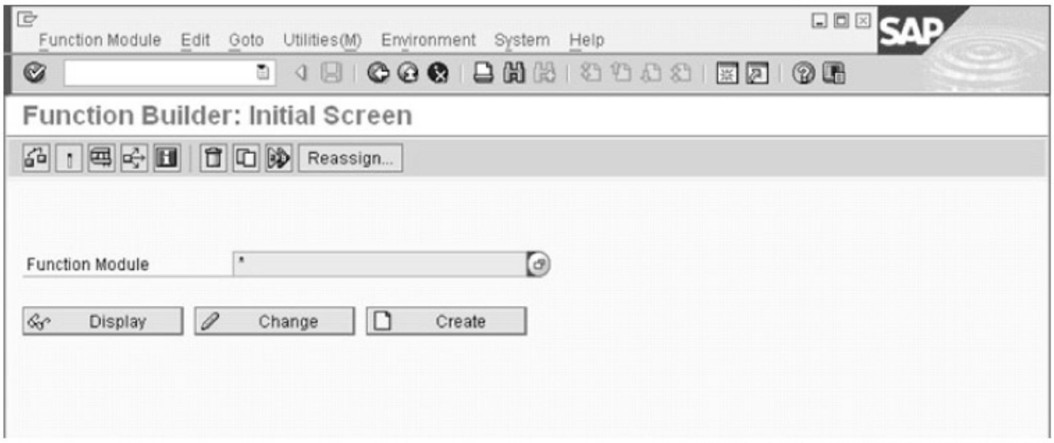


© SAP AG. All rights reserved.

Figure 9.20: Accessing function builder

Note An alternative way to access Function Builder is to enter the `SE37` transaction code in the `Command` field.

Figure 9.21 shows the initial screen of Function Builder:

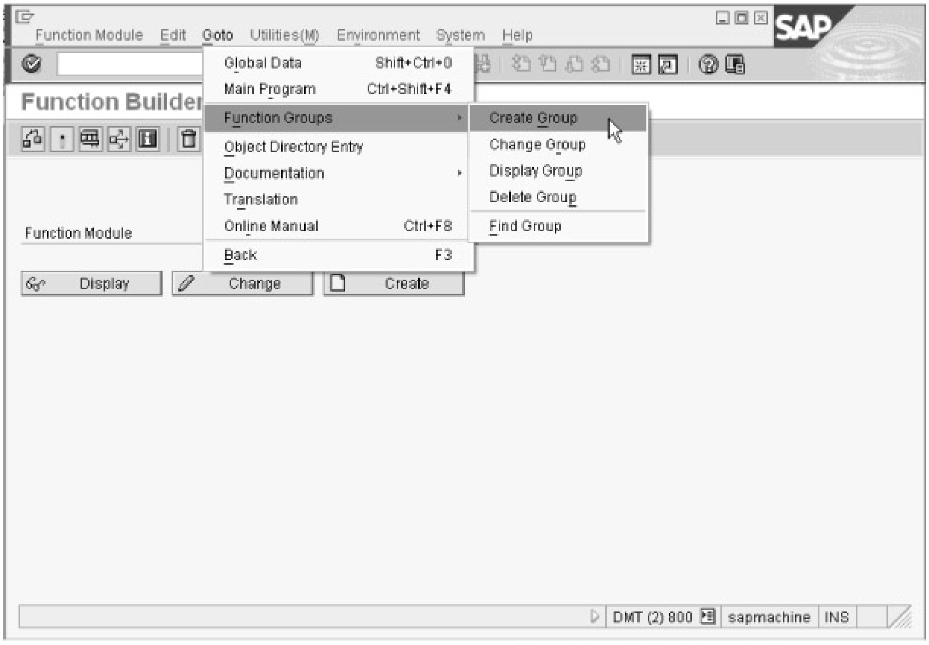

Function Module Edit Goto Utilities(M) Environment System Help
SAP
Function Builder: Initial Screen
Reassign...
Function Module
*
Display
Change
Create

© SAP AG. All rights reserved.

Figure 9.21: The initial screen of function builder

You can create a function module in the SAP R/3 system by performing the following steps:

1. Before creating a function module, a function group is created that actually holds the created function module. Function groups are created from the initial screen of the function builder. To create a function group, select `Go To > Function Groups > Create Group`, as shown in Figure 9.22:

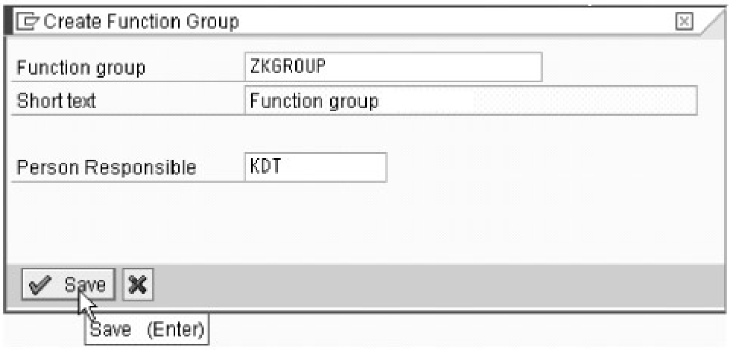


© SAP AG. All rights reserved.

Figure 9.22: Creating a function group

The `Create Function Group` dialog box appears.

2. Enter the name of the function group that you want to create in the `Function` group field, and a short description in the `Short text` field. Finally, click the `Save` button, as shown in Figure 9.23:



Create Function Group

Function group: ZKGROUP

Short text: Function group

Person Responsible: KDT

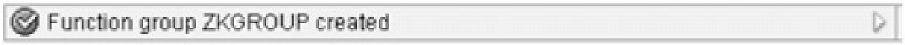
© SAP AG. All rights reserved.

Figure 9.23: Creating a function group

The `Create Object Directory Entry` dialog box appears.

3. In the `Create Object Directory Entry` dialog box, enter ZKOG_PCKG as the name of the package in the `Package` field. Click the `Save` (💾) icon.

A confirmation message appears on the status bar of the initial screen, as shown in Figure 9.24:



Function group ZKGROUP created

© SAP AG. All rights reserved.

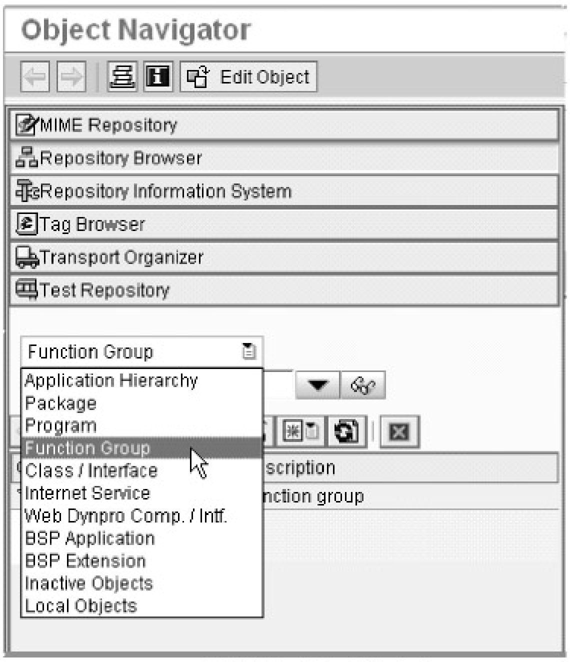
Figure 9.24: Confirmation message

Now, a function group named ZKGROUP has been created and saved in the SAP system.

4. Start a new session by clicking the `Create New Session` (▣) icon on the standard toolbar.
5. Enter the `SE80` transaction in the command field and press the `ENTER` key. The `Object Navigator` screen appears (Figure 9.25). The initial screen of Object Navigator appears where you select the Function Group option from the drop-down menu, as shown in Figure 9.25:
6. Enter `ZKGROUP` as the name of the function group in the next field below the `Function Group` field, as shown in Figure 9.26:
7. Press the ENTER key. The name of the function group `(ZKGROUP)` appears in the Object Name column, and the respective description of the function module appears under the Description column, as shown in Figure 9.27:

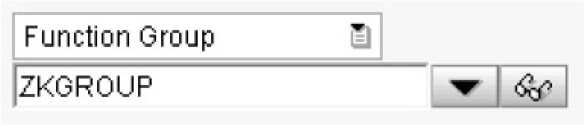
    Note When you create a function group in an SAP system, the system automatically generates a main program and its subsequent include programs. The name of the main program always contains the prefix `SAPL`, and is followed by the name of the function group. Therefore, in our case, the main program is `SAPLZKGROUP` and the include programs are `LZKGROUPTOP` and `LZKGROUPUXX`. In Figure 9.27, you can see the `Includes` folder in Object Navigator that contains the `LZKGROUPTOP` and `LZKGROUPUXX` include programs.

8. Right-click the name of the function group and select the Activate option from the drop-down menu to activate the function group, as shown in Figure 9.28:



Function Group

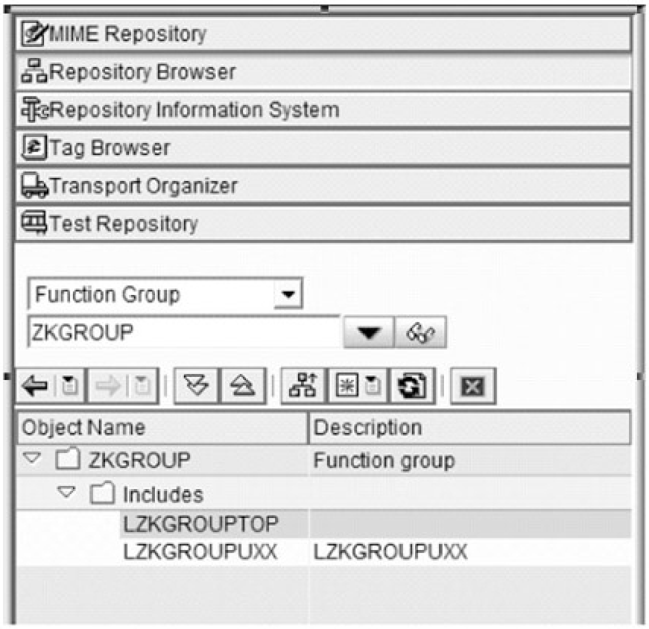
Figure 9.25: Selecting the function group



Function Group

ZKGROUP

Figure 9.26: Entering the name of the function group

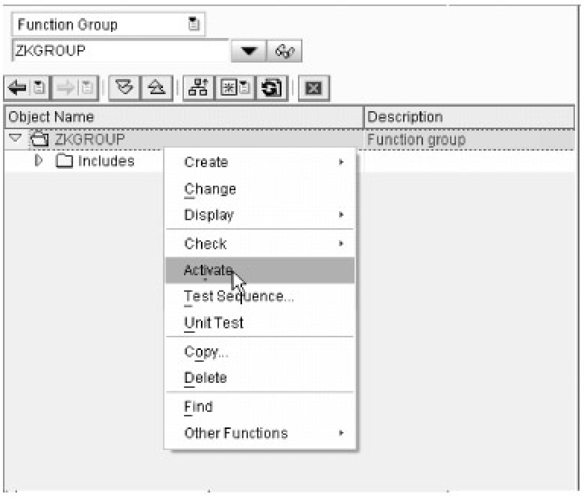


Function Group

ZKGROUP

Object Name	Description
ZKGROUP	Function group
Includes	
LZKGROUPTOP	
LZKGROUPUXX	LZKGROUPUXX

Figure 9.27: Viewing the function group



© SAP AG. All rights reserved.

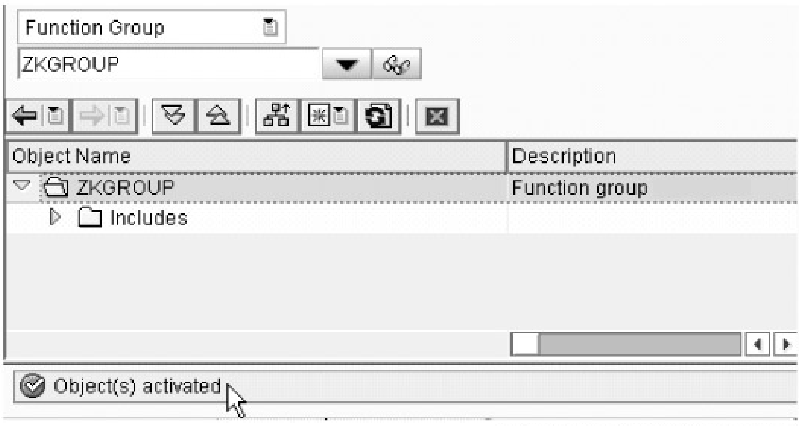
Figure 9.28: Activating the function group

The `Inactive Objects for KDT` dialog box appears.

9. Click the `Continue` (✓) icon. A message appears on the status bar, which states that the created function module is activated, as shown in Figure 9.29:

10. Start a new session by clicking the `Create New Session` (▣) icon on the standard toolbar. Enter the `SE37` transaction in the `Command` field. The initial screen of Function Builder appears, as shown in Figure 9.30:

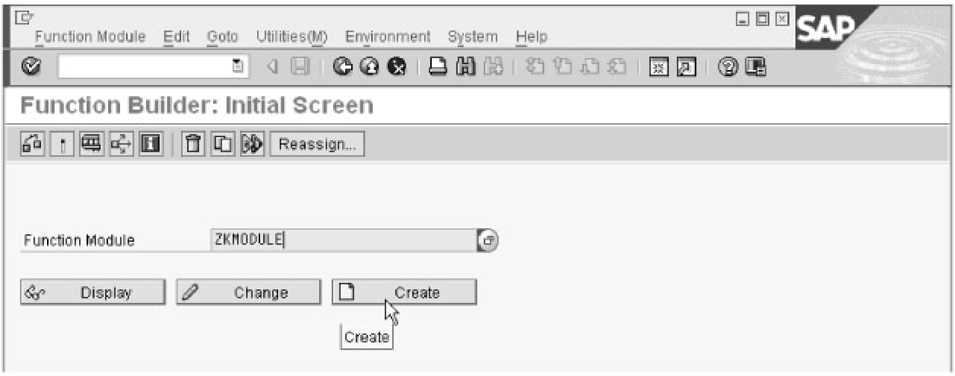
11. Enter the name of a function module in the `Function Module` field. In this case, we enter the name of the function module as ZKMODULE and click the `Create` button.



Object(s) activated

© SAP AG. All rights reserved.

Figure 9.29: Function group activated


Function Module Edit Goto Utilities(M) Environment System Help
SAP
Reassign...
Display
Change
Create
Create

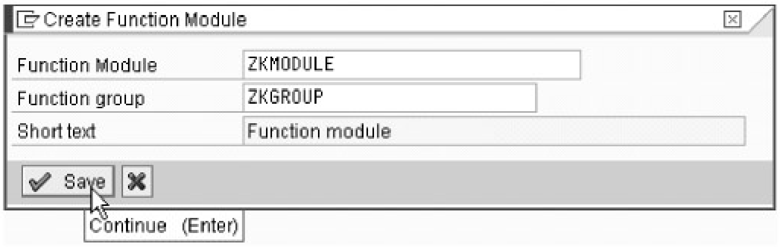
Function Builder: Initial Screen

Function Module ZKMODULE

© SAP AG. All rights reserved.

Figure 9.30: The initial screen of function builder

The `Create Function Module` dialog box appears (Figure 9.31).



Create Function Module

Function Module ZKMODULE

Function group ZKGROUP

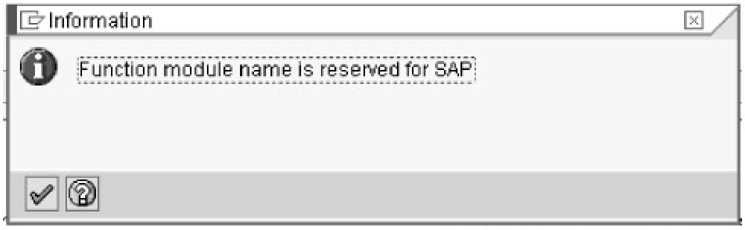
Short text Function module

© SAP AG. All rights reserved.

Figure 9.31: Displaying the name of the function group and short description

12. In the `Create Function Module` dialog box, enter the name of a function group in the `Function group` field. In addition, enter a description for the function module in the `Short text` field. In this case, the function group is ZKGROUP and the description is Function module. Finally, click the `Save` button, as shown in Figure 9.31:

The `Information` dialog box appears, as shown in Figure 9.32:



Information

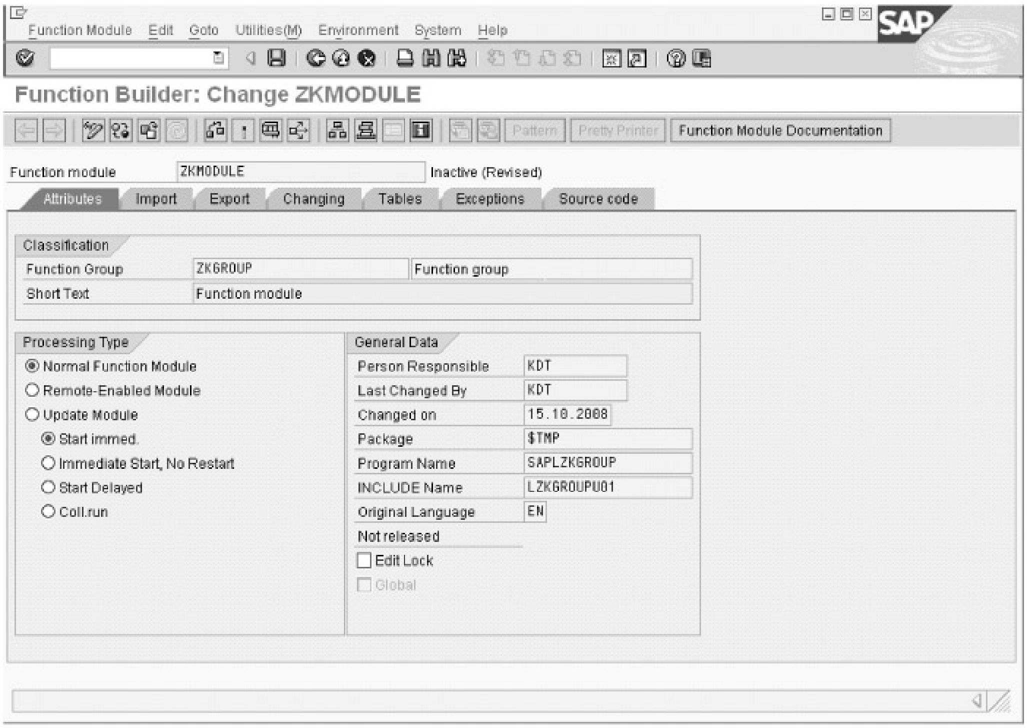
Function module name is reserved for SAP

© SAP AG. All rights reserved.

Figure 9.32: Displaying the information dialog box

13. Click the `Continue` (✓) icon:

The `Function Builder: Change ZKMODULE` screen appears, as shown in Figure 9.33:


Function Module Edit Goto Utilities(M) Environment System Help
Pattern
Pretty Printer
Function Module Documentation
Attributes
Import
Export
Changing
Tables
Exceptions
Source code

Function Builder: Change ZKMODULE

Function module ZKMODULE Inactive (Revised)

Classification

Function Group	ZKGROUP	Function group
Short Text	Function module	

Processing Type

● Normal Function Module
○ Remote-Enabled Module
○ Update Module
● Start immed.
○ Immediate Start, No Restart
○ Start Delayed
○ Coll.run

General Data

Person Responsible	KDT
Last Changed By	KDT
Changed on	15.10.2008
Package	$TMP
Program Name	SAPLZKGROUP
INCLUDE Name	LZKGROUPU01
Original Language	EN

Not released
□ Edit Lock
□ Global

© SAP AG. All rights reserved.

Figure 9.33: Displaying the function module

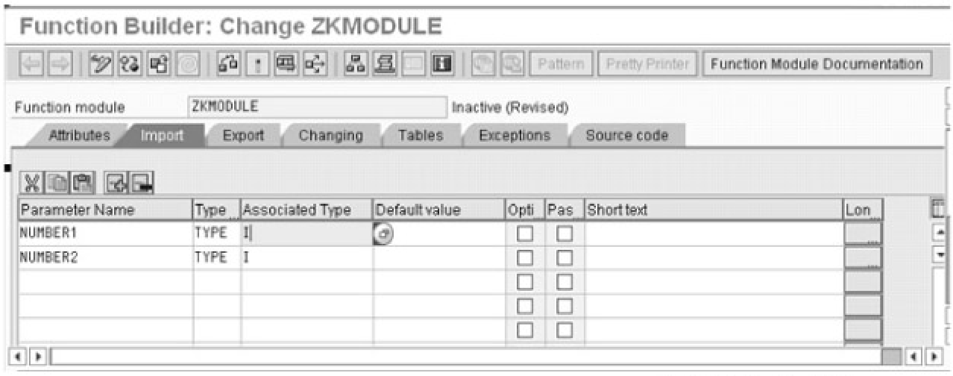
On this screen, you set the various subobjects, such as `Attributes`, `Import`, `Export`, `Changing`, `Tables`, `Exceptions`, and `the Source code`. These subobjects are known as interface parameters for the function module (see Figure 9.33).

14. Click the `Attributes` tab. The description of the function group, which was assigned earlier, that is, `Function module for kogent`, appears automatically in the `Short Text` field. You can also change the description and can provide a new description to the function module created. The processing type of the function module is selected as `Normal Function Module`. To call a function module from a program running in another SAP system, select the `Remote-Enabled Module` radio button.

    Note The `Remote-Enabled Module` radio button is selected when the function module that you have to use in your program exists on a different system. In an SAP system, such function modules are invoked by using the Remote Function Call (RFC) interface system. The RFC interface system enables a function call between SAP systems or between an SAP system and an external system (a non-SAP system).

15. Click the `Import` tab, where you set the variables that can be transferred from the calling program to the function module. In this case, we define two import parameters, as `NUMBER1` and `NUMBER2`, with `Type` as TYPE and `Associated Type` as I (refers to the integer type). You can also define some other associated types, such as predefined data element, any predefined structure, or any predefined database tables.

Note To get the list of the associated types, click the `Search Help` icon, which appears when you place the cursor in the `Associated` type field (see Figure 9.34).



Function module ZKMODULE Inactive (Revised)

Parameter Name	Type	Associated Type	Default value	Opti	Pas	Short text	Lon
NUMBER1	TYPE	I		□	□		
NUMBER2	TYPE	I		□	□		
				□	□		
				□	□		
				□	□		

Figure 9.34: Displaying the parameters on the import tab

Figure 9.34 shows the screen displaying the different parameters related to the `Import` tab:

16. Click the `Export` tab, where you set the variables whose values are transferred from the function module to the calling program. In this case, we define a parameter named `OPERATION` with the `Type` specification as TYPE and the `Associated Type` as I (refers to integer type). Figure 9.35 shows the different parameters related to the `Export` tab:

    Note You can also define some other associated types, such as predefined data element, a predefined structure, or a predefined database table by clicking the `search help` icon.

17. Click the `Changing` tab, where you define the variables that act as import and export parameters simultaneously.

18. Click the `Tables` tab to define the internal tables that can be either exported or imported. The content of the internal table is shifted from the calling program to the function module. In this function module, we do not use the `TABLES` subobject.

19. Click the `Exceptions` tab, where different parameters are defined. These parameters are used to determine any kind of error occurred in the function module.

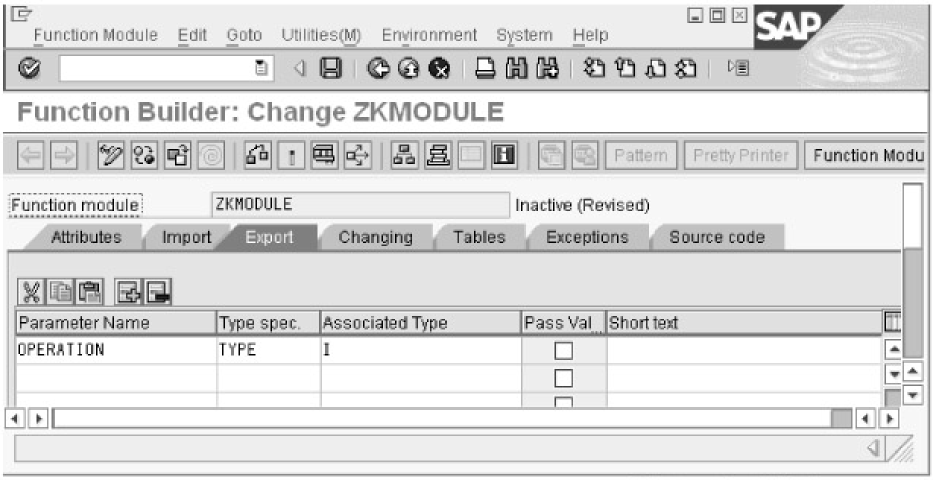
20. Click the `Source code` tab, where the program for the function module is actually written, as shown in Figure 9.36:

21. Click the `Check` icon to find whether there is any error in the function module, and then click the `Activate` icon to activate the source code.

22. Click the `Direct processing` icon to test whether the function module is working properly. The initial screen of Test Function Module appears, as shown in Figure 9.37:

23. Enter the values for the import parameters, `NUMBER1` and `NUMBER2`. In this case, we enter 10 as the value for `NUMBER1` and 20 as the value for `NUMBER2`. Now, according to the condition given in the source code (see Figure 9.36), if `NUMBER1` = 10, 30 is assigned as the value of the `OPERATION` export parameter.

24. Click the `Execute` icon to execute the created function module. Figure 9.38 shows the result screen of Test Function Module, displaying the value of the export parameter `OPERATION`:



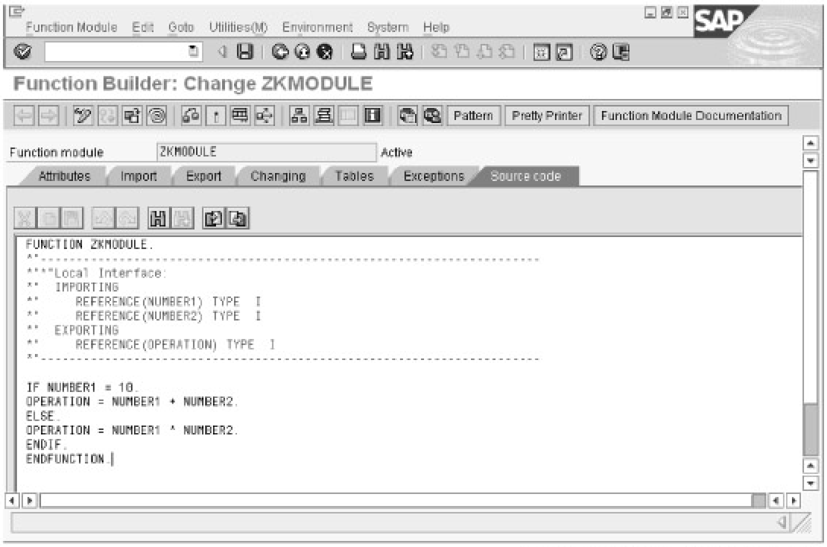
Function Builder: Change ZKMODULE

Function module ZKMODULE Inactive (Revised)

Parameter Name	Type spec.	Associated Type	Pass Val.	Short text
OPERATION	TYPE	I	☐	
			☐	

© SAP AG. All rights reserved.

Figure 9.35: Displaying the parameters on the export tab


Function Module Edit Goto Utilities(M) Environment System Help
Pattern
Pretty Printer
Function Module Documentation
Attributes
Import
Export
Changing
Tables
Exceptions
Source code

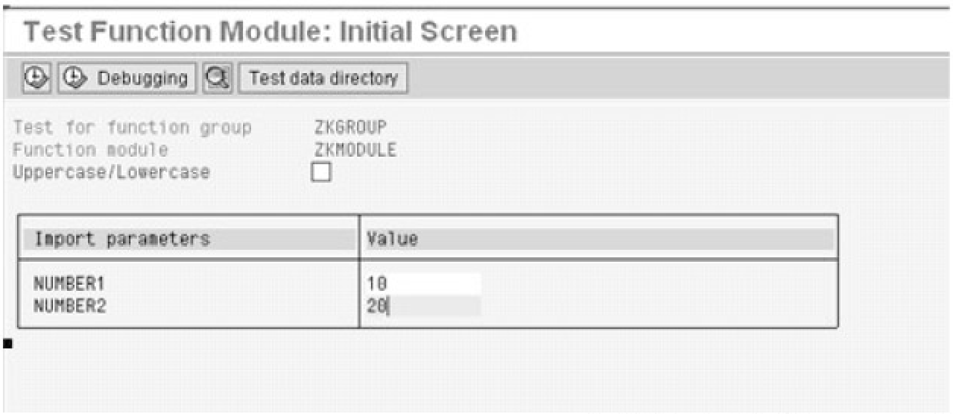
Function Builder: Change ZKMODULE

Function module ZKMODULE Active

```
FUNCTION ZKMODULE.
*"----------------------------------------------------------------------
*"*"Local Interface:
*"  IMPORTING
*"     REFERENCE(NUMBER1) TYPE  I
*"     REFERENCE(NUMBER2) TYPE  I
*"  EXPORTING
*"     REFERENCE(OPERATION) TYPE  I
*"----------------------------------------------------------------------

IF NUMBER1 = 10.
OPERATION = NUMBER1 + NUMBER2.
ELSE.
OPERATION = NUMBER1 * NUMBER2.
ENDIF.
ENDFUNCTION.
```

© SAP AG. All rights reserved.

Figure 9.36: Displaying the source code for the function module



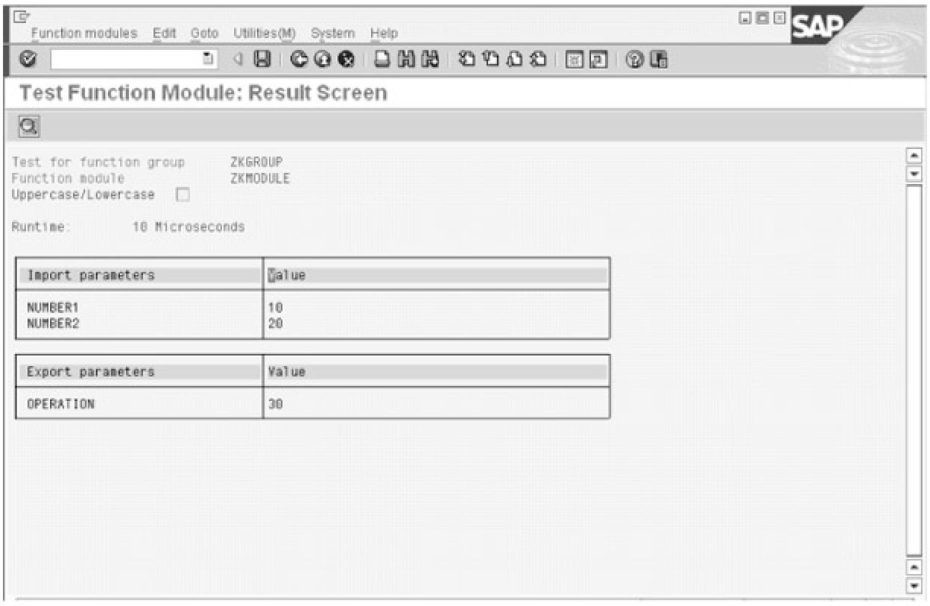
Test Function Module: Initial Screen

Test for function group ZKGROUP
Function module ZKMODULE
Uppercase/Lowercase ☐

Import parameters	Value
NUMBER1	10
NUMBER2	20

© SAP AG. All rights reserved.

Figure 9.37: Displaying the screen where values are entered



Test Function Module: Result Screen

```
Test for function group     ZKGROUP
Function module             ZKMODULE
Uppercase/Lowercase   □

Runtime:        10 Microseconds
```

Import parameters	Value
NUMBER1 NUMBER2	10 20

Export parameters	Value
OPERATION	30

© SAP AG. All rights reserved.

Figure 9.38: Showing the function module test screen

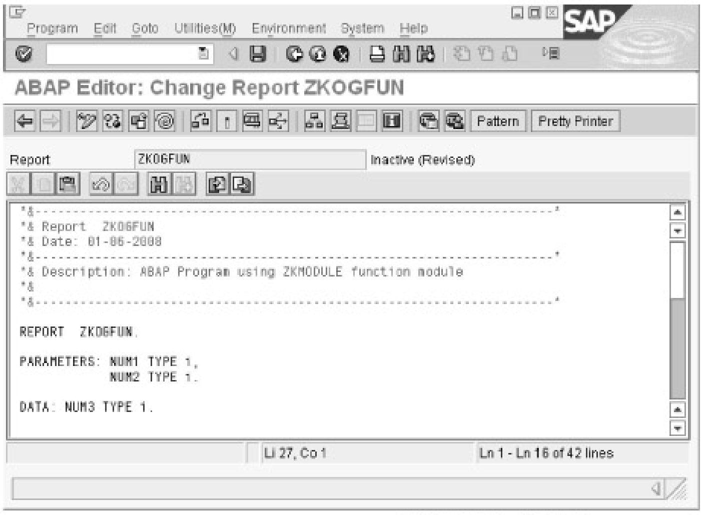
Now, let's learn how to use the created function module in this ABAP program example.

Calling Function Modules from ABAP Programs

The function modules created by using the Function Builder tool of ABAP Workbench can be called from ABAP programs. For this, you can call the generated function module by clicking the `Pattern` (Pattern) button on the application toolbar. Let's suppose that an executable type program, ZKOGFUN, is created in ABAP Editor. In this program, we define a function module named ZKMODULE. To call a function module, you need to create variables. In this case, we use two variables because the ZKMODULE function module consists of the two input variables, `NUMBER1` and `NUMBER2`, and an output variable, `OPERATION`. In this case, we use the `PARAMETERS` statement to assign values to variables. Values assigned to the variables are transferred from the function module to the calling program. A third variable, `NUM3`, is defined with the `DATA` statement so that the result from the function module can be passed to the calling program. The `NUM3` variable holds the value of the result. Perform the following steps to call a function module within an ABAP program:

1. Create an executable program type in ABAP Editor. In this case, we create a program named ZKOGFUN. Figure 9.39 shows the `ABAP Editor: Change Report ZKOGFUN` screen, containing the statements for this program:

2. Click the `Pattern` button on the application toolbar, as shown in Figure 9.40:



ABAP Editor: Change Report ZKOGFUN

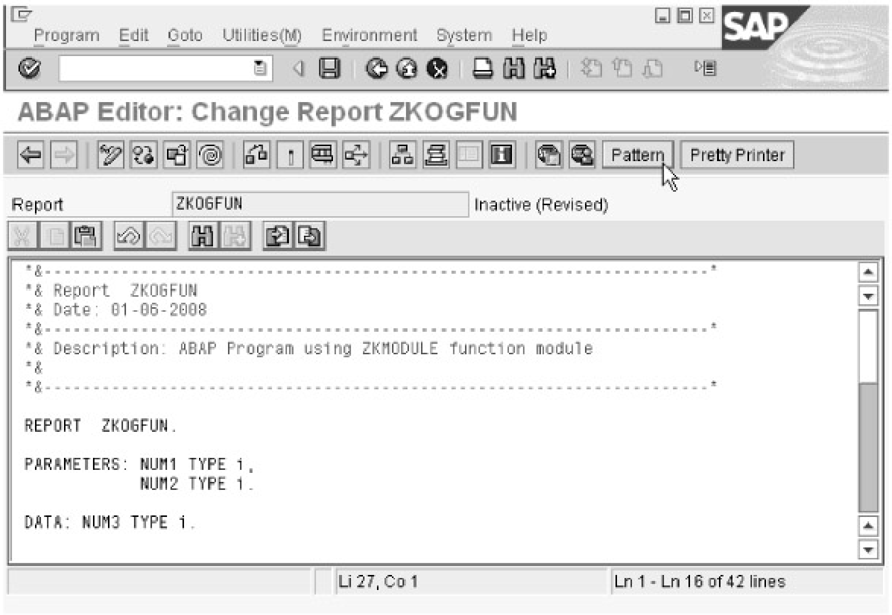
Report ZK06FUN Inactive (Revised)

```
*&---------------------------------------------------------------------*
*& Report  ZK06FUN
*& Date: 01-06-2008
*&---------------------------------------------------------------------*
*& Description: ABAP Program using ZKMODULE function module
*&
*&---------------------------------------------------------------------*

REPORT  ZK06FUN.

PARAMETERS: NUM1 TYPE i,
            NUM2 TYPE i.

DATA: NUM3 TYPE i.
```

Li 27, Co 1 Ln 1 - Ln 16 of 42 lines

© SAP AG. All rights reserved.

Figure 9.39: Displaying the ABAP editor screen



ABAP Editor: Change Report ZKOGFUN

Report ZKOGFUN Inactive (Revised)

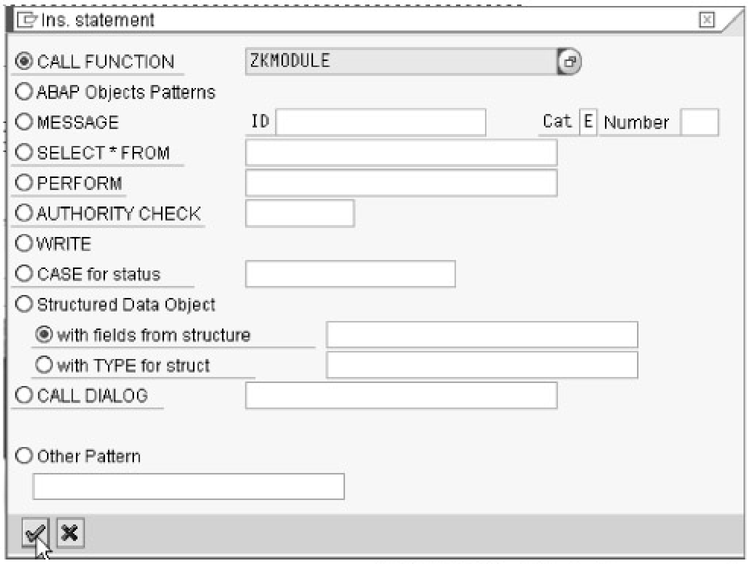
```
*&---------------------------------------------------------------------*
*& Report  ZKOGFUN
*& Date: 01-06-2008
*&---------------------------------------------------------------------*
*& Description: ABAP Program using ZKMODULE function module
*&
*&---------------------------------------------------------------------*

REPORT  ZKOGFUN.

PARAMETERS: NUM1 TYPE i,
            NUM2 TYPE i.

DATA: NUM3 TYPE i.
```

Li 27, Co 1 Ln 1 - Ln 16 of 42 lines

© SAP AG. All rights reserved.

Figure 9.40: Clicking the pattern button

The `Ins. Statement` dialog box appears, as shown in Figure 9.41.



Ins. statement

● CALL FUNCTION ZKMODULE
○ ABAP Objects Patterns
○ MESSAGE ID Cat E Number
○ SELECT * FROM
○ PERFORM
○ AUTHORITY CHECK
○ WRITE
○ CASE for status
○ Structured Data Object
● with fields from structure
○ with TYPE for struct
○ CALL DIALOG

○ Other Pattern

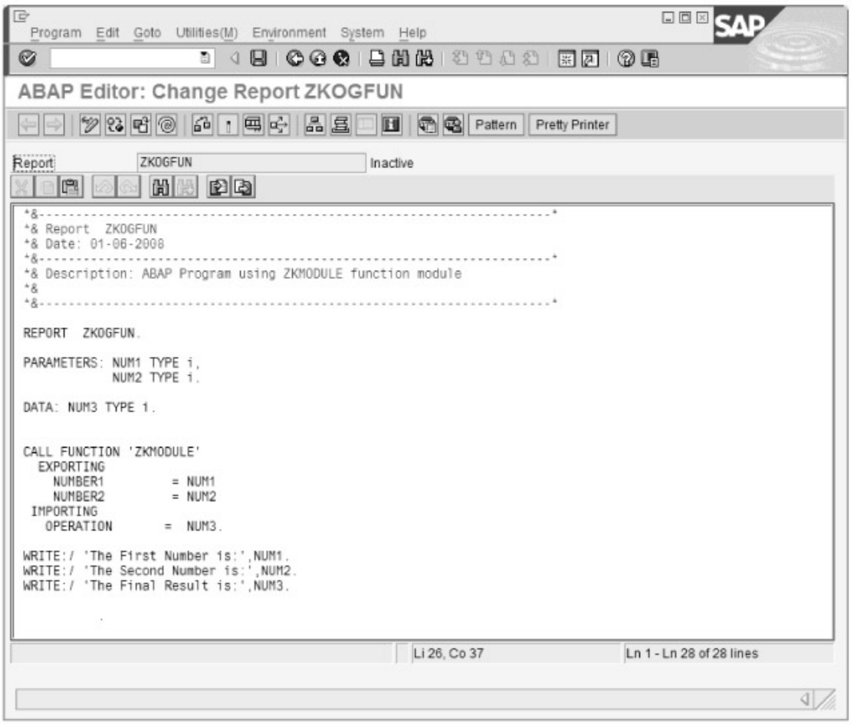
© SAP AG. All rights reserved.

Figure 9.41: Calling the function module

3. Enter the name of the function module to be included in the ABAP program. In this case, we enter ZKMODULE in the `CALL FUNCTION` field and click the `Continue` icon, as shown in Figure 9.41:

The function module ZKMODULE is included in the program ZKOGFUN.

4. Write the code for the ZKOGFUN program in the ABAP Editor, as shown in Figure 9.42:



ABAP Editor: Change Report ZKOGFUN

Report ZKOGFUN Inactive

```
*&---------------------------------------------------------------------*
*& Report  ZKOGFUN
*& Date: 01-06-2008
*&---------------------------------------------------------------------*
*& Description: ABAP Program using ZKMODULE function module
*&
*&---------------------------------------------------------------------*

REPORT  ZKOGFUN.

PARAMETERS: NUM1 TYPE i,
            NUM2 TYPE i.

DATA: NUM3 TYPE 1.


CALL FUNCTION 'ZKMODULE'
  EXPORTING
    NUMBER1          = NUM1
    NUMBER2          = NUM2
 IMPORTING
   OPERATION       =  NUM3.

WRITE:/ 'The First Number is:',NUM1.
WRITE:/ 'The Second Number is:',NUM2.
WRITE:/ 'The Final Result is:',NUM3.
```

Li 26, Co 37 Ln 1 - Ln 28 of 28 lines

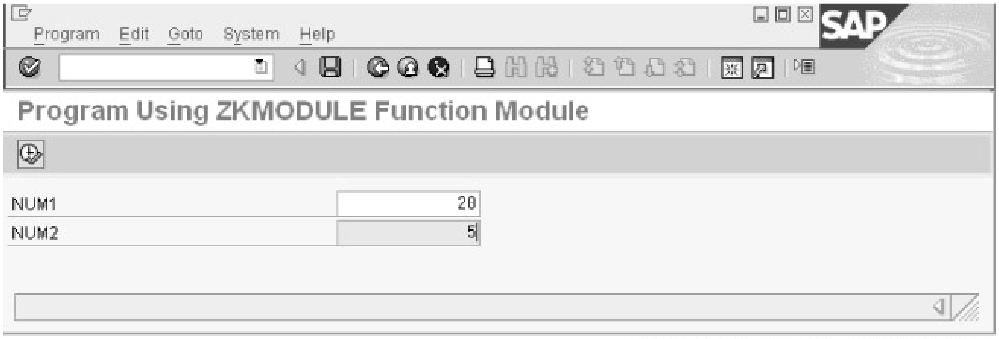
© SAP AG. All rights reserved.

Figure 9.42: Displaying the ABAP program using the function module

In Figure 9.42, you see in the ZKOGFUN program that `NUMBER1` and `NUMBER2` are `EXPORTING` parameters that are assigned with the values of the `NUM1` and `NUM2` variables. Now, assign the value of the `NUM3` variable to the `IMPORTING` parameter, `OPERATION`.

5. Finally, click the `Save` () icon to save the created program, the `Check` () icon to check for any errors, and the `Activate` () icon to activate the program.

Figure 9.43 shows the output of the program, where you need to enter the values for the variables:



Program Using ZKMODULE Function Module

NUM1 20

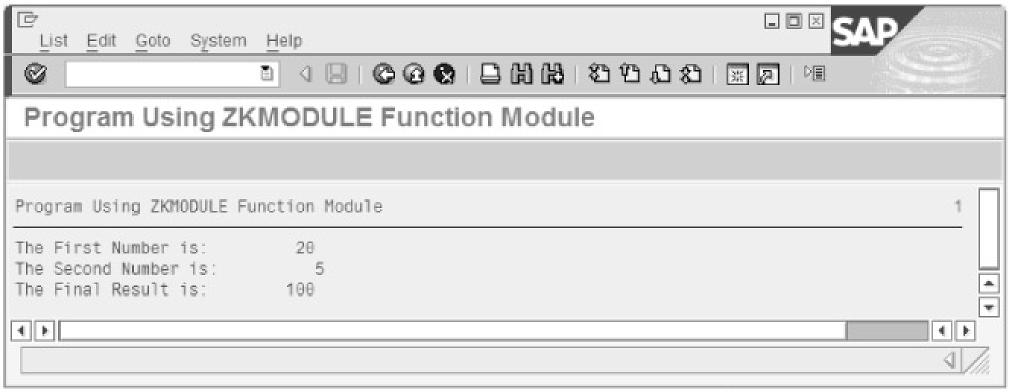
NUM2 5

© SAP AG. All rights reserved.

Figure 9.43: Showing the values assigned to the variables

6. Enter the values for the variables `NUMBER1` and `NUMBER2`. In our case, we enter the values as 20 and 5, respectively, as shown in Figure 9.43.

7. Click the `Execute` () icon; the output corresponding to the values entered in the `NUM1` and `NUM2` fields appears, as shown in Figure 9.44:



Program Using ZKMODULE Function Module

```
Program Using ZKMODULE Function Module                                         1

The First Number is:         20
The Second Number is:         5
The Final Result is:        100
```

© SAP AG. All rights reserved.

Figure 9.44: Output screen

Source Code Modules

Source code modules are used to modularize your source code. Modularizing a source code means placing a sequence of ABAP statements in a module. The modularized source code can be called in a program depending on the user's requirements. Source code modules enhance the readability and understandability of ABAP programs. They also help to minimize data redundancy by avoiding repeated occurrence of the same set of statements. It must be noted that source code modules do not modularize tasks and functions.

ABAP provides the following two types of source code modules:

- Macros
- Include programs

Macros

A macro is a set of statements that can be used more than once in the ABAP program. For example, a macro can be useful for long calculations or for writing complex `WRITE` statements. A macro can be used only in that program in which it is created. You can define a macro within the `DEFINE...END-OF-DEFINITION` statement. The following syntax defines a macro:

```
DEFINE macro.
   statements
END-OF-DEFINITION.
```

A macro must be defined prior to its call in the ABAP program. Execute the following statement to use a macro:

```
macro [prm1 prm2... prm9].
```

When a program is executed, the SAP system replaces the macro by appropriate statements and the placeholders `&1`, `&2`, `...`, `&9` by the parameters `prm1`, `prm2`, `...`, `prm9`.

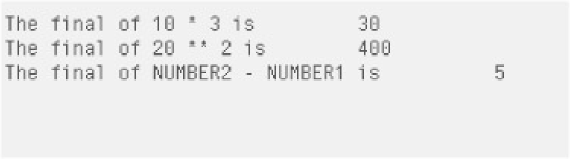
A macro can be used within macros; however, a macro cannot call itself. Listing 9.14 shows an example of a nested macro:

Listing 9.14: Nesting of a macro

```
REPORT ZMYMACRO.
*--------------------------------------------------*
*/ Defining Variables
*/
DATA: Finally TYPE i,
               number1      TYPE i VALUE 5,
               number2      TYPE i VALUE 10.
*--------------------------------------------------*

DEFINE operation.

               finally = &1 &2 &3.
               output &1 &2 &3 finally.
```

```
END-OF-DEFINITION.

DEFINE output.
      write: / 'The final of &1 &2 &3 is', &4.
END-OF-DEFINITION.
operation 10 * 3.
operation 20 ** 2.
  operation number2 - number1.
```

In Listing 9.14, two macros, `operation` and `output`, are defined. The `output` macro is nested within the `operation` macro. The `operation` macro is executed three times with different parameters. When the program is executed, the SAP system replaces the `operation` macro with the statements defined inside the macro and replaces each placeholder, `&1`, `&2`, and `&3`, by the parameters defined in the macro. Figure 9.45 shows the output of Listing 9.14:



```
The final of 10 * 3 is         30
The final of 20 ** 2 is        400
The final of NUMBER2 - NUMBER1 is         5
```

© SAP AG. All rights reserved.

Figure 9.45: Use a macro

Include Programs

Include programs are global repository objects used to modularize the source code. They allow you to use the same source code in different programs. Include programs also allow you to manage complex programs in an orderly way. To use an include program in another program, use the following syntax of the `INCLUDE` statement:

```
INCLUDE <incl>.
```

The `INCLUDE` statement has the same effect as copying the source code of the include program `<incl>` into another program. An include program cannot run independently; it must be built into other programs. You can nest include programs.

The following restrictions must be considered while writing the source code for include programs:

- Include programs cannot call themselves
- Include programs must contain complete statements

Perform the following steps to create and use an include program:

1. Create the program to be included in ABAP Editor. Remember to set the `Type` of the program to `INCLUDE program`, as shown in Figure 9.46:
2. Click the `Save` (✓ Save) button and save the program in a package named ZKOG_PCKG. Listing 9.15 shows the content of the program to be included in ABAP Editor:

   Listing 9.15: Program to be included

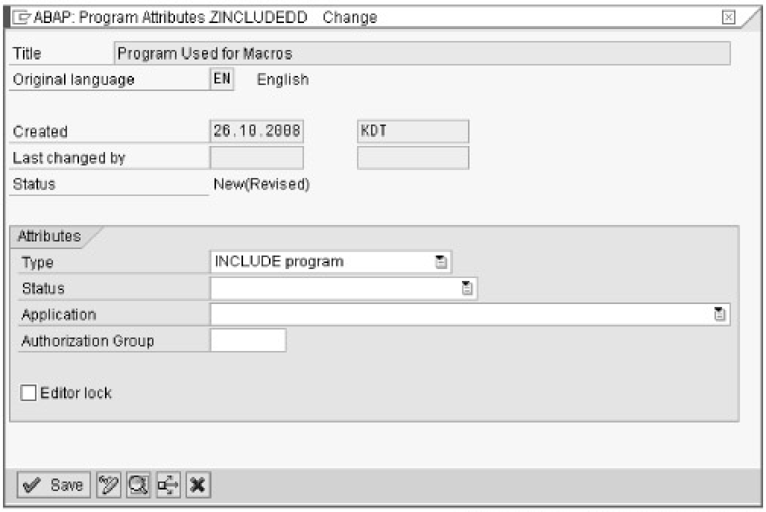
   ```
   PROGRAM ZINCLUDED.
   WRITE:     / 'Program started by::', SY-UNAME,
               / 'On host:', SY-HOST,
               / 'Date:', SY-DATUM,
               / 'Time:' SY-UZEIT.
   ```

3. Create another program where the program ZINCLUDED has to be used. In this case, we have created another program named ZINCLUDING and assigned a type for the program, such as `Executable program`. Now, the coding for the ZINCLUDING program includes the ZINCLUDED program with the help of the `INCLUDE` statement, as

shown in Listing 9.16:

Listing 9.16: Coding for the program ZINCLUDING

```
REPORT ZINCLUDING.
INCLUDE ZINCLUDED.
```



ABAP: Program Attributes ZINCLUDEDD Change

Title: Program Used for Macros
Original language: EN English

Created: 26.10.2008 KDT
Last changed by:
Status: New(Revised)

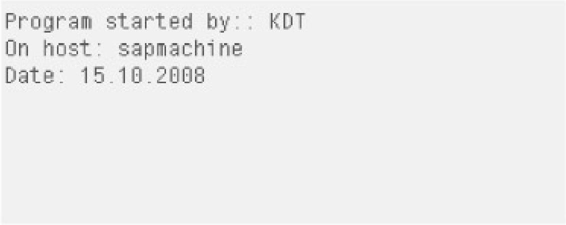
Attributes
Type: INCLUDE program
Status:
Application:
Authorization Group:

□ Editor lock

© SAP AG. All rights reserved.

Figure 9.46: Defining the attributes

In Listing 9.16, notice that the program named ZINCLUDED is called with the help of the `Include` statement. Figure 9.47 shows the output of Listing 9.16:



```
Program started by:: KDT
On host: sapmachine
Date: 15.10.2008
```

© SAP AG. All rights reserved.

Figure 9.47: Displaying the output of an INCLUDE program

Summary

In this chapter, you have learned about the different types of modularization techniques, including subroutines, which acts like a mini-program consisting of sequence of statements, function modules, and source code modules. Function modules encapsulate program code, and source code modules are used to modularize source code.