

# BC Extended Applications Function Library



**Release 4.6C**



## Copyright

© Copyright 2001 SAP AG. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP AG. The information contained herein may be changed without prior notice.

Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors.

Microsoft®, WINDOWS®, NT®, EXCEL®, Word®, PowerPoint® and SQL Server® are registered trademarks of Microsoft Corporation.

IBM®, DB2®, OS/2®, DB2/6000®, Parallel Sysplex®, MVS/ESA®, RS/6000®, AIX®, S/390®, AS/400®, OS/390®, and OS/400® are registered trademarks of IBM Corporation.

ORACLE® is a registered trademark of ORACLE Corporation.

INFORMIX®-OnLine for SAP and Informix® Dynamic Server™ are registered trademarks of Informix Software Incorporated.

UNIX®, X/Open®, OSF/1®, and Motif® are registered trademarks of the Open Group.






HTML, DHTML, XML, XHTML are trademarks or registered trademarks of W3C®, World Wide Web Consortium, Massachusetts Institute of Technology.

JAVA® is a registered trademark of Sun Microsystems, Inc.

JAVASCRIPT® is a registered trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.

SAP, SAP Logo, R/2, RIVA, R/3, ABAP, SAP ArchiveLink, SAP Business Workflow, WebFlow, SAP EarlyWatch, BAPI, SAPPHIRE, Management Cockpit, mySAP.com Logo and mySAP.com are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world. All other products mentioned are trademarks or registered trademarks of their respective companies.

## Icons

Icon	Meaning
	Caution
	Example
	Note
	Recommendation
	Syntax

## Contents

<b>BC Extended Applications Function Library.....</b>	<b>7</b>
<b>Standardized dialogs .....</b>	<b>8</b>
<b>Overview.....</b>	<b>9</b>
<b>Concept .....</b>	<b>10</b>
<b>Procedure.....</b>	<b>11</b>
Determine dialog type .....	12
Create dialog text Document.....	13
<b>Confirmation prompt dialogs .....</b>	<b>14</b>
<b>Dialogs for choosing from among alternatives.....</b>	<b>15</b>
<b>Dialogs for displaying, inputting and checking data.....</b>	<b>16</b>
<b>Data print dialogs .....</b>	<b>18</b>
<b>Text display dialogs .....</b>	<b>19</b>
<b>Scrolling in tabular structures .....</b>	<b>20</b>
<b>Extended table maintenance.....</b>	<b>21</b>
<b>Overview.....</b>	<b>22</b>
<b>Concept .....</b>	<b>23</b>
<b>Create maintenance dialog.....</b>	<b>25</b>
<b>Calling the standard maintenance dialog .....</b>	<b>27</b>
<b>Call via function module.....</b>	<b>28</b>
<b>Highest level entry .....</b>	<b>29</b>
<b>Middle level entry .....</b>	<b>30</b>
<b>Lowest level entry .....</b>	<b>33</b>
<b>Central address management.....</b>	<b>37</b>
<b>Overview.....</b>	<b>38</b>
<b>Calendar .....</b>	<b>39</b>
<b>Overview.....</b>	<b>40</b>
<b>Concept .....</b>	<b>41</b>
<b>Determine calendar ID .....</b>	<b>42</b>
<b>Calendar functions.....</b>	<b>43</b>
<b>Measurement units.....</b>	<b>44</b>
<b>Overview.....</b>	<b>45</b>
<b>Concept .....</b>	<b>46</b>
<b>Check measurement unit table .....</b>	<b>49</b>
<b>Measurement unit conversion .....</b>	<b>50</b>
<b>s Conversion of measurement units and possible entries (F4) help.....</b>	<b>52</b>
<b>Change documents .....</b>	<b>54</b>
<b>Overview.....</b>	<b>55</b>
<b>Concept .....</b>	<b>56</b>
<b>Procedure.....</b>	<b>58</b>
Define change document object .....	59
Set change document flag .....	61
Generate Update and INCLUDE Objects .....	62

Integrating the functionality into the program.....	64
Writing the fields in the program.....	65
<b>Creating change documents.....</b>	<b>69</b>
<b>Read and format change documents.....</b>	<b>70</b>
<b>Read and format planned changes.....</b>	<b>71</b>
<b>Delete change documents and planned changes.....</b>	<b>72</b>
<b>Archived change documents management.....</b>	<b>73</b>
<b>Create application log.....</b>	<b>74</b>
Overview.....	75
Concept.....	76
Procedure.....	77
Define application log objects.....	78
Create application log.....	79
Display application log.....	80
Read application log.....	81
Delete application log.....	82
<b>Platform-independent File Name Assignment.....</b>	<b>83</b>
Overview.....	84
Definitions of Platform-independent File Names.....	85
The Function Module FILE_GET_NAME.....	88
Using Platform-independent File Names in Programs.....	91
Reference.....	92
<b>Number ranges.....</b>	<b>93</b>
Overview.....	94
Concept.....	95
Number range object types.....	97
Procedure.....	100
Determine the number range object type.....	101
Maintain number range object.....	102
Function module calls.....	105
Number range and group maintenance dialogs.....	107
Number range and group read and maintain services.....	109
Number range object read and maintain services.....	111
Number assignment and check.....	112
Utilities.....	113
<b>Data Archiving - ADK.....</b>	<b>114</b>
Overview.....	115
ADK: Development Environment for Archiving Programs.....	116
Interaction between Program, ADK, and Archive File.....	117
Archiving using ADK.....	119
Archiving Process.....	121
Archiving Objects.....	122
Standard Class.....	124
Archiving Classes.....	125
Archive Administration.....	127
Network Graphic.....	129

<b>Developing Archiving Programs .....</b>	<b>130</b>
Defining Archiving Objects .....	131
Defining Standard Class Hierarchical Structure.....	133
Tables From Which You Only Delete Entries .....	134
Archiving Object-Specific Customizing .....	135
Assigning Archiving Classes .....	138
<b>Developing Archiving Programs .....</b>	<b>139</b>
Standard Archiving Program .....	142
Archiving Using Archiving Classes .....	143
<b>Developing Delete Programs .....</b>	<b>145</b>
<b>Developing Reload Programs .....</b>	<b>147</b>
<b>Developing Analysis Programs .....</b>	<b>149</b>
<b>Maintaining Network Graphic.....</b>	<b>151</b>
<b>Creating ADK Indexes and Using Them to Access Archive .....</b>	<b>152</b>
<b>Archiving Classes .....</b>	<b>154</b>
<b>Developing Archiving Classes.....</b>	<b>156</b>
Specifying Function Groups .....	157
Developing Function Modules.....	158
Developing Subprograms.....	161
Initializing the Archiving Classes for Writing .....	163
Getting Data .....	165
Deleting Local Memory of Archiving Class .....	166
Declaring an Archive Handle Invalid.....	167
Initializing Archiving Classes for Reading.....	168
Copying Data From the Data Container .....	170
Deleting Archived Data .....	171
Discarding the Data Selected for Deletion.....	172
Reloading Archived Data .....	173
Defining Archiving Classes .....	174
<b>Archiving Functions.....</b>	<b>176</b>

## BC Extended Applications Function Library

---

**Standardized dialogs**

## Standardized dialogs

This section explains how you can standardize the dialogs in your developments.

[Overview \[Page 9\]](#)

[Concept \[Page 10\]](#)

### Procedures

[Procedure \[Page 11\]](#)

[Determine dialog type \[Page 12\]](#)

[Create dialog text Document \[Page 13\]](#)

### References

[Confirmation prompt dialogs \[Page 14\]](#)

[Dialogs for choosing from among alternatives \[Page 15\]](#)

[Dialogs for displaying, inputting and checking data \[Page 16\]](#)

[Data print dialogs \[Page 18\]](#)

[Text display dialogs \[Page 19\]](#)

[Scrolling in tabular structures \[Page 20\]](#)



## Overview

Some dialog steps and functions are required frequently during the realization of application development dialogs. These are available as function modules in self-contained modules. Their use standardizes application dialogs, which in turn simplifies use.

---

**Concept****Concept**

The function modules provide a standardized dialog box with function keys which are tested at the end of the dialog. Depending on the function module, texts for information, for choices and/or for the available function keys can be passed.

## Procedure

No preparatory steps are necessary for the use of the function modules for standardized dialog boxes, with the exception of the text display function modules (function group SP06). In this case, the texts must be created previously.

To use standardized dialog boxes, perform the following steps:

1. [Determine dialog type \[Page 12\]](#)

If you want to use a function module from the function group SPO6: [Create dialog text Document \[Page 13\]](#)

2. Choose the appropriate function module in the function group found in step 1.
3. Call this function module in the application.

**Determine dialog type**

## Determine dialog type

Determine what information you want to provide the user, and the decision logic you require. Then choose the appropriate function group from the following table.

Situation	Function group
The user is to be warned of potential data loss	SPO1
The user should answer a question about further processing with Yes or No	SPO1
The user is to be warned about potential data loss, and decide whether he or she wants to continue with the action	SPO1
The user must choose between further processing alternatives	SPO2
The user must continue the current action or cancel	SPO2
The user is to input data (with or without check against a value table)	SPO4
Data are to be displayed to the user	SPO4
The user is to receive detailed information	SPO6
The user is to be able to scroll in a displayed list	STAB
The user is to print data from a table or a table view	STRP

## Create dialog text Document

To create a "Dialog text" (for function modules in the function group SP06), proceed as follows:

1. In the initial screen choose the function *Tools* → *Abap/4 Workbench* → *Environment* → *Documentation*.
2. Position the cursor on the document class output field and press F4.
3. Choose the class *Dialog text*.
4. Enter a document name and choose *Create*.
5. Enter the text and save it.  
Saving via the icon creates a raw document. Raw versions can not be transported or translated. The document must be a final version for these actions to be possible. You achieve this with the function *Save final version*. You must be authorized to save final versions of documents in this development class.

## Confirmation prompt dialogs

## Confirmation prompt dialogs

### Function group SPO1

This function group contains the following function modules:

- **POPUP\_TO\_CONFIRM\_STEP**  
With this function module you create a dialog box in which you ask the user during an action, whether he or she wishes to perform the step. You pass the title and a two-line question. The user must choose *Yes*, *No* or *Cancel*.  
The possible responses are provided by the function module.  
In the interface, the response "Yes" is pre-selected, but "No" can also be pre-selected via a parameter.  
The user response (*Yes*, *No* or *Cancel*) is returned in a parameter.
- **POPUP\_TO\_CONFIRM\_WITH\_MESSAGE**  
With this function module you create a dialog box in which you inform the user about a specific decision point during an action. You pass a title, a three-line diagnosis text and a two-line question, which he or she must answer with *Yes*, *No* or *Cancel*.  
The possible responses are provided by the function module. In the interface the response "Yes" is pre-selected, but "No" can also be pre-selected via a parameter.  
The user response (*Yes*, *No* or *Cancel*) is returned in a parameter.
- **POPUP\_TO\_CONFIRM\_WITH\_VALUE**  
With this function module you create a dialog box in which you ask the user, during an action, whether he or she wishes to perform a processing step with a particular object. You pass a title, a two-line decision question and the object, which is inserted between the two parts of the question. The user must choose *Yes*, *No* or *Cancel*.  
The possible responses are provided by the function module.  
In the interface the response "Yes" is pre-selected, but "No" can also be pre-selected via a parameter.  
The user response (*Yes*, *No* or *Cancel*) is returned in a parameter.
- **POPUP\_TO\_CONFIRM\_LOSS\_OF\_DATA**  
With this function module you create a dialog box in which you ask the user, during an action, whether he or she wishes to perform a processing step with loss of data. You pass the title and the two-line question. The warning that data are lost and the possible responses are provided by the function module. The user must answer "Yes", "No" or "Cancel".  
  
In the interface the response "No" is pre-selected and can not be changed.  
The user response (*Yes*, *No* or *Cancel*) is returned in a parameter.

## Dialogs for choosing from among alternatives

### Function group SPO2

This function group contains the following function modules:

- **POPUP\_TO\_DECIDE**  
With this function module you create a dialog box in which you require the user to choose between the two further processing alternatives offered, or to cancel the action.  
The action, the question and the alternative actions are passed as parameters. In the interface the alternative 1 is pre-selected, but alternative 2 can also be pre-selected via a parameter.  
The user action (Alternative 1, Alternative 2, or Cancel) is returned in a parameter.
- **POPUP\_TO\_DECIDE\_WITH\_MESSAGE**  
With this function module you create a dialog box in which you inform the user about a specific decision point via a diagnosis text, during an action. He or she can choose one of two alternative actions offered or cancel the action.  
The action, the diagnosis text, the question and the alternative actions are passed as parameters.  
The user action (Alternative 1, Alternative 2, or Cancel) is returned in a parameter.

## Dialogs for displaying, inputting and checking data

### Function group SPO4

This function group contains the following function modules:

- **POPUP\_GET\_VALUES**  
This function module sends a dialog box for data display and input.  
The input fields are passed in a structure and must be defined in the Dictionary. You can also specify individual field display attributes and a field text, if the key word from the Dictionary is not to be displayed as field text in the dialog box, in the structure.  
The standard help functionality (F1, F4) is supported.
- **POPUP\_GET\_VALUES\_DB\_CHECKED**  
This function module sends a dialog box for data to be input und checked against the database.  
The input fields are passed in a structure and must be defined in the Dictionary. You can also specify individual field display attributes and a field text in the structure, if the key word from the Dictionary is not to be displayed as field text in the dialog box.  
A comparison operator for checking the input data in the database is passed. You can specify whether the check is for the existence or absence of an object. A foreign key relationship check is supported.  
The standard help functionality (F1, F4) is supported.  
The user action is returned in a parameter.
- **POPUP\_GET\_VALUES\_USER\_CHECKED**  
This function module sends a dialog box for data to be input and checked in an external sub-routine (user exit). The input fields are passed in a structure and must be defined in the dictionary. You can also specify individual field display attributes and a field text in the structure, if the key word from the Dictionary is not to be displayed as field text in the dialog box.  
The Data input by the user in the dialog box are passed to the sub-routine specified in the interface for checking. Errors found by the check are entered in an error structure and are evaluated on return from the sub-routine by the function module.  
The standard help functionality (F1, F4) is supported.  
The user action (*Continue* or *Cancel*) is returned in a parameter.
- **POPUP\_GET\_VALUES\_USER\_HELP**  
This function module sends a dialog box for data to be input with the possibility of a check in an external sub-routine (user exit) and branching in a user F1 or F4 help.  
The input fields are passed in a structure and must be defined in the Dictionary. You can also specify individual field display attributes and a field text in the structure, if the key word from the Dictionary is not to be displayed as field text in the dialog box.  
You can pass the data which are entered by the user in a dialog box to a sub-routine which must be specified in the interface for checking. Errors occurring in the check are stored in an error structure and are analyzed by the function module upon return from the sub-routine.  
The data, and an error message, if appropriate, are displayed again.  
The standard help functionality (F1, F4) is supported.  
User exits for a user F1 or F4 help can also be specified.  
The user action (*Continue* or *Cancel*) is returned in a parameter.
- **POPUP\_GET\_VALUES\_USER\_BUTTONS**  
This function module is like the previous function module



---

**Dialogs for displaying, inputting and checking data**

POPUP\_GET\_VALUES\_USER\_HELP, with the additional possibility of passing one or two additional pushbuttons and a standard pushbutton, which the user can name.

- **POPUP\_GET\_VALUES\_SET\_MAX\_FIELD**  
With this function module you can specify the maximum number of fields which can be displayed in dialog boxes for this function group (SPO4). The specified value is stored in the function group local memory and applies for the rest of the application. Dialog boxes which display more than this number of fields are displayed with a scroll bar.

---

Data print dialogs

## Data print dialogs

### Function group STRP

With the two function modules in this function group you can print database table or view records. With a parameter you can control whether the table records are output with a standard list format or whether the user can specify the print format in a dialog box. The user can choose fields, specify a sort sequence and specify the column sequence and titles in these dialog boxes. Standard output (without dialog box) error cases are caught by exceptions. Output with user-defined format generates error messages in case of error.

- **TABLE\_PRINT\_STRUCTURE\_KNOWN**  
You pass data from tables whose structure is known in the program to this function module.
- **TABLE\_PRINT\_STRUCTURE\_UNKNOWN**  
You pass data from tables whose structure is not known in the program to this function module. These structure data are fetched independently by the function module.

## Text display dialogs

### Function group SPO6

With this function module you can display pre-prepared texts which exist in the system. These texts must have been created as documents of the class "Dialog text " with the documentation maintenance transaction (*Tools → ABAP/4 Workbench → Environment → Documentation*).

- **POPUP\_DISPLAY\_TEXT**  
With this function module you display a text which exists in the system in a dialog box.
- **POPUP\_DISPLAY\_TEXT\_WITH\_PARAMS**  
With this function module you display a text which exists in the system with parameters in a dialog box. The parameter values are passed in a table. The use of numbered texts is recommended, to make the parameter values translatable.  
The parameter names must be passed in upper-case letters.

---

**Scrolling in tabular structures**

## Scrolling in tabular structures

### Function group STAB

- **SCROLLING\_IN\_TABLE**

With this function module you enable the user to scroll in a list which you have created, e.g. as a logical part of an internal table. You can enable either page-wise scrolling or positioning on individual records.

## Extended table maintenance

This section explains how you can use the extended table maintenance standard maintenance dialog in your developments.

[Overview \[Page 22\]](#)

[Concept \[Page 23\]](#)

### Procedure

[Create maintenance dialog \[Page 25\]](#)

[Calling the standard maintenance dialog \[Page 27\]](#)

### References

[Call via function module \[Page 28\]](#)

[Highest level entry \[Page 29\]](#)

[Middle level entry \[Page 30\]](#)

[Lowest level entry \[Page 33\]](#)

---

**Overview**

## Overview

The extended table maintenance maintenance dialog provides the possibility of processing table data in a consistent maintenance dialog, independently of whether access is to be made directly via the table or via a view defined in the dictionary.

Integrating the maintenance dialog into user developments offers simplified access to table contents, and has the following advantages, among others:

- **The programming effort is considerably reduced.**
- **The operation is consistent, comprehensible and convenient.**

The convenience is produced by comprehensive operating functionality, which guarantees the comprehensibility and transparency of the maintenance procedures.

The maintenance dialog also allows table data to be viewed on two levels. You can branch from an overview screen to a detail screen for a selected record.

The maintenance dialog can be used in its entirety as a standard maintenance dialog, or table or view-specific modifications can be made.

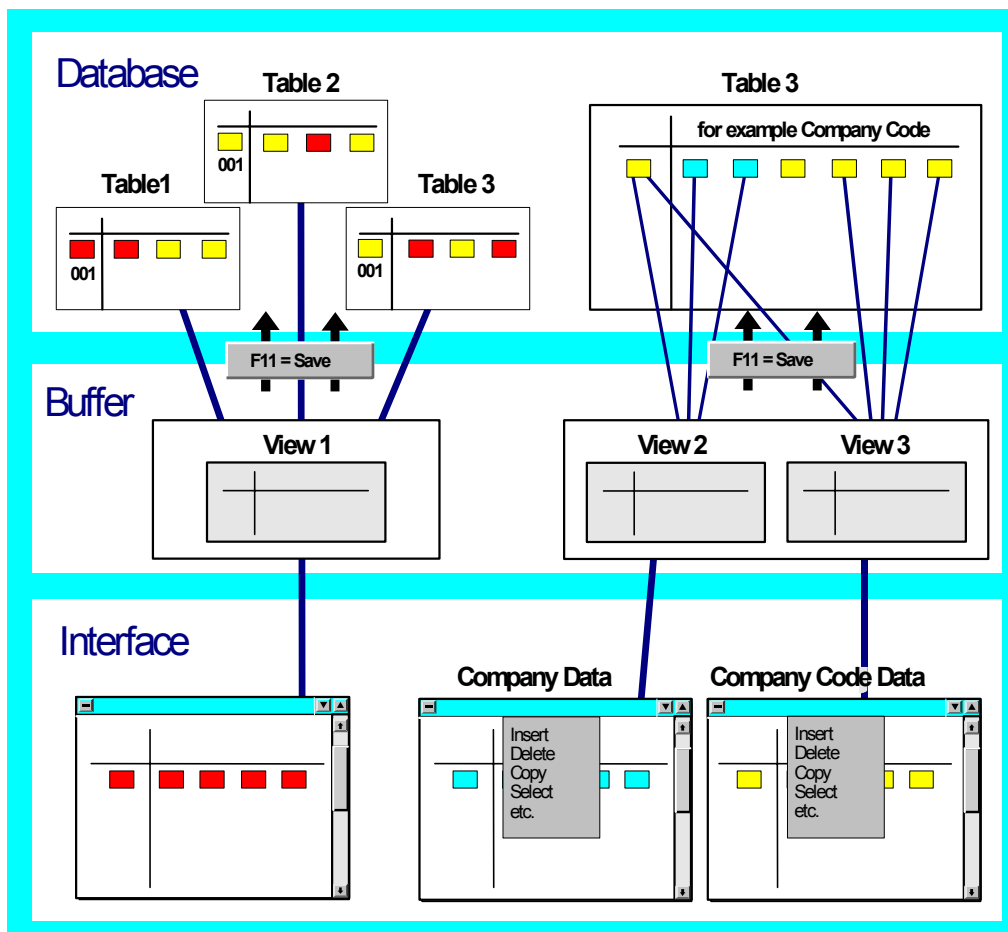
## Concept

The following standard table maintenance dialog functionality is already available in the system for user applications:

- A **central program module**, which contains all the maintenance functionality, including mark functions for multiple record processing, the possibility of selecting all records processed, recover deleted records, and much more besides.
- The **maintenance transaction**, which uses the table or view-specific components as well as the central program module.

### Internal processing sequence

The processing sequence is illustrated using views as an example, in the following figure:



A work area, which can contain all existing views or be restricted by a selection, is fetched from the database for processing on the screen. This work area is first loaded into internal tables. At this point, a further selection for processing by field contents can be made.

Field contents are maintained in the internal tables. The database is not accessed automatically after the maintenance of each individual view record. Changes made are only copied from the

## Concept

internal buffer to the database when the user chooses save. This buffering gives the maintenance processing the form of a transaction. This enables the user to cancel changes before the database access (user-controlled „Rollback“) and allows the calling main program to check entries for consistency in context.

## Maintenance dialog integration

The maintenance dialog can be integrated as follows:

- Maintenance transaction integration in a menu or an application.
- Maintenance dialog integration using function modules in an application with parameter passing.  
There are three possible entry levels, which differ according to the extent to which the maintenance procedure can be controlled.
  - **Highest level**  
With this interface you call the complete maintenance dialog for a table or a view. You can restrict the table entries which are read from the database using selections.
  - **Middle level**  
This level contains the actual maintenance dialog with the steps “Read, Edit, Save” in this order. You must program preparatory actions, such as locking the data, reading the Dictionary, etc., yourself.
  - **Lowest level**  
With this interface you control all maintenance procedures individually from your program. You pass the actions to be performed to the function module as parameters.  
In this way you can easily integrate individual maintenance steps (“Read, Edit, Save”) in your program.



## Create maintenance dialog

You can make the following maintenance components available with the table maintenance dialog generation transaction (*Tools* → *ABAP/4 Workbench* → *Development* → *Other Tools* → *Gen.tab.maint.dialog*):

- **Generated objects**  
Generates all maintenance modules which are required to call the maintenance dialog.
- **Table or view definition** in the Dictionary  
Processes the definition of the current table or view.  
During the definition, table and field level access type authorizations can be specified.  
It can also be specified whether there is to be a detail screen for each overview screen record.



This is necessary for two-step procedures, but also for tables or views whose records can not be completely displayed on the overview screen because of the number or length of their fields. It can also be appropriate when complicated maintenance procedures require long input field explanatory texts, e.g. by displaying data from foreign key tables.

- **Authorization groups**  
You can create authorization groups for tables or views.  
The activities defined for an authorization group apply during the use of the maintenance dialog for the tables or views in the authorization group.
- **Authorization group assignment**  
You can group tables or views.

To be able to call the maintenance dialog, you must generate the table or view-specific maintenance modules. Proceed as follows:

1. Choose *Tools* → *ABAP/4 Workbench* → *Development* → *Other Tools* → *Gen.tab.maint.dialog*. You enter the maintenance transaction initial screen.
2. Enter the name of the table or view.
3. Mark the option “Generated objects”.
4. Choose *Create/Change*.
5. Confirm that the maintenance module is to be created in the following dialog box,.



Instead of steps 1. to 5. you can call the function *Utilities* → *Gen.maint. dialog* in the Dictionary (*Tools* → *ABAP/4 Workbench* → *Development* → *Dictionary*) for the table or view in question. You go to the maintenance screen for the generated object for the current table.

6. Enter the data required for generation:
  - Function group to which the maintenance module is to belong

**Create maintenance dialog**

It is possible to store the maintenance modules for several tables or views. in one function group

- Authorization group
- Maintenance type (one/two-step)
- Maintenance screen (one-step) or screens (two-step) number
- Recording routine (standard/individual or none)

7. Then choose *Create*. All required maintenance modules are now generated.



If you subsequently want to make changes, you must call the function *Change*, to re-generate the maintenance module in question.

Then you can call the standard maintenance dialog or the maintenance function modules for the table or view in question.

## Calling the standard maintenance dialog

If you want to call the standard maintenance dialog in your application, code the maintenance dialog call in your program, and pass the name of the table or view as a parameter.

---

**Call via function module**

## Call via function module

When the maintenance dialog is called via function modules, three entry levels are distinguished:

- **ssHighest level**  
Call of the complete maintenance dialog
- **Middle level**  
You can control the maintenance dialog to a limited extent.
- **Lowest level**  
You call the maintenance object-specific function modules yourself and control the dialog completely.

## Highest level entry

You can call the standard maintenance dialog via the function module VIEW\_MAINTENANCE\_CALL. The function module performs the following activities:

- Authorization check
- Locking
- Fetching and formatting the necessary information from the Dictionary
- Selection, processing and saving the data
- Restrict the data selection in the sub-set field dialog
- Dynamic interface modification (menus and functions)



You can influence the maintenance dialog at run-time in the following ways:

- **Selection conditions**  
You specify the selection conditions with which you wish to restrict the data selection in the database, in an internal table.
- **Interface**  
You can dynamically deactivate functions of the central standard interface SAPLSVIM via an internal table.

Please see the function module documentation in the system for the interface description.

## Middle level entry

## Middle level entry

At this entry level, you call function modules to control the table maintenance.

Call the function module VIEW\_MAINTENANCE.

For this function module, only the table or view name need be specified. The control function module name for the maintenance dialog call is put together and then called.

The function module performs the selection, processing and saving of the data and the interface layout. You must have already performed the following activities yourself before you call VIEW\_MAINTENANCE.

- **Authorization check for the table/view**  
The function module VIEW\_AUTHORITY\_CHECK can be used.
- **Lock the table/view**  
The function module VIEW\_ENQUEUE can be used.
- **Fetching and formatting the required information from the Dictionary**  
The function module VIEW\_GET\_DDIC\_INFO can be used.
- **Possibly restricting the data area in dialog**  
The function module TABLE\_RANGE\_INPUT can be used.



At run-time you can influence the maintenance dialog in the following ways:

- **Selection conditions**  
You save the selection conditions with which you wish to restrict the data selection in the database in an internal table.
- **Interface**  
You can dynamically deactivate functions of the central standard interface SAPLSVIM via an internal table.

Please see the function module documentation in the system for the interface description.

## Interface description

### Import parameters

- **CORR\_NUMBER**  
Change request number of the change made, see function module VIEW\_MAINTENANCE\_CALL documentation
- **VIEW\_ACTION**  
Action (Display, maintain or transport)  
see function module VIEW\_MAINTENANCE\_CALL documentation
- **VIEW\_NAME**  
Name of the table/view to be processed

### Export parameters

none

## Tables

- **DBA\_SELLIST**

Database access selection conditions.

Structure: INCLUDE STRUCTURE VIMSELLIST, see function module VIEW\_MAINTENANCE\_CALL documentation.

All data which are read for table processing and are created by maintenance, are stored at run-time in the internal table TOTAL.

The table TOTAL has the structure:

- INCLUDE STRUCTURE <view name> or <table name>
- INCLUDE STRUCTURE VIMFLAGTAB

- **DPL\_SELLIST**

Selection conditions for the display of part of a work area on the maintenance screens.

Structure and documentation as for DBA\_SELLIST.

The data are stored at run-time in the internal table EXTRACT. The table EXTRACT always contains only the table records which were filtered out of the table TOTAL as a result of a user action.

The table EXTRACT has the same structure as the table TOTAL

- **EXCL\_CUA\_FUNCT**

Interface functions which can be dynamically de-activated.

Structure: INCLUDE STRUCTURE VIMEXCLFUN, see function module VIEW\_MAINTENANCE\_CALL documentation

- **X\_HEADER**

Control block table for the view/table.

Structure: INCLUDE STRUCTURE VIMDESC.

The table contains the table or view header information from the Dictionary, such as sub-set, selection conditions, maintenance status, delivery class. The table also contains information about the table or view generation and event times for user form routines. You can fill this table with the function module VIEW\_GET\_DDIC\_INFO.

- **X\_NAMTAB**

Control block table for the table/view fields.

Structure: INCLUDE STRUCTURE VIMDESC.

The table contains the table or view field information from the Dictionary, such as structure field positions, key information and maintenance characteristics of the field. You can fill this table with the function module VIEW\_GET\_DDIC\_INFO.

## Exceptions

- **MISSING\_CORR\_NUMBER**

Correction number missing

- **NO\_DATABASE\_FUNCTION**

Data processing module missing

---

**Middle level entry**

- **NO\_EDITOR\_FUNCTION**  
Control module missing
- **NO\_VALUE\_FOR\_SUBSET\_IDENT**  
Sub-set field value missing



## Lowest level entry

For this entry level, the function modules generated for data processing are available.

Call the function module VIEW\_MAINTENANCE\_LOW\_LEVEL. Knowledge of the view or table-specific function module name for the data processing is not necessary, as it is put together by the system and then called.

Pass the name of the table or view and the desired function when calling. You must evaluate the returned user commands.

You must also perform all the activities which the maintenance dialog otherwise performs:

- Authorization check
- Lock
- Fetch and format the required information from the Dictionary
- Select, edit and save the data
- Restrict the data selection in dialog for subset fields
- Dynamic interface modification (menus and functions)

## Interface description

### Import parameters

- FCODE  
desired function.
  - 'READ'      Read the data from the DB
  - 'EDIT'              process data
  - 'RDED'      Read and edit
  - 'SAVE'      Write the data to the DB
  - 'ORGL'      Re-set all marked entries
  - 'ORGD'      Re-set one entry
- **VIEW\_ACTION**  
Action (Display, maintain or transport).  
see function module VIEW\_MAINTENANCE\_CALL documentation
- **VIEW\_NAME**  
Name of the table or view.
- **CORR\_NUMBER**  
Change request number for the changes made.  
see function module VIEW\_MAINTENANCE\_CALL documentation

### Export parameters

- **LAST\_ACT\_ENTRY**  
Index of the record in table EXTRACT on which the cursor was positioned.

**Lowest level entry**

- **UCOMM**

Last maintenance dialog user command.

You must process the following commands yourself on this entry level:

- 'SAVE'      Save the data in the DB
- 'ORGL'      Re-set all marked entries in the display table (EXTRACT)

If this command is returned, you must call the lowest entry level function module again with this command and then with the previous command. You do not have to write your own re-set program.



The lowest entry level was called with the module VIEW\_MAINTENANCE\_LOW\_LEVEL and the function 'EDIT'. The user has called the function 'ORGL'. The module VIEW\_MAINTENANCE\_LOW\_LEVEL has now to be called first with the function 'ORGL'. The module runs in the background. Then the module VIEW\_MAINTENANCE\_LOW\_LEVEL has to be called again with the function 'EDIT'.

- 'ORGD'      Re-set the entry in the display table (EXTRACT) header.  
see command 'ORGL' for command processing.
- 'ANZG'      Change action: Change -> Display
- 'AEND'      Change action: Display -> Change
- 'ENDE'              End processing
- 'BACK'             Return to calling position
- 'ATAB'             Fetch another table or view  
This field also contains the commands which were realized in user modules in the maintenance screens.

- **UPDATE\_REQUIRED**

Flag: Entries changed, Save required.

The user has made changes which make it necessary to save the data before leaving the maintenance dialog.

**Tables**

- **CORR\_KEYTAB**

Table with the keys of the entries to be transported. The table is only used in transport mode.

Structure: INCLUDE STRUCTURE E071K

- **DBA\_SELLIST**

Selection conditions for the database access.

Structure: INCLUDE STRUCTURE VIMSELLIST see function module VIEW\_MAINTENANCE\_CALL documentation.

All data which are read in for the table processing or are created during maintenance, are stored in the internal table TOTAL at run time.

The table TOTAL has the structure:

- INCLUDE STRUCTURE <view name> or <table name>
- INCLUDE STRUCTURE VIMFLAGTAB

See also the function module VIEW\_MAINTENANCE\_CALL documentation.

- **DPL\_SELLIST**

Selection conditions for the display of part of a work area on the maintenance screens.

Structure and documentation as DBA\_SELLIST

The data are stored at run time in the internal table EXTRACT. The table EXTRACT always only contains the table records which have been filtered out of the table TOTAL as a result of a user action.

The table EXTRACT has the same structure as the table TOTAL

See also the documentation of the function module VIEW\_MAINTENANCE\_CALL

- **EXCL\_CUA\_FUNCT**

dynamically activated interface functions.

Structure: INCLUDE STRUCTURE VIMEXCLFUN, see function module VIEW\_MAINTENANCE\_CALL documentation.

- **TOTAL**

Data table, contains all data which have been read in and changed, deleted or added during the processing.

Structure:

- INCLUDE STRUCTURE <view name> or <table name>
- INCLUDE STRUCTURE VIMFLAGTAB

All data which are read in for the table processing or are created during maintenance are stored at run time in the internal table TOTAL. After the function has been carried out, the table gets a processing flag for each record processed.

- **EXTRACT**

Data display work table.

Structure as table TOTAL

The data are stored at run time in the internal table EXTRACT. The table EXTRACT always only contains the table records which have been filtered out of the table TOTAL as the result of a user action. After the function has been performed, the table contains all the data found by the last selection for display.

- **X\_HEADER**

Control block table for the table or view.

Structure: INCLUDE STRUCTURE VIMDESC

The table contains the Dictionary header information about the table or view, such as sub-set, selection conditions, maintenance status, delivery class. The table also contains the generation information and event time information for the table or view. You can fill this table with the function module VIEW\_GET\_DDIC\_INFO.

- **X\_NAMTAB**

Control block table for the fields of the table or view.

Structure: INCLUDE STRUCTURE VIMDESC.

**Lowest level entry**

The table contains the field information about the table or view, from the dictionary such as the position of the field in the structure, key information and the maintenance characteristics of the field. You can fill this table with the function module VIEW\_GET\_DDIC\_INFO.

**Exceptions**

- **MISSING\_CORR\_NUMBER**  
Correction number missing
- **SAVING\_CORRECTION\_FAILED**  
Error while saving the entries in a change request.

## Central address management

[Overview \[Page 38\]](#)

---

**Overview**

## Overview

Addresses can arise in many different forms. There are, on the one hand, various kinds of address, e.g. addresses of companies or of private individuals, they can also, e.g. internationally, have a different structure. A central address management has been created to simplify the address management across all applications, and to make access and processing easier.

## Calendar

This section explains which holiday and factory calendar data you can access in the system, and use in your own developments.

[Overview \[Page 40\]](#)

[Concept \[Page 41\]](#)

### Procedure

[Determine calendar ID \[Page 42\]](#)

### References

[Calendar functions \[Page 43\]](#)

---

**Overview**

## Overview

Location-specific calendars can be defined in the SAP system. These can take account of both regional holidays and location-specific conditions of service.

Function modules are available to enable you to use these data in your own developments.



## Concept

Public holidays can be defined and be combined into regionally valid holiday calendars. A holiday calendar is identified in the system by a two-character calendar ID.

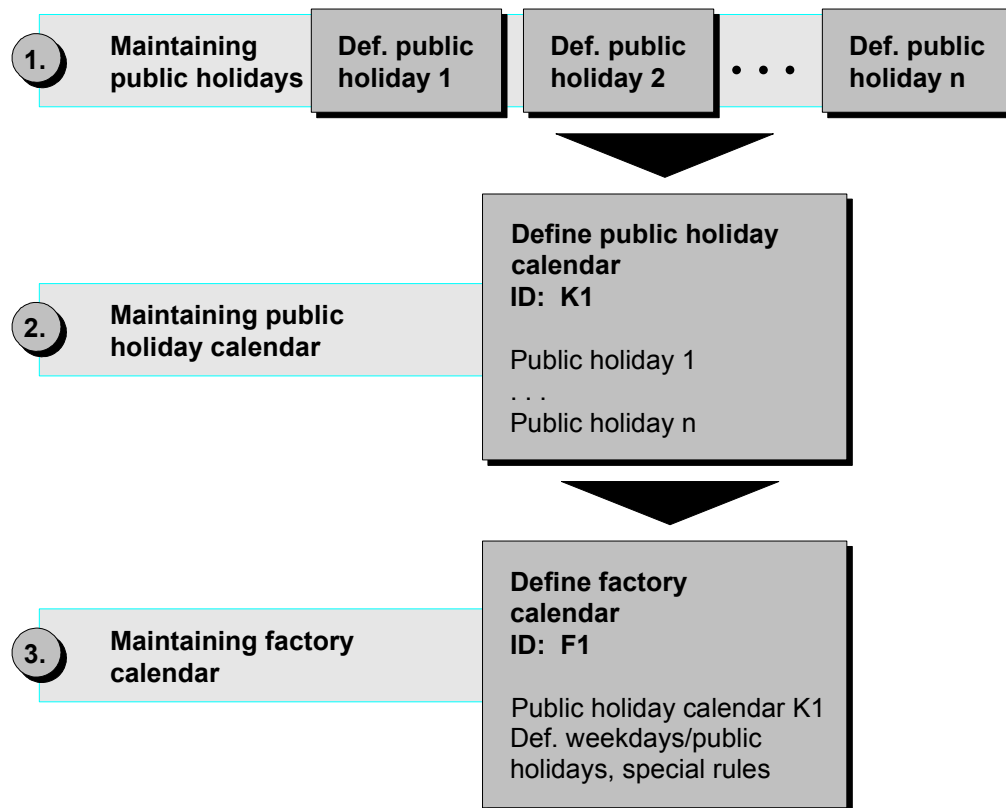
A holiday calendar is assigned to each factory calendar. The following information can also be defined and stored:

- Definition of the working days
- Special conditions

The days which count as working days according to this definition are numbered sequentially from 0 (unless otherwise defined). These numbers represent the factory date. The calendar date can be directly accessed via the factory date, e.g. to calculate delivery dates.

The factory calendar is identified in the system by a two-character calendar ID.

### Calendar hierarchy



---

Determine calendar ID

## Determine calendar ID

Some function modules only get general calendar data. They can be called without any preparation. Others provide data about particular holiday or factory calendars. To call these function modules, you need the relevant calendar ID. To find it, proceed as follows:

1. Call the calendar maintenance transaction in one of the following two ways:
  - In the implementation guide for *Global Settings* choose → *Maintain calendar* → *Execute*
  - Call the transaction SCAL in the OK-Code field.
2. Choose the option "Holiday calendar" or "Factory calendar".
3. Choose *Display*.  
You receive a list of all holiday or factory calendars which exist in the system, with descriptive text and ID.
4. Choose a calendar.  
Call the function *Display. definition*, to get the definition of the chosen calendar.  
Call the function *Display calendar*, to get a calendar overview. In the factory calendars overview screen you can choose a year and display a calendar page for the chosen year with the function *Display year*.

## Calendar functions

All function modules are contained in the function group SCAL.

- **DATE\_COMPUTE\_DAY**  
This function module returns the day of the week for the date passed.
- **DATE\_GET\_WEEK**  
This function module returns the week for the date passed.
- **WEEK\_GET\_FIRST\_DAY**  
This function module returns the first day of the week passed. (This is always a Monday, regardless of whether it is a working day or a holiday.)
- **EASTER\_GET\_DATE**  
This function module returns the date of Easter Sunday for the year passed.
- **FACTORYDATE\_CONVERT\_TO\_DATE**  
This function module returns the calendar date for the factory date and the factory calendar passed.
- **DATE\_CONVERT\_TO\_FACTORYDATE**  
This function module returns the factory date for the date and factory calendar passed. You can specify with a parameter whether the next or the previous working day is returned if the day is not a working day.
- **HOLIDAY\_CHECK\_AND\_GET\_INFO**  
With this function module, you test whether a particular date in the holiday calendar passed is a holiday. If so, the definition of the holiday is returned.

---

**Measurement units**

## Measurement units

This section explains which function modules you can use in your developments for processing measurement units.

[Overview \[Page 45\]](#)

[Concept \[Page 46\]](#)

### Procedure

[Check measurement unit table \[Page 49\]](#)

### References

[Measurement unit conversion \[Page 50\]](#)

[Conversion of measurement units and possible entries \(F4\) help \[Page 52\]](#)

## Overview

Measurement units often have to be converted in applications. In multi-lingual systems, or with language-dependent measurement units, there is also the problem of displaying the relationships between measurement units correctly in the interface. For these reasons, SAP function modules are provided, with which all tasks which arise in connection with measurement units can be performed. The required information for each measurement unit is stored in a measurement unit table, which is accessed by the function modules for performing conversions.

This includes conversion exits, which perform an automatic conversion between external and internal format when using certain domains for screen input/output fields.

The function modules are buffered by default to minimize the number of database accesses needed for the conversion. The buffer is created at the first call of a module in each function group. This call therefore takes somewhat longer to complete. In general though, many modules in both function groups are used repeatedly, so that the effort of creating the buffer is justified.

**Concept**

## Concept

Measurement units measure properties of business applications. These properties comprise physical properties, that can be associated with dimensions in a measurement system, and properties without dimension, that cannot be defined uniquely in a measurement system.

To use electronic data interchange (EDI) you must use the ISO code measurement units. Furthermore, measurement units carry several other names for internal and external presentation in the R/3 system.

### Physical Properties

Physical properties are basic or derived. Derived properties are algebraic combinations of basic properties. Which physical properties are viewed as basic and which as derived is a matter of expediency. There are many different measurement systems with different basic properties. Each basic property defines one basic dimension of a measurement system.

The SAP standard shipment uses the international measurement unit system (SI) with the seven basic properties length, time, mass, temperature, electrical current, light intensity, and molarity. The SI-System has seven basic dimensions.

The basic properties of each dimension can be measured in specific measurement units. The measurement units of the seven basic SI system dimensions are meter (m), second (s), kilogram (kg), Kelvin (K), ampere (A), candela (cd) and mol (mol).

The dimensions of all derived properties of a measurement system are algebraic combinations of its basic dimensions. In the SI system, the measurement units of derived properties are combinations of the SI units and some have their own names and abbreviations.



Derived property	Measurement Unit	Name
Speed	m/s	---
Acceleration	m/s <sup>2</sup>	---
Force	kg m/s <sup>2</sup>	Newton (N)
Energy	kg m <sup>2</sup> /s <sup>2</sup>	joule (J)

You can define any number of other measurement units besides the SI units for each SI system dimension. Different measurement units of one dimension have a linear relationship which allows conversion between them and to the corresponding SI unit.



Dimension	Measurement Unit	Conversion to SI Unit
Length	inch (")	0,0254 m
Mass	ton (t)	1000 kg

Temperature	Celsius (°C)	K - 273,14
Energy	erg (erg)	0.0000001 J

Such definitions can be more convenient for some purposes (for example centimeters and kilometers) or they are country specific (for example feet and miles).

As the relationships between the dimensions as well as the different measurement units of one dimension are defined uniquely, they are maintained centrally in Customizing tables in the R/3 System. The dimensions of derived properties are defined in these tables by defining the exponents of the underlying basic dimensions. The names of these Customizing tables start with T006. You maintain these tables with the transactions CUNI and OMSC. The function modules described in this section use these tables.

In the R/3 System, combinations of basic dimensions must be unique and can be related to one derived dimension only. For example, energy (force times distance) and torque (force times radius) cannot be defined in one R/3 System simultaneously.

In some applications (e.g. SAP Oil & Gas), the temperature and pressure values for which the measurement units of certain dimensions (e.g. volume) are valid must be specified. The measurement units of such dimensions are maintained as temperature and pressure-dependent. The measurement units for this dimension are then defined for a specified temperature and pressure. This is the case for example for natural gas whose volume depends on the temperature and pressure. A basic measurement unit of the dimension volume in the inventory describes the amount of gas under defined conditions.

## Properties without Dimensions

Measurement units for properties without dimensions are important for business applications as well as the measurement units for the seven physical properties of the SI system. These units are used for countable properties. For example palette, box, piece etc. There are no unique relationships between measurement units without dimensions. They depend on the business applications. For example, a box can contain one, six, or twelve pieces.

Conversions between the measurement units without dimensions in the R/3 System are defined material-specifically in the table MARM. You maintain table MARM with transactions MM01 and MM02 in the applications. The function modules described in this section also use this table.

## Abbreviations of Measurement Units – ISO Code

The ISO standard 31 describes measurement units. This standard does not prescribe official abbreviations (ISO codes) for the measurement units. Recommendation 20 of WP.4 of the UN/ECE (UN Economic Commission for Europe, Information Office, Palais des Nations, CH-1211 Geneva, phone +4122917 2893, fax +4122917 0036, e-mail [info.ece@unece.org](mailto:info.ece@unece.org), <http://www.unicc.org/unece/oes/info.htm>) makes recommendations for the ISO codes of measurement units. Since you need the ISO Code for electronic data exchange (EDI), you should maintain the recommended ISO code for each measurement unit in the R/3 System tables.

## Internal and External Measurement Unit Formats

Measurement units in the R/3 System have different internal and external formats. The internal presentation is language-independent, is only for internal processing, and does not appear on the interface. The external format is language-dependent appears on screens. The external format has different names for different uses:

**Concept**

•	commercial	(three upper-case characters)
•	technical	(six-character)
•	short text	(ten-character)
•	long text	(thirty-character)

These names types are maintained language-dependently in table T006A. For example, the commercial name of the dimension-less measurement unit piece is PC in English, ST (Stück) in German, and PI (Pièce) in French.



The commercial and technical formats together with the language form a language-dependent key for the corresponding internal format of the measurement unit. Therefore, they must be maintained uniquely for each language.

The system uses the language-dependent key that is defined from the commercial and technical formats in conversion exits. These conversion exits are called automatically in screens and by the WRITE command if the domains of the data elements involved use the conversion exits LUNIT (for technical measurement units) or CUNIT (for commercial measurement units).



## Check measurement unit table

Before you use the measurement unit function modules, you should ensure that the required measurement units and dimensions are maintained.

To do so, go to the implementation guide in section *Global Settings* → *Check unit of measurement*.

---

Measurement unit conversion

## Measurement unit conversion

### Function group SCV0

This function group contains the following function modules:

- **BUFFER\_CONTROL**  
If the default buffering is not wanted for the modules in this function group, you can switch it off and on again by calling this module. You can also use the module to refresh the buffer (if, e.g. the contents of the table T006 were changed during the program run).
- **CONVERSION\_FACTOR\_GET**  
With this function module, you determine the conversion factors for the conversion of a measurement unit into another using the measurement units table. This does not apply to measurement units within a dimension.  
The module also returns the number of decimal places to which the values in the unit UNIT\_OUT are to be rounded. This also applies to units with no dimension.  
The following formula applies for the conversion:  
$$(\text{value in the unit UNIT\_OUT}) = (\text{value in the unit UNIT\_IN}) * \text{numerator/denominator} + \text{additive constant.}$$
- **ROUND**  
With this function module, you round a value to the specified number of decimal places. You can choose between three rounding types:
  - Rounding up
  - Rounding down
  - Commercial roundingThe rounding is performed internally with the same field type as that of the field passed. Rounding errors can thus occur when rounding a FLOAT value. If you want a high degree of accuracy, the passed field should have the type P.
- **SI\_UNIT\_GET**  
You pass either a unit or a dimension to this function module to get the SI unit. If you pass both a unit and a dimension, the SI unit for the dimension is returned.
- **UNIT\_CONVERSION\_WITH\_FACTOR**  
With this function module, you convert a value according to the factor passed.
- **UNIT\_CORRESPONDENCE\_CHECK**  
With this function module, you can check whether the two units passed belong to the same dimension.
- **UNIT\_GET**  
With this function module, you get the appropriate measurement unit for the specified dimension and conversion factor.
- **UNIT\_CONVERSION\_SIMPLE**  
With this function module, you convert a value using the measurement unit table, and round it, if appropriate.  
You can also perform the rounding without conversion.  
Conversion with this function module requires that the measurement unit table is maintained for both units, and that both units belong to the same dimension, i.e. also that they have

---

**Measurement unit conversion**

dimensions.

The rounding can, however, also be performed for units which have no dimension.

---

s Conversion of measurement units and possible entries (F4) help

## s Conversion of measurement units and possible entries (F4) help

### Function group SCVU

This function group contains the following function modules:

- **BUFFER\_CONTROL\_SCVU**  
If you do not want the default buffering for the modules in this function group, you can switch it off, and on again, by calling this module. You can also use the module to refresh the buffer (e.g. if the contents of the table T006 were changed during the program run).
- **CONVERSION\_EXIT\_CUNIT\_INPUT**  
With this function module, you specify the internal measurement unit for a commercial measurement unit (three-character external measurement unit).  
It is automatically called when measurement units are input on the screen.
- **CONVERSION\_EXIT\_CUNIT\_OUTPUT**  
With this function module, you specify the language-dependent commercial measurement unit (three-character external measurement unit) and the associated short and long text, for an internal measurement unit.  
It is automatically called when measurement units are output to the screen, and by the WRITE command.
- **CONVERSION\_EXIT\_LUNIT\_INPUT**  
With this function module, you specify the internal measurement unit associated with a technical measurement unit (six-character external measurement unit).  
It is called automatically when measurement units are input on the screen.
- **CONVERSION\_EXIT\_LUNIT\_OUTPUT**  
With this function module, you specify the language-dependent technical measurement unit (six-character external measurement unit) and its associated short and long text for an internal measurement unit.  
It is called automatically when measurement units are output to the screen, and by the WRITE command.
- **DIMENSION\_CHECK**  
With this function module, you check whether the internal measurement unit corresponds to the specified dimension. It may also be checked whether it is a commercial unit. It is recommended, that you support the choice of valid measurement units for a specified dimension with the function module UNIT\_OF\_MEASUREMENT\_HELP.
- **DIMENSION\_GET**  
With this function module, you specify the dimension key and the dimension text, depending on the contributions of the basic units.  
As the seven possible contributions have the default value zero, you must only specify the non-zero contributions under EXPORTING when calling.
- **DIMENSION\_GET\_FOR\_UNIT**  
With this function module, you specify the dimension key associated with a measurement unit
- **UNIT\_OF\_MEASUREMENT\_HELP**  
With this function module, you display in a dialog box either all measurement units or all commercial measurement units of a specified dimension (external measurement unit and

---

**s Conversion of measurement units and possible entries (F4) help**

associated long text). If you do not specify a dimension, all measurement units are displayed. You can control whether the measurement units are only displayed, or are offered for selection, with a parameter.

---

**Change documents**

## Change documents

This section describes how you can log application changes, using change documents.

[Overview \[Page 55\]](#)

[Concept \[Page 56\]](#)

### Procedure

[Procedure \[Page 58\]](#)

[Define change document object \[Page 59\]](#)

[Set change document flag \[Page 61\]](#)

[Generate update and INCLUDE objects \[Page 62\]](#)

[Integrating the functionality into the program \[Page 64\]](#)

[Writing the fields in the program \[Page 65\]](#)

### References

[Creating change documents \[Page 69\]](#)

[Read and format change documents \[Page 70\]](#)

[Read and format planned changes \[Page 71\]](#)

[Delete change documents and planned changes \[Page 72\]](#)

[Archived change documents management \[Page 73\]](#)

## Overview

Many commercial objects are frequently changed. It is often useful, or even necessary, to be able to trace the changes made. If changes are logged, you can find out at any time, what was changed and when and how the change was made. This can sometimes make the analysis of errors easier. In financial accounting, for example, change documents are used to make auditing possible.

Changes are logged in change documents, which can be created for actual or planned changes.

## Concept

### Concept

For changes to a commercial object to be able to be logged in a change document, the object must have been defined in the system as a change document object. A **change document object** definition contains the tables which represent a commercial object in the system. The definition can also specify whether the deletion of individual fields is to be documented. If a table contains fields whose values refer to units and currency fields, the associated table, containing the units and currencies, can also be specified.

It must be specified for each table, whether a commercial object contains only one (single case) or several (multiple case) records. For example, an order contains an order header and several order items. Normally one record for the order header and several records for the order items are passed to the change document creation when an order is changed.

The name under which a change document object is created is an **object class**.



The object class BANF was defined for the change document object "Purchase requisition", which consists of the tables EBAN (purchase requisition) and EBKN (purchase requisition account assignment).

Changes to this commercial object can then be saved in the system under the object values of this change document object, i.e. the **object ID** and a change document number. The object ID is the key to the object value, i.e. all records which are defined as belonging to a given change document object.

All changes to a commercial object constitute an object value under this key. This is for example the order number for orders or the number range object name for number range objects. All changes to a given order or to a given number range object can be accessed in this way.



The object value BANF with the object ID "3000000000" consists of the records of the tables EBAN and EBKN with the order number "3000000000".

If changes are not yet to be made, but are planned, they can be logged as **planned changes**. A planned date for the changes can be specified. The planned changes can be analyzed and copied into the tables. You must program the copy yourself.

All logging functions are supported by SAP function modules. The application development must contain certain INCLUDE programs. Old and new status are passed to the change document creation. The included function modules determine the changes for all table fields which are flagged as being change-relevant in the Dictionary.

### Change document

A change document logs changes to a commercial object. The document is created independently of the actual database change. The change document structure is as follows:

- **Change document header**  
The header data of the change to an object ID in a particular object class are stored in the change document header. The change document number is automatically issued.
- **Change document item**  
The change document item contains the old and new values of a field for a particular change,



and a change flag.

The change flag can take the following values:

- **U(pdate)**  
Changed data. (Log entry for each changed field which was flagged in the Dictionary as “change document-relevant”)
  - **I(nsert)**  
Data inserted.  
Changes: Log entry for the whole table record  
Planned changes: Log entry for each table record field
  - **D(elete)**  
Data were deleted (log entry for the whole table record)
  - **I(ndividual field documentation)**  
Delete a table record with field documentation  
1 log entry per field of the deleted table entry, the deleted text is saved
- **Change document number**  
The change document number is issued when a change is logged, i.e. when the change document header is created by the change document creation function module (function group SCD0).



The change number is not the same as the change document number. The **change document number** is issued automatically by the function group SCD0 function modules when a change document is created for a change document object. The **change number** is issued by the user when changes are planned. The same change number can be used for various change document objects.

## Internal processing

When the object-specific update is called, the object-specific change document creation is called. The object-specific change document header is written with a change document number. The Dictionary is searched for which fields are to be logged for each table in the object definition. The log records for these fields are then created as change document items according to the object definition.

---

**Procedure**

## Procedure

To use the change document functionality in your application, proceed as follows:

1. Define the change document object
2. Check in the Dictionary, whether the data elements of the fields which are to be logged are flagged appropriately.
3. Generate the update.
4. Program the appropriate calls in your program.

## Define change document object

Proceed as follows:

1. Call the change document maintenance transaction (*Tools* → *ABAP/4 Workbench* → *Development* → *Other tools* → *Change doc. object*). An overview of existing change document objects is displayed.
2. Choose the menu option *Create*.
3. Enter a name for the change document object which is to be created. It can be any name starting with "Y" or "Z" (customer name area).
4. Choose *Continue*. A new window for inputting the associated tables appears.
5. Enter a descriptive short text for the change document object.
6. Make the following entries for each table whose changes are to be logged in the change document for this change document object:
  - *Table name*  
Name of the table, as defined in the Dictionary
  - *Copy as internal table flag*.  
If the change data are to be passed in an internal table (multiple case), mark this field. If it is not marked, the change data are passed in a work area (single case).
  - *Doc. for individual fields at delete flag*  
If you want separate log entries for each field when data are deleted, mark this field. If it is not marked, the deletion of all relevant fields is entered in one document item.
  - *Ref. table name*. (Name of the reference table)  
If the currency and unit fields are defined in a reference table, rather than in the table passed, you must pass the name of the reference table, and the field referred to, to the function module. Create an INTTAB structure in the Dictionary, and define fields for this structure, which are made up of the names of the associated reference table and the reference fields.  
Enter the name of this structure here.  
In the individual case, the reference information is passed in the form of two extra work areas (old, new). In the collective case, the internal tables are extended to include the reference structure.
  - *Name of the old record fields*  
Only possible for single case, i.e. when passing change data in a work area: If you do not want to use the \* work area, enter an alternative work area name here.
7. After inputting all relevant tables, choose *Insert entries*. The new entries are copied into the display.
8. Save your entries.

## Transport change document object

The change document objects are a transport object type, a change request is made when the object is created.

During transport the object-specific update is generated in the target system.

---

Define change document object

## Set change document flag

Now check whether the change document flag is set for the corresponding data element in the Dictionary for the fields whose changes are to be logged. This is necessary so that the object-specific function modules can identify which field of the defined object should be entered in the change document during logging.

If the flag is not set, you can change it. The flag becomes effective after the activation.



If the flag is set by hand, it can have undesirable side-effects: If a table field in another application, which is based on the data element in question, belongs to a change document object, but was not previously logged, setting the flag will start logging in this application as well.

It is therefore important to consider whether data elements are, or could be, change-relevant when creating them, and to set the flag accordingly. If the data element is not in any change document object via a table field, this has no negative effect on the system.

---

Generate Update and INCLUDE Objects

## Generate Update and INCLUDE Objects

The generation creates INCLUDE objects, which contain general and specific data definitions and the program logic for the update function module. Proceed as follows:

1. Call the change document maintenance transaction (*Tools* → *ABAP/4 Workbench* → *Development* → *Other tools* → *Change documents*).
2. Position the cursor on a change document object and choose the menu option *Generate update pgm*. A dialog box, in which you must make the following entries, is displayed:
  - *maximum 26 character INCLUDE name*  
This 26-character name (<K4>) is used to complete the name of the generated INCLUDE program parts.
  - *Function group*  
Enter the name of the function group to which the change document update program is to belong. If this function group does not yet exist in the system, it is automatically created during generation. Exactly one function group must belong to each change document object. Other function modules may not be assigned to this function group.
  - *FM structure prefix (12-char.)*  
Multiple-case table transfer structures are created at generation. Their names are constructed from this prefix and the name of the multiple case tables. A value is proposed.  
  
An update-compatible function parameter must not be longer than 28 characters, so the prefix and the longest table name must not be longer than 28 characters together.
  - *Error message ID*  
The application-specific error messages generated are stored under this message ID (work area). A value is proposed.
  - *Error number*  
Number with which errors occurring in connection with this change document object can be identified in the system. A value is proposed.
  - *Processing type*  
Update type flag:
    - immediate
    - delayed
    - in dialog
  - *Special text handling flag*  
Select this field to log long text changes.  
  
The old and new status of long texts is not logged. Only the fact that they have been changed is noted.
3. Choose *Generate*.  
  
The following INCLUDE objects are generated:

Generate Update and INCLUDE Objects

– **<Change document object>\_WRITE\_DOCUMENT**

The object-specific update function module calls the following function modules with object-specific parameters:

<b>CHANGEDOCUMENT_OPEN</b>
CHANGEDOCUMENT_SINGLE_CASE and/or
CHANGEDOCUMENT_MULTIPLE_CASE and possibly
CHANGEDOCUMENT_TEXT_CASE
<b>CHANGEDOCUMENT_CLOSE</b>

– **F<K4>CDC**

INCLUDE program part with FORM statement for calling the object-specific update program.

– **F<K4>CDT**

INCLUDE program part containing two INCLUDE program parts (F<K4>CDF and F<K4>CDV, see below), which contain the data definitions which are to be passed to the update program. The data definitions correspond to the function group SCD0 function modules interface definition. The fields, record fields and tables are to be filled in in the application program and passed to the update program.

– **F<K4>CDF**

INCLUDE program part with data definitions which are the same for all change document objects.

– **F<K4>CDV**

INCLUDE program part with data definitions, which are specific to the change document object.

– **V<Dictionary structure name >** (only for multiple case tables)

This structure comprises the following INCLUDE structures:

INCLUDE	<table name>
INCLUDE	KZ
INCLUDE	<ref. table name>

The generated program parts contain the object-specific program code, and are included in the application program per INCLUDE statement. The data definitions of the change document-relevant fields correspond to the function group SCD0 function module interface definitions.



If several change document objects are to be processed in one application program, the update program must be generated for each change document object with a different <K4> code (e.g. XX01, XX02, XX03 etc.).

---

Integrating the functionality into the program

## Integrating the functionality into the program

1. Include the generated program parts in your program code with an INCLUDE statement.
2. When application changes are made, complete the change-relevant fields as appropriate.
3. To create the change document, call the object-specifically generated update program with a PERFORM statement using the name defined in F<K4>CDC.



The INCLUDE program part F<K4>CDT can only be included once, because it contains a further INCLUDE program part (F<K4>CDF, with generally valid data definitions), which is also contained in the other F<K4>CDT program parts (e.g. FXX02CDT, FXX03CDT, etc.). It must be included in the global data definitions. In this case the F<K4>CDT program part must be included for the first change document object or <K4> code, which contains the INCLUDE program parts for general (F<K4>CDF) and object-specific data definitions (F<K4>CDV), for all others only the F<K4>CDV program parts.



## Writing the fields in the program

Complete the change document-relevant fields and tables as follows.

### General data

These are the fields which are defined in the INCLUDE program part **F<K4>CDF**.

- **OBJECTID**  
Object value (key) of the object
- **TCODE**  
Transaction, with which the change was made
- **UTIME**  
Change time
- **UDATE**  
Change date
- **USERNAME**  
Changed by

### Object-specific data

These are the fields which are defined in the INCLUDE program part **F<K4>CDV**.

#### Single case tables:

- Table **\*<table name >** or record fields **<old record fields name >** (with the table structure)  
The table header record or the record fields must contain the original data.
- Table **<table name >**  
The table header record must contain the new data.
- Table **\*<ref. table name >**  
(only if *ref. tab. name* was specified when the change document object was defined)  
The table header record must contain the original currencies and units.
- **UPD\_<table name >**  
With this flag, you specify the processing logic.  
The following values are possible:
  - **"D"** (DELETE)  
A change document item is to be created for the record in **\*<table name >** or **<old record fields name >** which is to be flagged as deleted. **<table name >** is not processed.
  - **"I"** (INSERT)  
A change document item is to be created for the record in **<table name >** which is to be flagged as created. **\*<table name >** or **<old record fields name >** is not processed.
  - **"U"** (UPDATE)  
**\*<table name >** or **<old record fields name >** and **<table name >** are compared and a change document item is created for each changed field. The keys of **\*<table name >** or **<old record fields name >** and **<table name >** must be identical.

## Writing the fields in the program

- " " (space, no processing)  
\* <table name > or <old record fields name > and <table name > are not processed by the update program. (If no changes have been made, the processing can be skipped to save time.)

### Multiple case tables:



These tables must be passed sorted by key.

- **Y<table name >**  
The table must contain the original version of the changed or deleted records. The structure consists of the table, as specified in the change document object definition under *table name*, a processing flag (TYPE C, length 1) and possibly the structure of the associated currency and units table, as specified in the definition of the change document object under *Ref. table name*. It is created during the change document object INCLUDE generation and saved under the name V<table name > in the Dictionary.

The processing flag can be switched from space to "D", if it is to be processed in the application. Otherwise it has no effect.

- **X<table name >**  
The table must contain the current version of the changed or created records. The structure is the same as Y<table name > (see above).  
The following values are possible for the processing flag:

- "I" (INSERT)  
Records were created, or table records were deleted, then a record with the same key was created in the same transaction, and this is to be documented as "Delete" and "Create" (special case), not as "Change".
- "U" or " " (space) (UPDATE)

The parameter UPD\_<table name > (see below) initially determines whether the record is new or changed. The processing flag is only checked when, with the parameter value "U", the following key comparison between the two tables TABLE\_OLD and TABLE\_NEW finds two records with the same key.



Multiple case internal table processing flags can always contain space, with the exception of the special case (in X<table name >). The possibility of setting the processing flag to "D", "I" or "U" as well was created so that the tables could also be used for other purposes in which such processing flags are useful, in application programs.

- **UPD\_<table name >**  
With this flag you determine the processing logic.  
The following values are possible:
  - "D" (DELETE)  
A change document item is to be created for each record in Y<table name > which is to be flagged as deleted. X<table name> is not processed.
  - "I" (INSERT)  
A change document item is to be created for each record in X<table name> which is to be flagged as created. Y<table name> is not processed.

## Writing the fields in the program

- "U" (UPDATE)  
The keys of TABLE\_OLD and TABLE\_NEW are compared. The following cases are distinguished:
  - **1. Record exists in TABLE\_OLD but not in TABLE\_NEW:** Change document items are to be created for the record in TABLE\_OLD which is to be deleted.
  - **2. Record exists in TABLE\_NEW but not in TABLE\_OLD:** A change document item is to be created for the records in TABLE\_NEW which are to be flagged as created.
  - **3. Record exists in both TABLE\_OLD and TABLE\_NEW:** A change document item is created for each changed field which is defined as change document-relevant in the Dictionary.
- " " (space, no processing)  
Y<table name> and X<table name > are not processed by the update program. (If no changes have been made, the processing can be skipped to save time.)

### Text changes:

If text changes are to be logged (according to the change document object definition), the following fields are to be completed:

- **ICDTEXT\_<Object>**  
This structure contains the change document-relevant texts with corresponding details:
  - TEILOBJID  
Key of the changed table record
  - TEXTART  
Text type of the changed texts
  - TEXTSPR  
Language key
  - UPDKZ  
Change flag for the table record: **D**(elete), **I**(nsert) or **U**(pdate)
- **UPD\_ICDTEXT\_<Object>**  
Change flag for the text table:
  - " " (space)  
Table is ignored by the update program
  - "U"  
Table is taken into account by the update program

### Optional parameters

You can also use the following INCLUDE program part **F<K4>CDF** parameters:

- **CDOC\_PLANNED\_OR\_REAL**  
With this parameter you control whether the changes to be logged are actual or planned changes.  
Possible values
  - "R"            actual (real) changes
  - "P"            planned changes

---

**Writing the fields in the program**

- “ ”(space) if no plan number exists: actual change  
if a plan number exists: planned change
- **CDOC\_UPD\_OBJECT**  
If the change document is relevant for determining which change action was performed for the object, you can pass the action performed here.  
Possible values:
  - “I” the object was inserted.
  - “U” the object was changed.
  - “D” the object was deleted.

## Creating change documents

### Function group SCD0

Object-specific update change documents for a particular object ID are created with the function modules in this function group.



These function modules are called, in the right order, by the object-specifically generated update program, as soon as it is called. They are generally not required for application developments. Only in exceptional cases, in which an individual update is to be programmed, should the change document creation be programmed by the user with these function modules.

- **CHANGEDOCUMENT\_OPEN**  
This function module is required by every change document creation. It initializes the internal fields for a particular change document object ID.
- **CHANGEDOCUMENT\_MULTIPLE\_CASE**  
This function module creates change document items. The change data are passed in tables.
- **CHANGEDOCUMENT\_SINGLE\_CASE**  
This function module creates change document items. The change data are passed in a work area.
- **CHANGEDOCUMENT\_TEXT\_CASE**  
Change document-relevant texts are passed in a structure with this function module.
- **CHANGEDOCUMENT\_CLOSE**  
This function module is required for every change document creation. It writes the change document header for a particular change document ID, and closes the document creation.
- **CHANGEDOCUMENT\_PREPARE\_TABLES**  
With this function module, you compare the records in two tables, which you pass as TABLE\_OLD and TABLE\_NEW.  
You can specify via a parameter, whether these internal tables should be prepared for the multiple case. Identical records are then deleted, and a processing flag is set in changed records.

---

Read and format change documents

## Read and format change documents

Two function groups exist for these tasks:

### Function group SCD1

With the function modules in this function group, you can read change documents.

- **CHANGEDOCUMENT\_READ\_HEADERS**  
This function module reads the change document numbers, with the associated header information, for a particular change document object. The search can be restricted by various parameters (changed by, date, time).  
You can use this function module in the database and in the archive.
- **CHANGEDOCUMENT\_READ\_POSITIONS**  
This function module reads the change document items for a given change document object number, and formats the old and new values according to their type.  
You can use this function module in the database and in the archive.
- **CHANGEDOCUMENT\_PREPARE\_POS**  
You format a previously read change document item for printing with this function module.

### Function group SCD2

You can process change document objects by classes with the function modules in this group.

- **CHANGEDOCUMENT\_READ**  
With this function module, you read change document headers and the associated items for a given object class and format the old and new values according to their type. The search can be restricted by various parameters (changed by, date, time).  
You can use this function module in the database and in the archive.

## Read and format planned changes

### Function group SCD3

With the function modules in this function group, you find the planned changes.

- **PLANNED\_CHANGES\_READ\_HEADERS**

With this function module, you find the document headers of planned changes for a given change document object. The search can be restricted by various parameters (changed by, date, time, change number).



The change number is not the same as the change document number. The **change document number** is automatically issued by the function group SCD0 function modules when a change document object change document is created. The **change number** is assigned by the user when changes are planned. The same change number can be used for various change document objects.

- **PLANNED\_CHANGES\_READ\_POSITIONS**

This function module reads the change document items for a given change document number, and formats the old and new values according to their type.

---

Delete change documents and planned changes

## Delete change documents and planned changes

### Function group SCD4

With the function modules in this function group, you delete log entries of changes or planned changes.

- **CHANGEDOCUMENT\_DELETE**  
This function module deletes the change documents for a given change document object. The deletion can be restricted to a given change document number and/or a change date. An authorization check is made before the deletion.
- **PLANNED\_CHANGES\_DELETE**  
With this function module, you delete planned changes. The deletion can be restricted to a given change document object, a change document number, or specified change numbers.



The change number is not the same as the change document number. The **change document number** is issued automatically by the function group SCD0 function modules when a change document object change document is created. The **change number** is issued by the user when changes are planned. The same change number can be used for various change document objects.



## Archived change documents management

### Function group SCD5

This function module is the archiving class for the change document (see also [Archiving \[Page 114\]](#) in the archiving section in this document):

- **CHANGEDOCU\_ARCHIVE\_OBJECT**  
With this function module, you pass the objects for which change documents are to be archived, to the archiving.

---

**Create application log**

## Create application log

This section explains how you can log events in the application log in your application.

The function modules described here (beginning with APPL\_LOG\_\*) exist since Release 3.0. New, more flexible and powerful function modules (beginning with BAL\_\*) exist since Release 4.6.

The three most important function modules are:

- BAL\_LOG\_CREATE  
Open a log
- BAL\_LOG\_MSG\_ADD  
Put a message in the log
- BAL\_DSP\_LOG\_DISPLAY  
Display log

Function module documentation exists for these and all other function modules. There is also a complete technical documentation.

You can still use the old function modules (before Release 4.6). They now call the new function modules from Release 4.6.

## Overview

Application events can be centrally logged in the application log. The advantage is system-wide standardized and uniform event logging, which is convenient to analyze.

Several different logs (for various objects) can be written at the same time by an application.

The application log is, in principal, similar to the system log. Whereas system event information is logged in the system log, relevant application events should be captured in the application log.

The application log can also be used as a message collector.

## Concept

## Concept

Application log objects are defined in the system. The object definition assigns a work area. An object can be divided into sub-objects.

Logging is performed object-specifically, via function modules.

An object log entry has the following structure:

- **Log header** with a unique log number.  
It contains the information, who, when, with which program or which transaction, gave rise to which event.  
It also contains the problem class of the message in the log with the greatest urgency.
- Any number of **log messages** with their urgency.  
The messages are divided into problem classes according to their urgency.

The log data are initially collected in local memory, and are then written to the database. This procedure speeds up processing and reduces the number of database accesses. It is also possible to write log data to the database individually, to avoid losing the log records collected up to that point in the event of termination of the application, for example if the system crashes.

The logged data can be read in the database and displayed on the screen. It is also possible to read and to display the log data which is buffered in local memory with a log number (Message collector).

The logs have an expiry date, by which time at the latest they must be in the database. They can later be removed from the database again, by a delete program.

Detailed information, either for the whole log or for each individual log message, can be saved in two ways:

- Text module with any number of parameters
- User exit with any number of parameters

When the log is analyzed either the text module with the specified parameters is displayed, or the user exit is performed, on request.

Additional information can be saved in an INDX-type table, which is used by the user exit analysis.

In this way it is possible, for example, to save lists which can be displayed when the log is analyzed, with the help of the user exit.

Classifying attributes can also be specified (importance of the log, or of the message).

If you want to perform your own log analysis, you can use the function modules to read from the database or from local memory.

Logs can be deleted if necessary.

## Procedure

Before you can log events, you must first define an object, and possibly sub-objects. Application log objects are maintained in an extra maintenance transaction.

You can then create and analyze the object log by calling the appropriate function modules.

---

Define application log objects

## Define application log objects

1. Call the maintenance transaction with *Tools* → *ABAP/4 Workbench* → *Development* → *Other tools* → *Application log*.
2. Choose *New entries*. An empty input area is displayed.
3. Enter an object name according to the naming convention:
  - first character “Y” or “Z”
  - second and third character: application ID (e.g. FI)
  - fourth position, any character
4. Enter a descriptive short text.
5. Save your entries.
6. If you want to define sub-objects:
  - a) Choose the line with the object.
  - b) Choose *Table view* → *Other view*. A structure overview is displayed for selection.
  - c) Position the cursor on “Sub-objects”, and choose *Choose*. The sub-object display window for the chosen object is displayed.
  - d) Choose *New entries*.
  - e) Enter a sub-object name (beginning with “Y” or “Z”) and a descriptive short text.
  - f) Save your entries.



If several systems are being used, the object data must be transported. Sub-object data are not automatically transported with the object. They must each be entered separately in a change request.

## Create application log

### Function group SLG0

You write the application log records with these function modules.

- **APPL\_LOG\_WRITE\_HEADER**  
With this function module, you write the log header data in local memory.
- **APPL\_LOG\_WRITE\_LOG\_PARAMETERS**  
With this function module, you write the name of the log parameters and the associated values for the specified object or sub-object in local memory.  
If this function module is called repeatedly for the same object or sub-object, the existing parameters are updated accordingly.  
If you do not specify an object or sub-object with the call, the most recently used is assumed.
- **APPL\_LOG\_WRITE\_MESSAGES**  
With this function module you write one or more messages, without parameters, in local memory.
- **APPL\_LOG\_WRITE\_SINGLE\_MESSAGE**  
With this function module you write a single message, without parameters, in local memory. If no header entry has yet been written for the object or sub-object, it is created.  
If you do not specify an object or sub-object with the call, the most recently used is assumed.
- **APPL\_LOG\_WRITE\_MESSAGE\_PARAMS**  
With this function module you write a single message, with parameters, in local memory. Otherwise the function module works like APPL\_LOG\_WRITE\_SINGLE\_MESSAGE.
- **APPL\_LOG\_SET\_OBJECT**  
With this function module, you create a new object or sub-object for writing in local memory. With a flag you can control whether the APPL\_LOG\_WRITE\_... messages are written in local memory or are output on the screen.
- **APPL\_LOG\_INIT**  
This function module checks whether the specified object or sub-object exists and deletes all existing associated data in local memory.
- **APPL\_LOG\_WRITE\_DB**  
With this function module you write all data for the specified object or sub-object in local memory to the database.  
If the log for the object or sub-object in question is new, the log number is returned to the calling program.

---

**Display application log**

## Display application log

### Function group SLG3

With these function modules you display logs for analysis.

- **APPL\_LOG\_DISPLAY**  
With this function module you can analyze logs in the database.
- **APPL\_LOG\_DISPLAY\_INTERN**  
With this function module you can analyze logs in local memory, e.g. when you have only collected log records at runtime and do not want to write to the database.



## Read application log

### Function group SLG1

If you want to analyze the log yourself, you can read the logs with these function modules.

- **APPL\_LOG\_READ\_DB**  
With this function module you read the log data in the database for an object or sub-object according to specified selection conditions.
- **APPL\_LOG\_READ\_INTERN**  
With this function module you read all log data whose log class has at least the specified value, from local memory, for the specified object or sub-object.

---

**Delete application log**

## Delete application log

### Function group SLG2

With this function module you delete logs.

- **APPL\_LOG\_DELETE**  
With this function module you delete logs in the database according to specified selection conditions.

## Platform-independent File Name Assignment

This section explains how to use platform-independent file names in your application programs to address files to be stored.

[Overview \[Page 84\]](#)

[Definitions of Platform-independent File Names \[Page 85\]](#)

[The Function Module FILE GET NAME \[Page 88\]](#)

[Using Platform-independent File Names in Programs \[Page 91\]](#)

[Reference \[Page 92\]](#)

---

Overview

## Overview

Application data must often be stored in files outside the database. Depending on the particular operating system in use, files are stored in different directories, and file and path names must comply with different syntax requirements. Therefore, many SAP application programs use platform-independent logical file names and call the function module `FILE_GET_NAME` when storing data in files. The function module takes a logical file name as input and returns the corresponding platform-specific file name and path.

By using this function module, you can assign file names in your application programs in a standardized way and independently of different hardware and software platforms.

To achieve this, logical file names and paths must be defined in the system. These definitions are maintained in the implementation guide in section *Basis Components* → *System Administration* → *Platform-independent File Names*, or with transactions `FILE` and `SF01`.

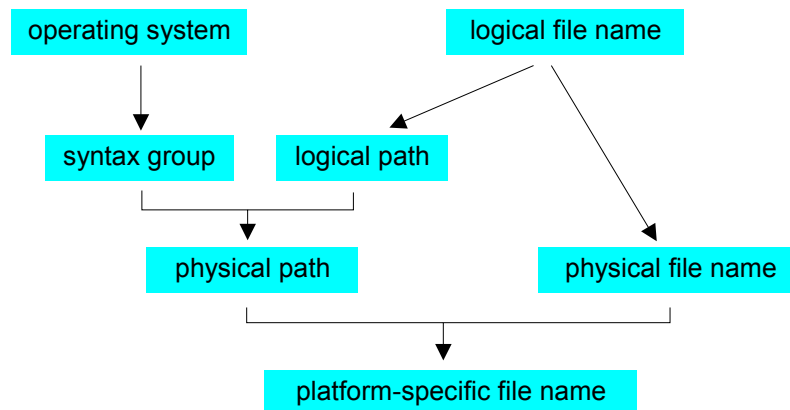
Definitions used by SAP applications are delivered with the system and possibly adjusted in the implementation process. See the application-specific documentation for information on which applications use which logical file names. Further definitions can be added according to requirements.

## Definitions of Platform-independent File Names

The conversion of a platform-independent file name to a platform-specific one is controlled by definitions that are stored in tables. These definitions refer to the following objects:

- Operating systems and Syntax groups**  
 All operating systems are assigned to syntax groups. A syntax group is a group of operating systems that share a common syntax for file names and paths. The definition of a syntax group specifies, for instance, how long file names may be, and whether file name extensions are permitted or not.
- Logical file name**  
 A logical file name is a platform-independent descriptive name for a file to be stored in the file system. Its definition applies to all clients of an R/3 system. In addition, it is possible to specify client-specific definitions of a logical file name.
- Physical file name**  
 A physical file name is assigned to every logical file name.
- Logical path**  
 A logical path is a platform-independent descriptive name for a path where files are to be stored. For the conversion of a logical file name to work for different platforms, it is necessary that a logical path be assigned to that logical filename.
- Physical path**  
 One or more physical paths are assigned to a logical path, each one applying to a different syntax group (platform).

The following figure shows the relationships between these objects that determine how a logical file name is converted to a platform-specific file name:



### Parameters in physical file names and paths

Physical file names and paths may contain the following keywords enclosed in angle brackets which are replaced at runtime:

**Table: Keywords**

Keyword	Substitution Value
<OPSYS>	operating system according to function module FILE_GET_NAME

**Definitions of Platform-independent File Names**

<INSTANCE>	instance of R/3-system
<SYSID>	name of R/3-system according to system field SY-SYSID.
<DBSYS>	database system according to system field SY-DBSYS
<SAPRL>	R/3-Release according to system field SY-SAPRL
<HOST>	host name according to system field SY-HOST
<CLIENT>	client according to system field SY-MANDT
<LANGUAGE>	logon language according to system field SY-LANGU
<DATE>	date according to system field SY-DATUM
<YEAR>	year according to system field SY-DATUM, 4-character
<SYEAR>	year according to system field SY-DATUM, 2-character
<MONTH>	month according to system field SY-DATUM
<DAY>	day according to system field SY-DATUM
<WEEKDAY>	week day according to system field SY-FDAYW
<TIME>	time according to system field SY-UZEIT
<STIME>	hour and minute according to system field SY-UZEIT
<HOUR>	hour according to system field SY-UZEIT
<MINUTE>	minute according to system field SY-UZEIT
<SECOND>	second according to system field SY-UZEIT
<PARAM_1>	value of Parameter 1 in function module FILE_GET_NAME
<PARAM_2>	value of Parameter 2 in function module FILE_GET_NAME
<P=name>	value of profile parameter of current system
<V=name>	value of variable as defined in variable table
<F=name>	value of export parameter OUTPUT of a function module



All physical paths must contain the keyword <FILENAME> as a placeholder for the file name.

Inclusion of these parameters in physical file names and paths helps to both differentiate and standardize the assignment of file names. The keyword <TIME>, for instance, can be useful when a file needs to be stored several times in a row within a short time interval. Apart from the system fields, the following keywords, in particular, give you considerable flexibility in assigning file names:

- <PARAM\_1> and <PARAM\_2> are replaced by values that are passed explicitly to the function module FILE\_GET\_NAME in your program.
- <P=name> is replaced by values of profile parameters of the current system. To get the list of profile parameters and their values, start report RSPARAM.

### Definitions of Platform-independent File Names

- <V=name> is replaced by values of variables from the customizing tables for platform-independent file names.
- <F=name> is replaced by values that are returned by function modules. The names of these function modules must have the prefix "FILENAME\_EXIT\_". Note that in the keyword such a function module is addressed only with the part of its name that follows this prefix. For example, when the function module FILENAME\_EXIT\_EXAMPLE is used, the appropriate keyword would read <F=EXAMPLE>. The function module used must have the export parameter OUTPUT and no reference type must be specified for this parameter. Import parameters must have default values. Table parameters are not supported.

## The Function Module FILE\_GET\_NAME

## The Function Module FILE\_GET\_NAME

Platform-independent file names are used in applications programs by the function module FILE\_GET\_NAME. For a given logical file name, the function module generates the corresponding platform-specific file name at runtime, based on definitions stored in customizing tables for converting platform-independent file names.

The following table gives an overview of its import and export parameters and of its exceptions.

Table: Interface of function module FILE\_GET\_NAME

IMPORT parameter	Function
CLIENT	Logical file names can be client-specific. Here you can specify the client to be used. The current client as stored in the system field SY-MANDT is used as default.
LOGICAL_FILENAME	Here you specify the logical filename. (Uppercase letters must be used!)
OPERATING_SYSTEM	Here you can specify the operating system for which to generate the appropriate file name. The application server's operating system as stored in the system field SY-OSYS is used as default.
PARAMETER_1 PARAMETER_2	Here you can specify values that substitute the placeholders <PARAM_1> and <PARAM_2> in physical file names and paths.
USE_PRESENTATION_SERVER	Specifies that the presentation server's operating system be used as the basis for generating a platform-specific file name.
WITH_FILE_EXTENSION	Specifies that the logical file name's data format be used as filename extension.
USE_BUFFER	Specifies that the customizing tables for converting platform-independent file names be buffered in main memory.

EXPORT parameter	Function
EMERGENCY_FLAG	If the returned value is not SPACE, then no physical path has been found for the logical filename under the current operating system. In this case the path specified in the profile parameter DIR_GLOBAL will be used as physical path.
FILE_FORMAT	Returns the data format defined for the logical file name. You can use this parameter to decide in which mode to open the file. It is also required as a parameter for DOWNLOAD of files to the presentation server.
FILE_NAME	Returns the fully instantiated platform-specific file name and path.

Exceptions	Function
FILE_NOT_FOUND	Raised if logical file name is not defined.



The Function Module FILE\_GET\_NAME

OTHERS	Raised if other errors occur.
--------	-------------------------------



If the function module cannot find a physical path for the current operating system (see parameter EMERGENCY\_FLAG), this may have various causes:

- the operating system is not defined in the customizing tables
- the operating system is not assigned to a syntax group
- no physical path is assigned to the logical path for the relevant syntax group
- no logical path is assigned to the logical file name.



Assume that in the customizing tables for platform-independent file names the following definitions exist for the logical file name DATA\_FILE and the logical path DATA\_PATH:

DATA_FILE	<i>phys. file:</i>	file<PARAM_1>
	<i>data format:</i>	BIN
	<i>logical path:</i>	DATA_PATH
DATA_PATH	<i>syntax group:</i>	UNIX <i>phys. path:</i> /tmp/<FILENAME>
	<i>syntax group:</i>	DOS <i>phys. path:</i> c:\tmp\<FILENAME>

Assume also that the application server's operating system has been assigned to syntax group UNIX while the presentation server's operating system has been assigned to syntax group DOS.

The following two calls of the function module will then return the respective values.

```
CALL FUNCTION 'FILE_GET_NAME'

  EXPORTING
    LOGICAL_FILENAME      = 'DATA_FILE'
    PARAMETER_1           = '01'

  IMPORTING
    EMERGENCY_FLAG        = FLAG
    FILE_FORMAT            = FORMAT
    FILE_NAME              = FNAME

  EXCEPTIONS
    FILE_NOT_FOUND        = 1
    OTHERS                 = 2.
```

Returned values:

```
FLAG:
FORMAT:  BIN
FNAME:  /tmp/file01
```

---

The Function Module FILE\_GET\_NAME

```
CALL FUNCTION 'FILE_GET_NAME'

  EXPORTING
    LOGICAL_FILENAME      = 'DATA_FILE'
    USE_PRESENTATION_SERVER = X
    WITH_FILE_EXTENSION   = X

  IMPORTING
    EMERGENCY_FLAG        = FLAG
    FILE_FORMAT            = FORMAT
    FILE_NAME              = FNAME

  EXCEPTIONS
    FILE_NOT_FOUND        = 1
    OTHERS                 = 2.
```

Returned values:

```
FLAG:
FORMAT:  BIN
FNAME:  c:\tmp\FILE.BIN
```

## Using Platform-independent File Names in Programs

1. Make sure that the customizing tables contain definitions for the logical file name you want to use, and that these definitions produce the intended file name conversion. Use transaction FILE to inspect existing definitions or to specify new ones. (For details refer to the implementation guide in section *Basis Components* → *System Administration* → [Platform-independent File Names \[Page 83\]](#).)
2. Make sure the physical paths referred to in these definitions do actually exist in the file system at runtime. If necessary, create the respective directories or consult your system administrator.
3. Test the file name conversion by calling the function module FILE\_GET\_NAME, using transaction SE37.
4. Include a call of the function module in your program. (In the ABAP/4 editor you can do this with function *Edit* → *Insert statement...*)

For more information on storing files on the application server and on the presentation server please refer to *Working with Files* in the *ABAP/4 User's Guide*.



Problems with storing files may sometimes be due to a mismatch between the paths defined for platform-independent file names and the file system. Generation of a valid platform-specific file name by the function module FILE\_GET\_NAME is not sufficient; the path generated must also exist in the file system at runtime.

---

**Reference****Reference**

- **FILE\_GET\_NAME**

With this function module you can generate a platform-specific file name for a platform-independent logical file name in your application program at runtime.

## Number ranges

This section explains how you can use the automatic number assignment in your applications.

[Overview \[Page 94\]](#)

[Concept \[Page 95\]](#)

[Number range object types \[Page 97\]](#)

### Procedure

[Procedure \[Page 100\]](#)

[Determine the number range object type \[Page 101\]](#)

[Maintain number range object \[Page 102\]](#)

[Function module calls \[Page 105\]](#)

### References

[Number range and group maintenance dialogs \[Page 107\]](#)

[Number range and group read and maintain services \[Page 109\]](#)

[Number range object read and maintain services \[Page 111\]](#)

[Number assignment and check \[Page 112\]](#)

[Utilities \[Page 113\]](#)

---

**Overview**

## Overview

It is often necessary to directly access individual records in a data structure. This is done using unique keys. Number ranges are used to assign numbers to individual database records for a commercial object, to complete the key. Such numbers are e.g. order numbers or material master numbers.

These numbers provide, apart from unique identification of a data record, the possibility of encoding differentiating information for an object. One could tell from the number e.g., to which material type a material belongs.

The R/3 number range management also monitors the number status, so that numbers which have already been issued are not re-issued.

All dialogs, database accesses or other activities which are necessary for the maintenance of number range objects and number ranges and number allocation in user developments, can be performed using SAP function modules.

## Concept

A commercial object, for which part of the key is to be generated via number ranges, is defined as a **number range object** in the SAP system. If this commercial object contains sub-objects, e.g. company codes or controlling areas, this differentiation can also be made in the number ranges. This happens by specifying a field for the sub-object when defining the number range objects. (Example: company code as sub-object of documents)

The number range interval within a commercial object and sub-object never overlap. The number range intervals in various sub-objects of a commercial object can overlap.

A number range interval is assigned to a commercial object via the number range number. This assignment is usually saved in a table belonging to the commercial object, the group table. The field *Element* must be all or part of this table's key. Elements which refer to the same number range interval form a group. You can decide whether you want to make this assignment possible for the user during the number range object maintenance (via the assignment of elements to groups) or whether you want to program it yourself.



When a new material master is created, the material type should determine from which number range interval a number to complete the material master key should be assigned. The commercial object is the material master, the group table the material type table, with the element field material type as key field. The number range numbers for the various element values (material types) are saved in this table. Material types are e.g. semi-finished or finished products.

A **number range** contains a number range interval with a defined character set. The **number range interval** consists of numeric or alpha-numeric characters (only for external number ranges) and is delimited by the fields *From number* and *To number*. Either one interval, or several if financial years are to be distinguished, is assigned to a number range.

The **number range number** identifies a number range for the system and makes system-internal access to the number range interval possible. It can be numerical or alpha-numeric. This number is generally assigned system-internally. If you do not need grouping or if you want to program the group table maintenance for the grouping yourself, you must enter the number range number during interval maintenance yourself.

If financial years are to be distinguished in the number assignment, there can be several intervals. Separate intervals are then specified for each financial year. Number ranges can be **external** (number to be assigned manually by the user) or **internal** (number assigned automatically by the system).

A commercial object can either have only one number range (external **or** internal) or two number ranges (external **and** internal).

The various distinctions between commercial objects gives rise to eight [Number range object types \[Page 97\]](#).

## Element and group

Element is the field in the group table according to whose value a commercial object can be grouped. The grouping is done by number range assignment.

## Concept

Element values to which the same number ranges are assigned constitute a group. For the material master, for example, the groups are managed in the material type table. The groups can be maintained via the standard maintenance dialog for number range intervals. If you do not want to do this, you must program it yourself.

Groups can be either dependent or independent of sub-objects. This depends on whether the sub-object is a group table field.

The following table shows as an example the grouping of material types and the associated number range assignment.



**Grouping material types**

Group	Material type	Internal no. range	External no.range
Group 1	<b>Finished</b>	01	02
Group 1	<b>Semi-finished</b>	01	02
Group 2	<b>Raw material</b>	03	04

The material types **Finished** and **Semi-finished** form one group, and the material type **Raw material** another.

## Until financial year

Fixed time periods (financial years) are assigned to number range intervals within a number range with a year value. These intervals can overlap within a number range. In this case the financial year or until financial year, as well as the assigned number, must be part of the application table key.



## Number range object types



The description "with group" means that the assignment of number ranges to elements, i.e. the grouping, should be done via the standard maintenance dialog.

- Objects without sub-objects
  - without group**
    - (1) one, two or several number ranges
  - with group**
    - (2) one number range, external **or** internal, per group
    - (3) two number ranges, external **and** internal, per group
- Objects with sub-objects
  - without group**
    - (4) one, two or several number ranges
  - with group**, independent of sub-object
    - (5) one number range, external **or** internal, per group
    - (6) two number ranges, external **and** internal, per group
  - with group**, dependent on sub-object
    - (7) one number range, external **or** internal, per group
    - (8) two number ranges, external **and** internal, per group

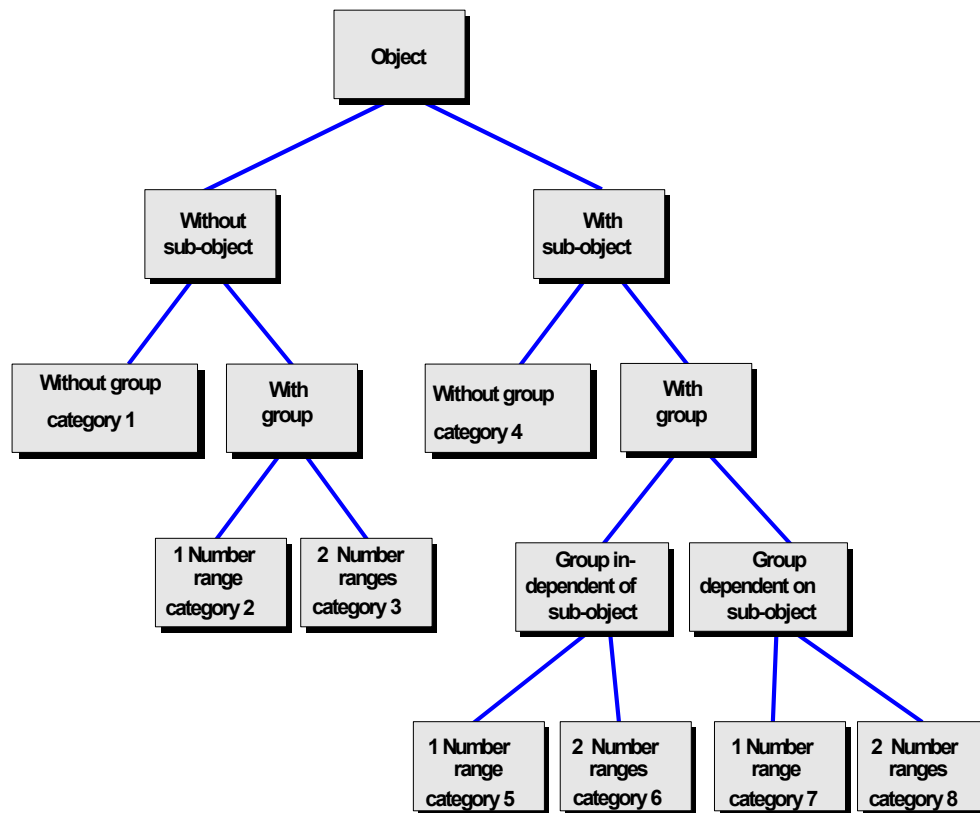
All eight object types can also be distinguished by until financial years.

The definition of an object controls the number range maintenance dialog. If you, e.g. specify a group table, the assignment of number ranges to element values in the group table can be carried out by the user in the standard maintenance dialog.

The following illustration provides an overview of the object types.

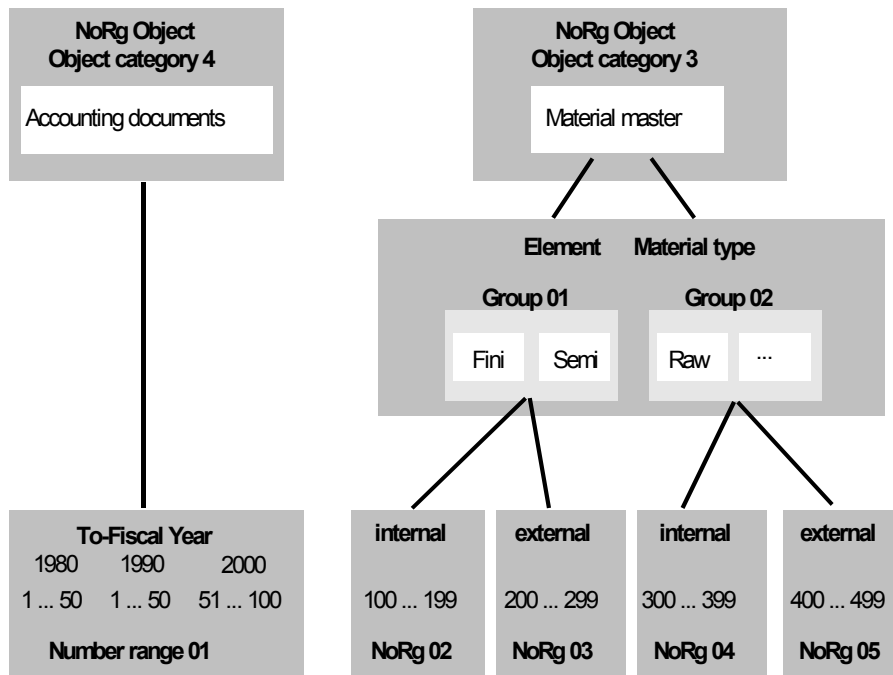
## Number range object types

## The eight number range object types



The following illustration shows, using two examples, the relationships between the concepts described in the previous sections.

Examples of number range object types 1 and 3



Example of  
accounting document table

...	001	0037	1980	GI	...
...	001	0037	1990	GR	...

Example of  
material master table

...	00100	FINI	CLOCK RADIO	...
...	00101	FINI	TOASTER	...
...	00400	RAW	IRON PIN	...
...	00200	SEMI	TRANSISTOR	...

---

**Procedure**

## Procedure

If you want to use the SAP number range functionality in your application, proceed as follows:

1. Determine which type the new number range object is to have, and create the definition.
2. Maintain the number range intervals for the new object, or have them maintained by the end users.
3. Use the [Number assignment and check \[Page 112\]](#) function modules in your application program.

## Determine the number range object type

To determine the type of number range to be used, you must clarify the following points:

- How many number ranges are required: 1, 2 or more?
- Are the number ranges dependent on a sub-object (company code, plant, controlling area, etc.)?
- Is a group to be formed (e.g. by material type)?
- If so, is the group dependent on the sub-object?
- Is the number range to depend on the financial year?

From the answers to these questions, using the illustration “Eight number range object types”, you can uniquely determine the type.

---

**Maintain number range object**

## Maintain number range object

1. Call the number range transaction (*Tools* → *ABAP/4 Workbench* → *Development* → *Other Tools* → Number ranges).
2. Enter an object name and choose *Create*. An input window appears for the development class in which you want to save the number range object.
3. Enter a development class, and choose *Save*. The object definition fields window now appears.
4. To define the number range object, enter the following fields:
  - *Short text*  
Object short text (length 20), number range maintenance dialog explanations
  - *Long text*  
Object long text (length 60), number range maintenance dialog explanations
  - *Number length domains*  
The domains determine the lengths of the numbers to be issued. They must be of type NUMC or CHAR, and have a field length of 1-20. Choose an appropriate domain from the Dictionary or create a new one.
  - *Percent warnings*  
This value specifies from what percent free interval a warning is issued when numbers are assigned. It must lie between 0.1 and 99.9.
  - *Number range transaction*  
If you enter a transaction code here, You can maintain the intervals for just this object by calling this code.

### Create sub-object

If you want differentiate a number range object, enter the data element according to whose value you want to differentiate:

- *Data element sub-object* (object types 4-8)  
This data element must exist and be active in the Dictionary and have a check table. The domains must have a field length between 1 and 6.

### Distinguishing by financial year

If the commercial object records are to be distinguished by financial year, mark the field:

- *Until financial year flag*. (all object types)

### Create groups

If the commercial objects are to be grouped by elements, you can specify, by completing the following fields, that the group table is to be maintained via the standard maintenance dialog. Otherwise you must program the assignment yourself.

- *Group table* (object types 2, 3, 5-8)  
Enter the name of the table which contains the grouping element, e.g. for the material master, the material type table. The table must exist and be active in the Dictionary and contain the number range element field as key. If the groups depend on the sub-object, the sub-object must be part of the key. Otherwise the group table must not have any other key

## Maintain number range object

fields.

A group table can only be assigned to at most one number range object.

- *Sub-object field in group table.* (object types 7 and 8)  
If the commercial object is differentiated by sub-object, and the groups are dependent on the sub-object, enter here the group table field which contains the sub-object value (object types 7 and 8). The sub-object field must be part of the key.
- *No. range element field* (object types 2, 3, 5-8)  
If the commercial object is to be grouped, enter here the group table field which contains the value according to which groups are to be formed. The number range element field must be part of the key.
- *Int./ext.no. range no. field* (object types 3, 6, 8)  
Enter here the group table fields for internal and external number ranges, if the application is to support both external and internal number assignment. A group table must be specified at the same time. The fields must have the format char (2) or num (2).
- *No. range no. field* (object types 2, 5, 7)  
Enter here the group table field for the number ranges, if the application is to support only one number range (external or internal). The field must be part of the key and have the format char (2) or num (2).  
Whether it is an external or internal number range is indicated when the interval for this number range is created.

## Group maintenance element text display

If the element text is to be displayed during group maintenance, mark this field:

- *Display element text*

You must also maintain the following element text table entries with the text entries maintenance function:

- *Element text table*
- *Language field*
- *Sub-object field*  
This field only appears in the interface when the groups are defined as being dependent on sub-objects.
- *Element field*
- *Text field*

When you have saved the input data, number range intervals can be created for the object.

## Delete number range object

To be able to delete a number range object, you must first delete the number range intervals which belong to it.

## Maintain number ranges

Have the end-user create number ranges with intervals, using the implementation guide. You can find information about this in the system administration document in the section on number ranges.

---

**Maintain number range object****Transport number range objects**

When number range objects are maintained, they are entered in a change request. When the transport is released, various consistency checks are made, to avoid the transport of objects with errors. Error messages or warnings appear in the transport log. If errors occur, the export or import is refused.



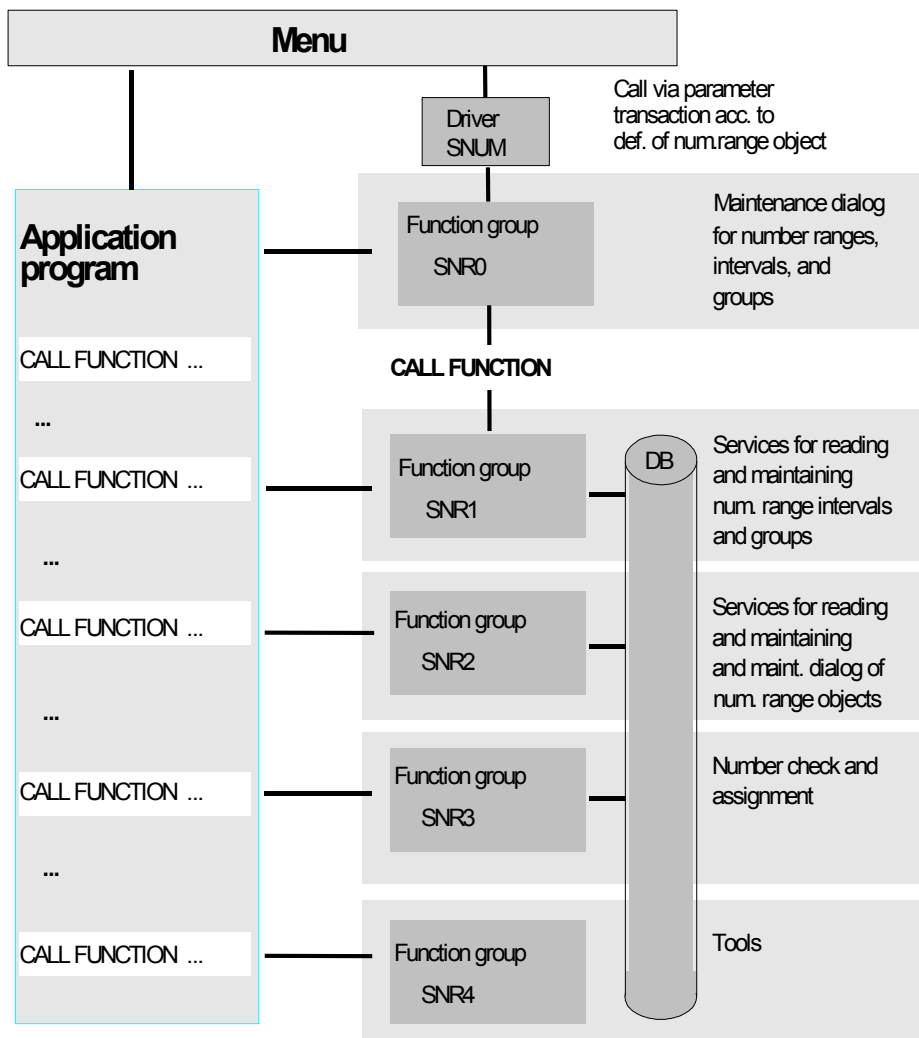
## Function module calls

The function modules distinguish between the following objects:

- **Number range objects**
- **Number range intervals and groups**

The following illustration shows the context of the function groups and their possible connections to application programs.

### Function groups overview



If you want to work with the standard number range functionality, you only need the function modules in the function group SNR3.

---

Function module calls

## Number range and group maintenance dialogs

### Function group SNR0

The function modules in this function group constitute the dialog with which number ranges, number range intervals and number range groups can be maintained.



The function modules which are labeled "\*" can only be used for the object types 2 and 3 and 5-8 (see illustration in number range object types).

- **NUMBER\_RANGE\_SHOW**  
This function module displays the groups which exist for a particular number range object, with their number range intervals.  
After return, the return code chosen by the user (*Back* or *Cancel*) is available.
- **NUMBER\_RANGE\_ELEMENTS\_SHOW \***  
This function module displays all elements, which are assigned to a number range interval.  
After return, the return code chosen by the user (*Back* or *Cancel*) is available.
- **NUMBER\_RANGE\_INTERVAL\_MAINTAIN**  
With this function module the maintenance dialog for number range intervals for a given number range object is offered. A parameter specifies the processing type. Possible processing types are:
  - Maintain intervals
  - Change number status
  - Display intervals
  - Create new groups (only for object types 2 and 3 and 5-8)

The dialog path is determined by the object type.  
After return, the return code chosen by the user (*Back* or *Cancel*) is available.
- **NUMBER\_RANGE\_GROUP\_MAINTAIN \***  
This function module is the maintenance dialog (Create, Change, Display) for number range groups for a given number range object. A processing flag determines whether the object is to be displayed only or whether it can be maintained. Groups are deleted by deleting their intervals.  
After return, the return code chosen by the user (*Back* or *Cancel*) is available.
- **NUMBER\_RANGE\_SUBOBJECT\_COPY** (only object types 4-8)  
This function module enables you to copy number range objects from groups and intervals of an existing sub-object of a given number range object to another of its existing sub-objects.  
After return, the return code chosen by the user (*Back* or *Cancel*) is available.
- **NUMBER\_RANGE\_SUBOBJECT\_GET** (only object types 4-8)  
This function module provides a dialog box in which the user can enter a sub-object for a given number range object. If the specified sub-object already exists for the number range object, it is returned in the export parameter. If it does not exist, either an exception is raised or the return code "A" for user abort is returned.



## Number range and group read and maintain services

### Function group SNR1

The function modules in this group perform all number range, number range interval and group read and maintenance database accesses.



Function modules marked with "\*" can only be used for the object types 2 and 3 and 5-8 (see illustration in number range object types).

- **NUMBER\_RANGE\_ENQUEUE**  
With this function module, you lock the number range object which is to be maintained, and its groups and intervals, for access by other users. Lock errors are returned as exceptions.
- **NUMBER\_RANGE\_DEQUEUE**  
With this function module, you unlock the number range object which has been maintained.
- **NUMBER\_RANGE\_ELEMENT\_LIST \***  
This function module gets the elements which are assigned to a particular number range interval for a number range object. The elements found are passed in a table. Errors are returned as exceptions.
- **NUMBER\_RANGE\_ELEMENT\_TEXT\_LIST \***  
With this function module you can find element texts in the specified language for a given number range object. The texts are returned in a table. Execution errors are returned as exceptions.
- **NUMBER\_RANGE\_GROUP\_LIST \***  
This function module gets information about groups and the associated group and element texts for a specified number range object. The information is put in a table. The table can be used to change the element assignment or the group text. The change request is to be passed to the function module NUMBER\_RANGE\_GROUP\_UPDATE.  
Errors are returned as exceptions.
- **NUMBER\_RANGE\_GROUP\_UPDATE \***  
With this function module, already assigned elements can be assigned to other intervals, or the assignment can be withdrawn. Group texts can also be maintained. All change requests are checked. Request errors are returned in an error table.  
  
The changes are passed in an internal table and are copied into the local memory of the function group.  
Before you call this function module, you must lock the number range object in question with NUMBER\_RANGE\_ENQUEUE, and unlock it again with NUMBER\_RANGE\_DEQUEUE after writing the changes to the database.  
To copy the contents of local memory to the database, call the function module NUMBER\_RANGE\_UPDATE\_CLOSE.
- **NUMBER\_RANGE\_INTERVAL\_LIST**  
This function module gets the existing intervals to a given number range object, and puts them in a table. The table can be passed to the function module NUMBER\_RANGE\_INTERVAL\_UPDATE to change intervals.
- **NUMBER\_RANGE\_INTERVAL\_UPDATE**  
With this function module you maintain intervals for a given number range object.

---

**Number range and group read and maintain services**

The changes are passed in an internal table, and are copied into local memory. Before you call this function module, you must lock the number range object in question with `NUMBER_RANGE_ENQUEUE`, and unlock it again with `NUMBER_RANGE_DEQUEUE`, after the changes have been written to the database. To copy the contents of local memory to the database, call the function module `NUMBER_RANGE_UPDATE_CLOSE`.

- **NUMBER\_RANGE\_OBJECT\_GET\_INFO**  
This function module gets information for a given number range object. This information is put in a table structure, which must be declared like the table structure `INROI`.
- **NUMBER\_RANGE\_SUBOBJECT\_LIST** (only object types 4-8)  
This function module gets the existing sub-objects of a given number range object, and puts them in the table passed.
- **NUMBER\_RANGE\_SUBOBJ\_GET\_INFO**  
This function module gets information about the existing sub-objects of a given number range object. This information is put in a table structure, which must be declared like the table structure `INROI`.
- **NUMBER\_RANGE\_UPDATE\_CLOSE**  
With this function module you write changes which have been made to local memory to the database, with `NUMBER_RANGE_GROUP_UPDATE` and `NUMBER_RANGE_INTERVAL_UPDATE`. After calling this function module, you should unlock the changed number range object.
- **NUMBER\_RANGE\_UPDATE\_INIT**  
With this function module, you can initialize local memory if you want to discard the changes which have not yet been copied to the database.

## Number range object read and maintain services

### Function group SNR2

The function modules in this group perform all read and maintenance accesses to number range objects in the database.

- **NUMBER\_RANGE\_OBJECT\_MAINTAIN**  
This function module provides all the screens needed to maintain a given number range object, with the possibility of branching to interval maintenance and change document display.  
An export parameter states which action the user has performed with the number range object.
- **NUMBER\_RANGE\_OBJECT\_CLOSE**  
With this function module, you write all changes to a given number range object, which were put in local memory with `NUMBER_RANGE_OBJECT_UPDATE`, to the database. If intervals are affected by the changes, they are updated. Change documents are created for all changes. A flag states whether intervals have been updated.
- **NUMBER\_RANGE\_OBJECT\_DELETE**  
With this function module, you can delete either the whole definition of a given number range object, including texts, or only the texts. The deletion is performed directly in the database. The function module provides no connection to the correction and transport system.
- **NUMBER\_RANGE\_OBJECT\_INIT**  
With this function module, you initialize local memory for a given number range object. You only need this call when you offer number range object maintenance in a user transaction, in which you want to provide the possibility of canceling changes which have not been saved.
- **NUMBER\_RANGE\_OBJECT\_LIST**  
This function module gets a list of all number range objects with their texts and attributes. The information is put in a table.  
The contents of local memory are not taken into account.
- **NUMBER\_RANGE\_OBJECT\_READ**  
This function module gets the texts and attributes of a given number range object. The records returned can be used for changes with the function modules `NUMBER_RANGE_OBJECT_UPDATE` and `NUMBER_RANGE_OBJECT_DELETE`.
- **NUMBER\_RANGE\_OBJECT\_UPDATE**  
This function module copies new number range objects or changes to existing number range objects into local memory, after error checks.  
The function module does **not** provide a connection to the correction and transport system.

---

Number assignment and check

## Number assignment and check

### Function group SNR3

The function modules in this group manage the number assignment.

- **NUMBER\_CHECK**  
You only need this function module for external number assignment. It checks whether a number range object number lies in a specified number range interval.
- **NUMBER\_GET\_INFO**  
This function module gets information for a specified number range object number range interval.
- **NUMBER\_GET\_NEXT**  
You need this function module for internal number assignment. It assigns the next free number(s) in a number range interval of a specified number range object. If the last number in the interval has been issued, the number assignment begins again with the first number in the interval.  
The return code states whether the assigned number was assigned without any problem, or whether it lies in the critical range.



## Utilities

### Function group SNR4

- **NUMBER\_RANGE\_INTERVAL\_INIT**  
With this function module you initialize all internal number range intervals of a specified number range object or sub-object.

---

**Data Archiving - ADK**

## **Data Archiving - ADK**

This section explains how you can create your own archiving programs using the Archive Development Kit (ADK).

## Overview

Data archiving compresses system data and stores it on external storage media. The archived data can then be deleted in the system. Space can then be re-used, and the archived data is stored safely.

Data archiving is recommended in the following cases:

- The database is occupied by mass data that must be stored externally
- Master data is no longer required

SAP provides the **Archive Development Kit (ADK)** for the realization of secure and efficient archiving procedures, to support and simplify the development of archiving programs.

The ADK is designed to be used in client/server architecture. The system load is shared among the database and application servers. This makes efficient use of system resources.

At the same time, data localization and the provision of access function modules support object oriented procedures, which in turn help you to keep your data consistent.

The ADK provides the interfaces, function modules, example programs and documentation you need to develop your own archiving programs.

This includes:

- Connection to the storage system
- Compression during archiving
- The possibility of archiving while the system is running
- Greater ease of use
- Network graphic for showing object dependencies
- Access to individual data objects in the archive



You can use the program RSARCH09 to copy old transaction F040 archiving procedures.

You must identify from which archiving objects you want to copy the old control data into the new archive management by entering the name of the archiving object under which the control data is to be copied into the new archive management, in the field *Reorg. variant* in the table TR01 (*System* → *Services* → *Table maintenance*).

When the control data has been successfully copied, the program RSARCH09 deletes the entry in table TR01, so the old archiving program can no longer be executed through transaction F040.

## ADK: Development Environment for Archiving Programs

The Archive Development Kit (ADK) supports the development of standardized archiving programs and their delete, read, and reload programs.

The data archiving functions include the following components:

- [Archiving Objects \[Page 122\]](#) and their methods
- [Standard Class \[Page 124\]](#) (generally valid) with its methods
- [Archiving Classes \[Page 125\]](#) (created by the user) with their methods
- [Archive Management \[Page 127\]](#)
- [Network Graphic \[Page 129\]](#)
- Authorization check (see [Archive Management \[Page 127\]](#))
- Generator for archiving programs

These components provide an environment in which you can develop your own archiving program. Sample programs are also provided.

The archiving object and standard class methods are provided as function modules. You do not need to know the technical details of the archiving procedures, as they are localized in the function modules. You can use all the ADK functions because your archiving programs communicate with these function modules.

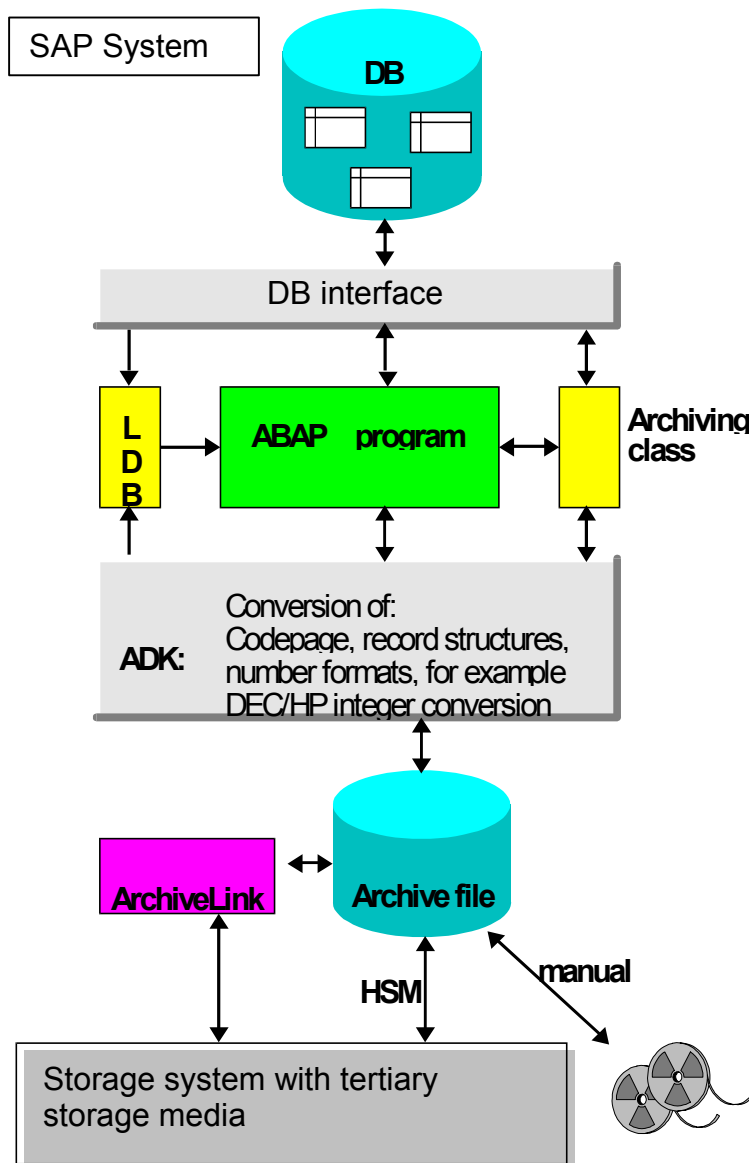
The ADK performs many of the necessary archiving activities for you. You can develop simple programs very quickly using the standard class (which already exists in the system) function modules and appropriate control parameters.

If you want to develop more complex, object-oriented archiving programs, you should use the functions provided in dedicated archiving classes (provided by an expert in the area), which contain all the information specific to the area data to be archived. This offers you the advantage that you no longer need detailed knowledge of the business object in question, because you need not concern yourself with the technical realization. You only need to call the appropriate function modules, which then perform the technical processing for you.

The development of an archiving program is based on the definition of an [Archiving Object \[Page 122\]](#), which specifies which tables comprise a business object. Archiving objects are complex objects of interdependent tables. They can contain sub-objects used repeatedly in the system, such as change documents.

## Interaction between Program, ADK, and Archive File

The following graphic illustrates the various SAP System components involved in archiving sessions.



As of Release 4.6C, the Content Management Service (CMS) is used to store archive files in an external storage system, instead of SAP ArchiveLink.

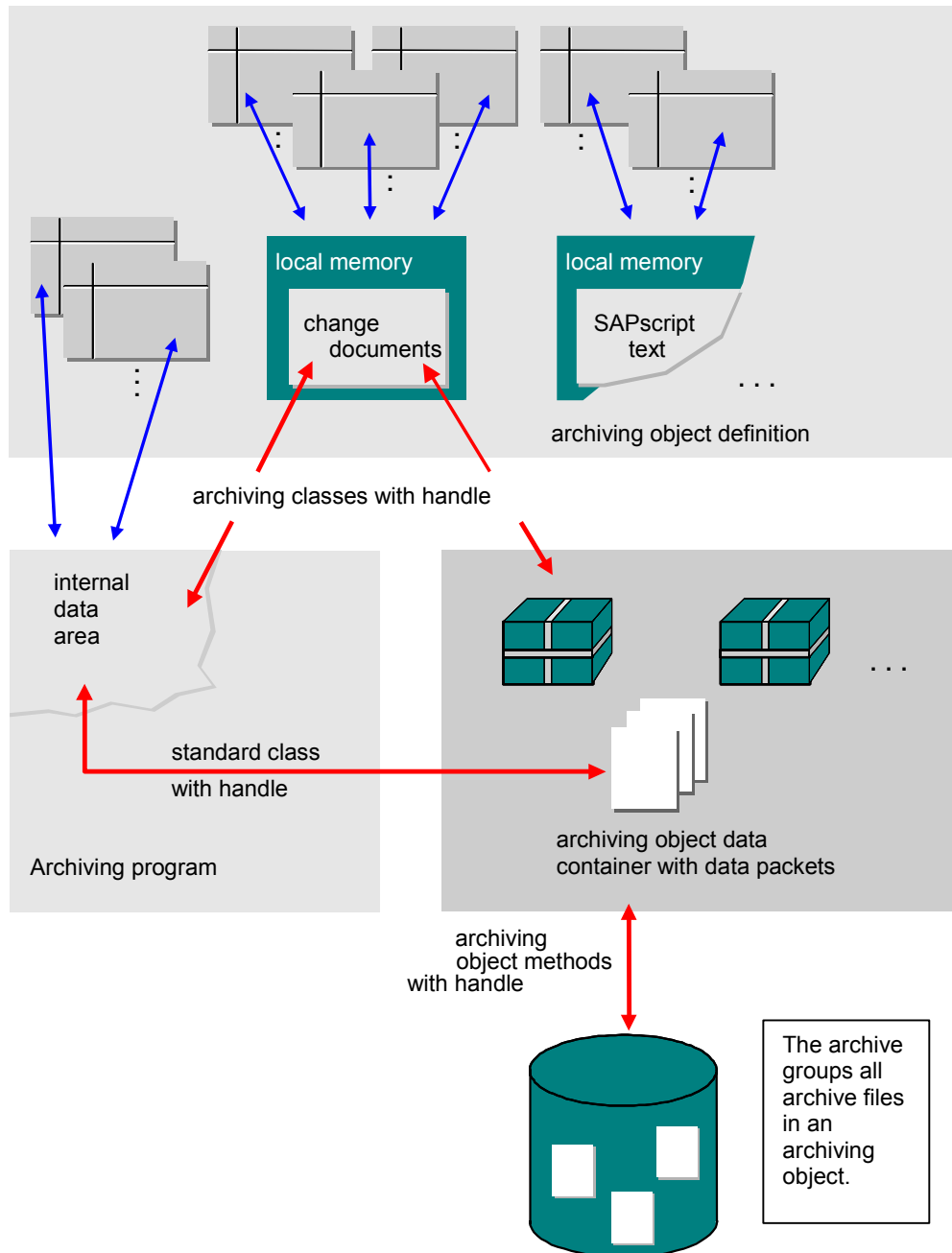
Hierarchical Storage Management (HSM) manages archive files on external storage media as though they were a system component, independently of the medium on which they are actually stored.



## Archiving using ADK

### Use

The ADK archiving concept and data flow are illustrated in the following graphic.



## Archiving using ADK

### Explanations

- **The archiving program** selects the data to be archived to the archive file from the database according to specified criteria. Records can be marked for subsequent deletion by the delete program. A single table field is the smallest recognizable entity in the program.
- The data object to be archived or read from the archive file is passed to a **logical data container**. The archiving object standard class or archiving class methods access this data container. The smallest recognizable entries when reading or writing to the data container are table entries (for archive programs) and archiving class data packets (for read programs). The data is passed between the data container and the archive file using the archiving object methods.
- The **archive** contains the archived data objects of an archiving object, and consists of several archive files. In Customizing, you set the maximum archive file size (in MB) and the maximum number of data objects that can be added to an archive file. If the maximum file size or maximum number of data objects is reached while writing to an archive file, the archive file is closed and a new file is opened. The program accesses the archive file using archiving object function modules. For these function modules, a data object is the smallest recognizable unit.

### Identifying archive files and archiving sessions

An archiving sessions consists of the archiving program and the archive files created by the program. Each archiving session has a unique ID, which is assigned by the ADK and stored in archive management along with other data, such as the sizes and names of the corresponding archive files.

If archive reads or writes are programmed, the ADK displays a dialog box at runtime that includes a list of the available archiving sessions. For a program to be run in the background, you can use this list to select the relevant archiving sessions.

At program runtime, a temporary number, the archive handle, is assigned. The archive handle is used to identify the previously selected archiving sessions (read or write programs) or the new archiving session generated (write program). This means that archived data can be read within a program and written to the archive in parallel.



## Archiving Process

### Process Flow

The full archiving process consists of at least two activities:

- **Writing the data to be archived from the database to the archive file**  
The data objects to be archived are written sequentially to the archive file. A data object consists of a record sequence according to the object definition. The data is compressed by object.



This program can generally be executed during dialog processing. The program is executed in the background by choosing *Archive* in *Archive Administration* (transaction *SARA*). The system looks for the next free background process, preferably on the database server. You select a server for archiving using *Server selection* in archiving object-specific Customizing.

- **Deleting archived data from the database**  
The program first reads the archived data in the archive file. After reading, the data is usually deleted from the database. Sometimes only a delete indicator is set and the deletion is carried out later. The program can then pass the archive files to the Content Management Service (CMS), if a storage system is connected to the CMS.  
In [archiving object-specific Customizing \[Page 135\]](#) archiving object-specific Customizing, you can specify whether the delete program is to be called automatically after the system closes an archive file.



As of Release 4.6C, the Content Management Service (CMS) is used to store archive files in an external storage system, instead of SAP ArchiveLink. This means archive files can be created wherever you specify and they no longer have to be created in the basic directory of the storage system.

Other archiving activities such as archiving analyses are also possible.



As of Release 4.6C, the Content Management Service (CMS) can be used to store archive files in an external storage system before the data contained in the files is deleted from the database during the delete phase.

## Archiving Objects

# Archiving Objects

## Definition

If the data for a business object is to be archived, you must first determine which data belongs to the object, how it is to be archived, and which processing options should be available. This definition is saved in the system as an archiving object.

The archiving object specifies the database tables from which archiving data is to be taken. Archiving classes, which are tailored to particular logical objects (for example, change documents), can be used.



An accounting document with all its items, change documents, and long texts is an archiving object.

## Use

### Archiving object methods

Each archiving object automatically has certain methods that can be used by archiving programs.

- **For all access types:**

- Open archive file for writing  
ARCHIVE\_OPEN\_FOR\_WRITE
- Open archive file for reading  
ARCHIVE\_OPEN\_FOR\_READ
- Open archive file to reload  
ARCHIVE\_OPEN\_FOR\_MOVE
- Open archive file to delete  
ARCHIVE\_OPEN\_FOR\_DELETE



These function modules return an identifying number (archive handle), with which you can access the associated archive file and its data container from the program.

- Close archive file

- **For write access:**

- Get new data object (initialize data container)  
ARCHIVE\_NEW\_OBJECT
- Write data object from data container to archive file  
ARCHIVE\_SAVE\_OBJECT



If archiving classes are used, this is the point at which the archive class data is read from the database and written to the data container.

- **For read access (incl. reload):**

- Read next data object from the archive into the data container  
ARCHIVE\_GET\_NEXT\_OBJECT
- **For all access types:**
  - Close archive file(s)  
ARCHIVE\_CLOSE\_FILE

How data is transferred from the database to the data container depends on whether there are archiving classes for the relevant tables:

- **Archiving classes exist**  
The data is passed by the archiving class function modules.
- **No archiving classes exist**  
The data must first be copied into the program area using user program logic, and can then be written to the data container using the standard class function modules.

---

**Standard Class**

## Standard Class

### Definition

The standard class is a set of function modules available to all archiving objects. These function modules are used for general access to archived data.

The function modules can read and write data records and execute simple conversions during reading, such as changing the code page, the number format, or the record layout. Archived data cannot be deleted from the database or reloaded using the standard class. If you use the standard class for an archiving object, you must program these actions yourself.

This standard class is for general use, it must therefore be told which archiving object it is dealing with. This information is passed in control parameters.

### Standard class tasks

- Write data records for the data container
- Read data from the data container
- Automatic conversion while reading
  - Code page
  - Number format, for example, DEC/HP integer conversion
  - Simple structure changes

### Prerequisites

The archiving object must be defined in the system in transaction AOBJ.

### When is the standard class used?

The standard class methods should always be used when no archiving class exists for the relevant table.

If the data to be archived contains sub-objects used repeatedly in the system as service functions, such as change documents, you should consider developing an archiving class.

## Archiving Classes

### Definition

Archiving classes make possible object-oriented processing of the data to be archived. They simplify access to the data of business entities for archiving.

An archiving class is a set of function modules and subprograms for a business object, which is usually used repeatedly as a service function in the system and is archived with the data where it is used, rather than independently, for example, SAPscript texts, change documents, or purchase requisitions. It collects the data and passes it as a data packet to the data container. This data packet can only be processed by the archiving class' own function modules.

Because of the localization of the data, the calling program no longer needs to know the particular data structures and hierarchies. By using archiving classes, you can easily archive data of which you have no detailed knowledge.

The development of archiving classes involves an initial additional work load, but this is recovered in simplified program development and consistent data. It is worth making sure that archiving classes are created for the data you have to archive.



If tables to be archived overlap between two archiving objects, you do not necessarily have to develop an archiving class. The tables can be checked to determine which data is to be deleted. Developing archiving classes is complicated and therefore you should check each time whether a class really needs to be developed, or whether you can implement checks and processes for overlapping archiving objects.

### Use of and advantages of using archiving classes

Provide archiving classes if complex logical objects used for service functions during archiving of application data need to be archived as well. This prevents data inconsistencies. This is especially true when a logical object is to be archived as well, but the processing logic cannot be covered by the standard class methods.

For data that represents a business object, you should create an archiving class when you can answer one of the following questions with "Yes":

- Is the data used repeatedly in the system in the same form and is it also archived in the various contexts?
- Will other applications use and archive my data structures?

Archiving objects can be created with the archiving classes if necessary. An archiving class can be used in any number of archiving objects. This can either be done statically, by specification in ADK, or dynamically in the archiving program (by calling an ADK function module).



The archiving object "FI document" data is copied into the data container with the standard class. Archiving classes are used to transfer long text and change document data between the database and the data container.

The use of archiving classes has the following advantages:

## Archiving Classes

- **Re-use** without extra work
- **Simple extension of the archiving object** using new classes
- **Dynamic inclusion in archiving objects**
- **Consideration of the object itself**, not its physical data structure
- **Data integrity**
- **Modularity** of the data and methods
- **Central implementation** of archiving functions

### Archiving classes are used to:

- Communicate with the archiving object function modules
- Get complex data
- Prepare data packets for the data container
- Read archived data packets from the data container
- Delete archived data from the database
- Write archived data back to the database

### Prerequisites for using archiving classes

Detailed knowledge of the data structure is necessary for the development of archiving classes.

For an existing archiving class to be used, it must be assigned to the archiving object in question. There are two possibilities:

- **Static connection**  
A permanent assignment is made using a table entry in ADK (transaction `AOBJ`). The function modules are connected to the archiving object when the archive file is opened. (This is the normal type of connection.)
- **Dynamic connection**  
The archiving class is connected to the archiving program by a call of function module `ARCHIVE_REGISTER_CLASS`. Use this procedure when the system only determines at runtime (for example, by user input) which sub-objects of the archiving objects are to be archived. A dynamic connection must be used to include a class in another class.

For more information, see [Using Archiving Classes \[Page 154\]](#).

## Archive Administration

### Use

Programs that process archived or to-be-archived data, generally have a long runtime. Therefore, they must always run in the background.

The ADK provides archive administration with which you can generate background jobs for all archiving programs (archive, delete, analyze, and reload). Call archive administration using transaction `SARA`. On the initial screen, enter the name of the archiving object and choose *Enter*. A list of the available actions for the selection appears.

The *Management* action is always available. You use this to display all archiving sessions that have been executed for this object. The list includes the following information about each archiving session:

- Date and time
- User that executed the session
- Archive file name and path
- Status (whether deleted or moved to storage system)
- Archive file size and number of data objects in archive file

### Creating notes

You can create notes about each archiving session, for example, about the archive file location or other information.

### Authorization check

Access to the archiving object programs is controlled using authorization object `S_ARCHIVE`. The ADK checks this authorization when an archive file is opened for one of the following actions:

- Write
- Delete
- Read
- Reload

The following authorizations can be assigned per archiving object and application (for example, FI or BC):

- All authorizations:
  - Write, read, and reload archives
  - Execute delete program
  - Change mode in archive management (notes)
- Change mode in archive management
- Read and analyze archives, display mode in archive management

Additional application-specific authorization checks may be made for database accesses.





## Network Graphic

### Definition

In the ADK, the network graphic enables you to display archiving object dependencies in the archiving process (transaction SARA: choose *Goto* → *Network graphic*). The archiving object hierarchy displayed allows you to easily determine the sequence in which the data must be archived for optimal data storage.

### Structure

Each node in the graphic represents an archiving object and includes the following information:

- Name of the archiving object
  - Name of the application
  - Short description
  - Date of the last archiving
- Color legend:
- Green  
Last archiving and delete successful
  - Yellow  
Archiving running or  
Archiving finished or  
Delete running or  
Delete terminated
  - Red  
Not yet archived or  
Archiving terminated

## Developing Archiving Programs

To develop programs for archiving application data with the ADK, proceed as follows:

1. [Define archiving objects \[Page 131\]](#)
2. [Define standard class hierarchical structure \[Page 133\]](#) (optional)
3. [Customizing settings \[Page 135\]](#) (optional)
4. [Assign archiving classes \[Page 138\]](#) (optional)
5. Program development:
  - [Develop archiving program \[Page 139\]](#)
  - [Develop delete program \[Page 145\]](#)
  - [Develop reload program \[Page 147\]](#)
  - [Develop analysis program \[Page 149\]](#) (optional)
6. [Maintain network graphic \[Page 151\]](#)

## Examples

To explain how the function modules work, the ADK includes sample programs for the archiving objects *EXAMPLE* and *BC\_SBOOK*:

- Generate archive files  
**RSARCH04** and **SBOOKA**
- Read archiving files  
**RSARCH05** and **SBOOKR**
- Read archive files for deleting data and maintain indexes  
**RSARCH06** and **SBOOKD**
- Read archive files for reloading data  
**RSARCH07** and **SBOOKL**
- Read individual data objects in the archive using an index  
**RSARCH13**

## Defining Archiving Objects

### Procedure

1. Call transaction AOBJ.
2. Choose *New entries* and enter the following data:
  - **Object name**  
Name of the archiving object
  - **Text**  
Short description
  - **Work area**  
Organizational category for assigning archive files
  - **Application component**  
Used for assigning archive files



The programs specified below must already exist in the system, because checks are carried out when an archiving object is created.

- **Write program**  
Name of the program that writes the archive files.
- **Delete program**  
Name of the program that deletes the data from the database after the archiving program. If the *Start at end* checkbox is selected, the delete programs do not start until the write programs are finished.
- **Reload program** (optional, but recommended)  
Name of the program with which the data can be loaded from the archive back into the database.
- **Preproc. prog.** (optional, only when absolutely necessary)  
Name of the program with which data is to be prepared for archiving.
- **Post-processing program** (optional, only when absolutely necessary)  
Name of the program with which data is to be processed after it has been archived. If, for example, the data is only marked for deletion in the delete program, the actual deletion can be executed in the post-processing program.
- **Prog. generated** indicator (program generated)  
The program is generated.
- **Cross-client** indicator  
Archiving is client-independent.
- **End dialog** indicator  
Archiving is not to be executed in dialog mode.
- **Build idx prg** and **Prg. for IdxDel** indicators  
Name of the programs for building and deleting indexes.
- **ArchiveSelectnLive** indicator

## Defining Archiving Objects

The *Archive selection* pushbutton appears in the management transaction for building and removing indexes.

- **Build index** indicator

An index can be created for this archiving object. For more information, see [Creating ADK Indexes and Using Them to Access Archive \[Page 152\]](#).

The actual index creation can be controlled by a [Customizing \[Page 135\]](#) entry.

- **“Invalid” indicator fixed** indicator

If this is selected, the “Invalid” indicator for archiving sessions cannot be reset in archive management after it is set.

- **No reload files** indicator

If this field is selected, no new archiving session is generated when reloading archiving sessions. The reload program is not authorized to call the function module ARCHIVE\_SAVE\_OBJECT.

- **Archive hints**

Name of a document containing help text for the object-specific archiving program.

- **Delete hints**

Name of a document containing help text for the object-specific delete program.

- **Reload hints**

Name of a document containing help text for the object-specific reload program.

- **Prep.Prog. hints**

If a preprocessing program is planned, you can enter the document that contains the help text for the preprocessing program.

- **Post-proc. hints**

If a post-processing program is planned, you can enter the document that contains the help text for the post-processing program.

- **Read info**

Name of a document containing the help text for object-specific read programs.



The help function documents are created using the documentation maintenance transaction (SE61).

3. Save your entries and return to the initial screen of transaction AOBJ.

## Result

Your new archiving object is included in the list of objects in the system. You can now create additional information about your archiving object by selecting the line and choosing one of the actions under *Navigation*.

## Defining Standard Class Hierarchical Structure

In the archiving hierarchy, interdependent tables are described as segments in a structure. Their dependencies must be represented in the structure of the standard class. In archiving classes program logic creates the structure.

### Procedure

To define the structure of the tables in your archiving object, proceed as follows:

1. Select the archiving object in the list on the initial screen of transaction AOBJ, and choose *Structure Definition* under *Dialog Structure*. The hierarchy maintenance screen appears.
2. Choose *New Entries*, and enter the following data:
  - **Record number**  
Organizational numbering without any functional significance. We recommend you use sequential values for transparency.
  - **Parent segment**  
Structure name of the superior segment (this field is empty for the top segment)
  - **Segment**  
Structure name
  - **Structure**  
Name of the structure, if you are working with a logical database and the structure name under *Segment* is a pseudonym for a real structure.
  - **Do not delete** indicator  
Identifies the segments not to be deleted. If the delete program is generated, a segment so marked is not deleted. This information is also used in the [Display of Tables and Archiving Objects \[Ext.\]](#) (transaction DB15).

Save your entries, and return to the initial screen of transaction AOBJ.

---

**Tables From Which You Only Delete Entries**

## Tables From Which You Only Delete Entries

Data archiving includes tables from which entries are only deleted and not archived. There are the following types of tables:

- Tables whose entries are deleted but not archived:  
This can include tables whose entries can be rebuilt at any time using documents in the system (for example, index or match code tables).
- Archived structures and views:  
This can include tables used in structures and views. The structure and views are not themselves deleted.

To assign tables from which you only delete entries to an archiving object, proceed as follows:

1. Select the relevant archiving object on the initial screen of transaction `AOBJ`. Under *Dialog Structure*, choose *Tables From Which You Only Delete Entries*. The hierarchy maintenance screen appears.
2. Choose *New entries* and enter the relevant tables in the *Table name* field.
3. Save your entries and return to the initial screen of transaction `AOBJ`.

## Archiving Object-Specific Customizing

### Procedure

To maintain the archiving control parameters for an archiving object, proceed as follows:

Select the archiving object in the list on the initial screen of transaction AOBJ, and choose *Customizing settings* under *Dialog Structure*.



Variants must be maintained for each client. All other values are cross-client.

You can set the following parameters:

### Logical file name

Logical name for the archive file used for platform-independent data storage.



You must have maintained the logical file name using transaction FILE (*Logical File Path Definition*).

### Server selection

You use this to select the servers where the archiving programs are to run in the background.

### Archive file size

If one of the following parameters is exceeded during writing of an archive file, the system automatically creates a new archive file.

- *Maximum size in MB*
- *Max. number of data objects*



The absolute maximum size of an archive file is 2 GB.

### Settings for delete program

The archived data records in the database are usually deleted. Under certain circumstances, however, this deletion is only logical, that is, when the *Delete* indicator is set. The physical deletion occurs later in a postprocessing program.

#### *Commit counter*

Number of data objects after which the delete program sends **COMMIT** to the database.

#### *Test session variant*

Specify the delete program variant for the test session.

#### *Production session variant*

Specify the delete program variant for the production system.

## Archiving Object-Specific Customizing



Use the *Variant* pushbutton to maintain variants for the delete program.



The program variants specified under *Test session* and *Production session* are client-dependent. They must be created in each client under the same name, with the same parameters.

- *Deletion jobs*

- *Not scheduled*

The delete jobs are not automatically started.

- *Start automatic.*

The delete jobs are started automatically by the write program immediately after an archive file is finished. If files are to be stored in a storage system, you can specify whether the data is to be deleted from the database before or after the files are successfully stored.

- *After event*

The delete jobs are started by events. The name of the event must be entered in the *Event* field. If the event requires a parameter be set, enter the parameter in the *Parameter* field.

- *Build index*

Controls whether the archived data objects are to be added to the index.



The *Build index* check box is only displayed if the indicator is set in transaction AOBJ (Definition of Archiving Objects).

## Postprocessing program settings

If you have selected an archiving object that uses a postprocessing program, you can maintain the following parameters:

- *Production session variant*

Use this to set a production variant of the postprocessing program.



Use the *Variant* pushbutton to maintain the production session variants.

- *Start automatic.*

Controls whether the postprocessing program is automatically called after the delete is finished.





The postprocessing program then only starts when the last delete program of the archiving session has completed and no archive file has the status *Archiving completed*, *Archiving running*, or *Delete running*.

## Data storage in storage system

If a storage system is connected through the Content Management Service and you want to store files in a storage system, you can set the following:

- **Content Repository**

Name of the Content Repository

- **Start automatic.**

Controls whether an archive file is automatically stored in the Content Repository.

- **Sequence**

The time point at which the files are moved to the Content Repository is determined by how the archive files are processed after they are created:

- **Delete before storage**

The files are moved to the Content Repository after the delete program has processed the file in production mode. If the delete program is running in test mode, the files are not automatically stored after the delete program has finished.

- **Store before delete**

The files are moved to the Content Repository after the write program has written the archive file and before the delete program starts. This means the delete program cannot process the files until they are successfully stored.

The *Delete prog. reads from stor.system* check box specifies whether the delete program reads the data to be deleted from the storage system or the file system.

If the delete program option *Start automatic.* is selected, the delete program is called after file storage is completed. In this case, it makes no difference whether the delete program is running in test mode or production mode. Similarly, if the *After event* option is selected, the delete jobs are scheduled and automatically started after the specified event occurs.

Save your entries, and return to the initial screen of transaction AOBJ.

---

Assigning Archiving Classes

## Assigning Archiving Classes

If you want to use existing archiving classes or have developed an archiving class, you must assign this class to the relevant archiving object, so that the ADK can find the required interfaces during archiving. You can either specify this statically, if archiving is always to be structured in the same way, or dynamically by calling function module `ARCHIVE_REGISTER_CLASS` in your archiving program, if you want use input parameters to control, for example, whether particular sub-objects are to be archived or not.

### Procedure

To specify static assignment, proceed as follows:

1. Select the archiving object in the list on the initial screen of transaction AOBJ, and choose *Archiving classes used* under *Dialog Structure*. The assignment maintenance screen appears.
2. Choose *New entries*, and enter the desired archiving classes. You can use the F4 help, which lists the existing archiving classes in the system.
3. Save your entries and go back to the initial screen of transaction AOBJ.

## Developing Archiving Programs

### Purpose

Archiving programs write the data to an archiving object in an archive file. These programs can be adjusted to individual requirements. The data objects to be archived can either be called directly in the archiving program or through a logical database.

All archiving programs have one thing in common. They use the ADK function modules (function group ARCH), to save their data by objects in archive files. All archive file access methods are covered by these function modules (see also [Standard Archiving Programs \[Page 142\]](#)).

For the development of archiving programs it is also advantageous if archiving classes are defined. These contain the program logic for the data transfer between the database and the data container (see also [Archiving Using Archiving Classes \[Page 143\]](#)). If no archiving classes exist, the archiving program must get the data from the database. The data is passed to the data container through the ADK standard class.

### Guidelines

- To avoid loss of data, the archiving program must not delete any data in the database itself. (Deletion is performed by an independent delete program.)
- The archiving program can change the database, to set an archiving indicator, for example. This should only happen when absolutely necessary, as every change increases the database load and the archiving runtime.
- As archiving programs run in online operation, the data selection should not severely affect performance of the R/3 System.
- Every time the ADK creates a new archive file a COMMIT (not the ABAP command COMMIT WORK) is sent to the database.



Extensive documentation is available for all ADK function modules

## Process Flow

### Function modules call sequence

#### 1. Open archiving – ARCHIVE\_OPEN\_FOR\_WRITE

This archiving object function module is called **once only for each archiving session (per archiving object)** and returns a unique handle which is required for all further archive operations. The function module performs the following tasks for the archiving object passed:

- Controls whether an archive file is to be created
- Controls whether the delete program should be called in test mode
- Creates a header entry in archive management
- Includes the static archiving classes
- Opens the first archive file

## Developing Archiving Programs

- Writes the header entry in the first archive file (for example, information about the ABAP Dictionary (Nametab) tables involved)

### 2. Dynamically include archiving classes – **ARCHIVE\_REGISTER\_CLASS**

If you want to use archiving classes, and they are to be included dynamically, you must call this function module and pass the handle for each archiving class. The function module writes the information to the archive file (for use by the succeeding programs).



Must be called directly after `ARCHIVE_OPEN_FOR_WRITE`.

**The following steps (3 - 5) must be called in a loop for all data objects to be archived.**

### 3. Get new data object – **ARCHIVE\_NEW\_OBJECT**

You call this function module for each data object. Only then can you pass data to archiving object function modules. The function module performs the following tasks:

- Initializes data container
- Calls the archiving class initialization subprograms
- Passes ADK index entry value through `OBJECT_ID` (if required)  
The composition of this character string value must not be changed, and each value passed must be unique (see also [Creating an ADK index and Using it to Access Individual Data Objects in the Archive \[Page 152\]](#)).



If you have saved a data object in the archive file using function module `ARCHIVE_SAVE_OBJECT` and you want to archive additional data, you must call this function module again.

### 4. Build the data object

Call either the standard class function module `ARCHIVE_PUT_RECORD` or archiving class function modules. These are subject to a naming convention, in which “class” represents the archiving class name: `class_ARCHIVE_OBJECT`.

You must decide whether the data passed should be deleted by the delete program or not. The archiving class function modules provide the parameter `OBJECT_DELETE_FLAG` for this purpose.

The function module `ARCHIVE_PUT_RECORD` provides you with this function through the parameter `RECORDS_FLAGS`.

The archiving class function modules select the data for you and optimize database access. First they collect the requests and only then access the database when the function module `ARCHIVE_SAVE_OBJECT` is called. These function modules recognize the data object to which the data belongs through the handle that must be passed to the function module interfaces.



If you have already passed data you do not actually want to archive, call function module `ARCHIVE_NEW_OBJECT`. The system then discards the passed standard class and archiving class data.

### 5. Store data object in archive file – **ARCHIVE\_SAVE\_OBJECT**

You must call the function module `ARCHIVE_SAVE_OBJECT` to request the actual archiving of a data object. It performs the following tasks:

- Gets the archiving class data packets
- Compresses the standard class data
- Collects the statistics data
- Updates the archive management records
- Writes the data object to the archive file (from the data container in which the records were stored by the archiving classes and the standard class)
- Closes the archive file when it reaches the specified maximum size or contains the specified maximum number of objects, and opens a new one
- Calls the delete program after an archive file has been closed
- Locks the data container, so that no more data can be written to the data container after a data object has been written to the archive file.

**End of the loop (step 5.)**

**6. End archiving I – ARCHIVE\_WRITE\_STATISTICS**

You use this function module to generate statistics about the archived data at the end of archiving. Data records passed by the standard classes (ARCHIVE\_PUT\_RECORD) are listed individually.

**7. End archiving II – ARCHIVE\_CLOSE\_FILE**

You end archiving by calling this function module. The handle passed becomes invalid and can no longer be used.



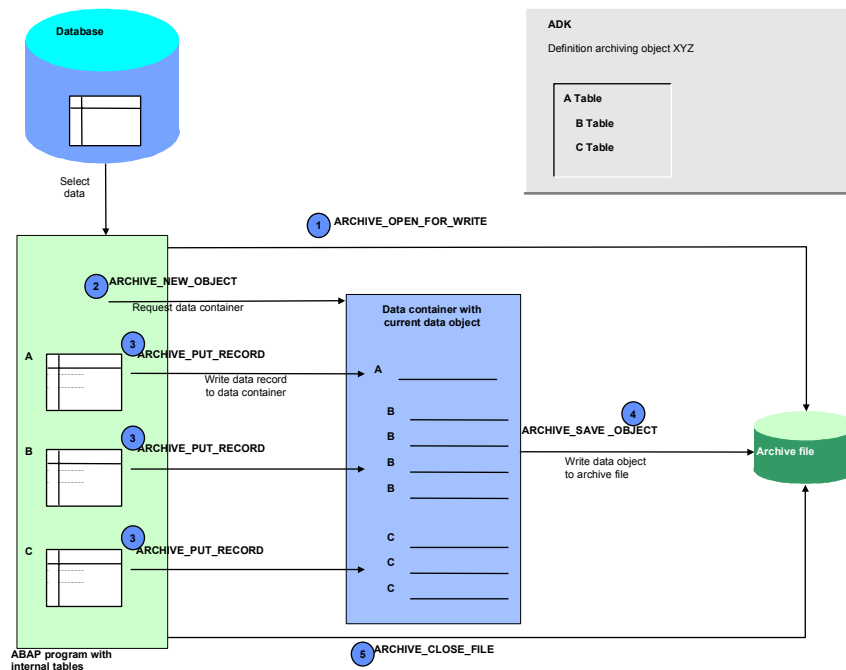
This call must not be forgotten, otherwise the last (physical) archive file to be processed is lost, and archiving is incomplete. The function module does the following:

- Updates the archive management records
- Closes the current archive file
- Releases the included archiving classes
- Discards the current handle
- Calls the delete program for the archive (if automatic deletion is specified)

## Standard Archiving Program

## Standard Archiving Program

The following simplified illustration describes the logic of an archiving program for an archiving object without archiving classes. The archiving object ABC consists of the entity table A with two dependent tables B and C.



The program opens the archive file for writing, and receives a number (archive handle) that identifies the file for all further data container and archive file accesses.

A new data container must be requested for each record in the entity table **A**. All dependent records **B** and **C** are appended to it individually, until the data object is complete.

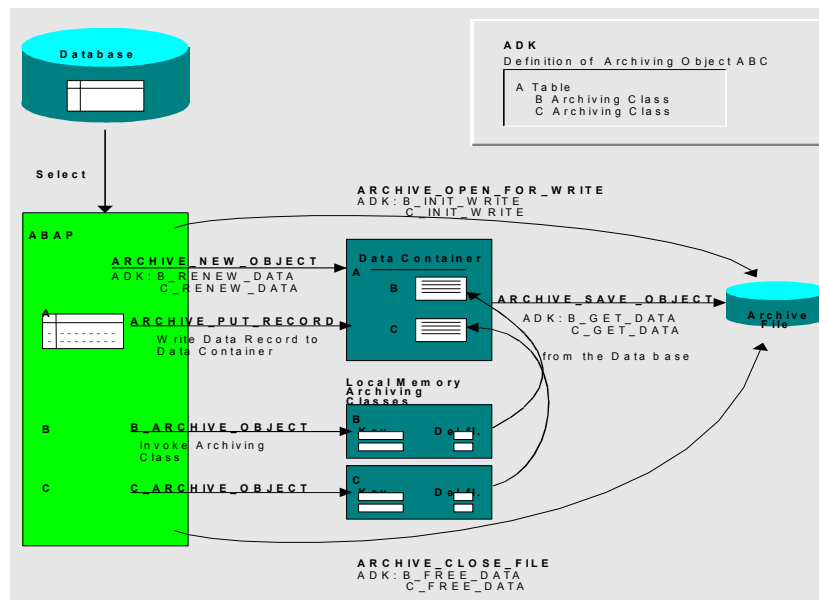
The complete data object is written to the archive file.

After all data objects to be archived have been written to the archive file, the archive file is closed.

## Archiving Using Archiving Classes

### Purpose

The following illustration shows the logic of an archiving program which processes archiving object data using archiving classes, in simplified form. The archiving object consists of the entity table A with two dependent archiving classes B and C. These archiving classes contain, in turn, n tables.



### Process Flow

When the archive file is opened, ADK calls the subprogram `*_INIT_WRITE` for archiving classes B and C. ADK determines which archiving classes must be called for the current archiving object as follows:

- Which archiving classes belong to the archiving object is determined either statically in transaction AOBJ (action *Assigning archive classes*), or dynamically in the program, before the archive is opened.
- The ADK determines the function group the required subprograms are assigned to using the assignment of function groups to archiving classes defined in transaction ACLA (*Define Archiving Classes*).

A new data container, in which the data object is constructed when writing, must first be requested for each record in table A.

The program puts a table A record in the data container and passes the key to archiving classes B and C, which save it in an internal table.

### Archiving Using Archiving Classes

The program issues the command to write the data object to the archive file. The B and C records, which depend on A, are read from the database by the archiving classes and put in the data container, and the complete data object is written to the archive file.

After all data objects to be archived have been written to the archive file, the archive file is closed.



You can also develop an archiving program using only archiving classes. In this case, only the selection logic of the data objects to be archived must be programmed in the archiving program.



## Developing Delete Programs

### Purpose

For data security reasons, the archiving program itself must not delete any data in the database. The archived data must therefore be deleted in the database by a separate delete program.

The delete program uses the function modules to read the archives and then deletes the data from the database. This ensures that no data can be lost during archiving, and makes online archiving possible.

The delete programs can be adjusted to individual requirements. The data for the data objects to be deleted can be read either directly in the delete program or through a logical database. The ADK function modules read the data by object from the archive files.

You can use control parameters in [Archive Object-Specific Customizing \[Page 135\]](#) to specify whether the delete program should be called automatically as soon as the archiving program closes an archive file, and whether the index should be updated (see also [Creating an ADK Index and Using it to Access Individual Data Objects in the Archive \[Page 152\]](#)). You can also create delete program test and production variants in archive Customizing.

A delete program always only processes one archive session file at a time. Several delete programs can run in parallel and process one archive session file each.

### Guidelines

The delete programs must determine which data is to be deleted from the database by reading the archive files. This guarantees that only data that has been legibly stored in the archives is deleted from the database.

The archiving classes' function modules must be called to delete the data. These function modules do not send COMMIT WORK and do not delete the data immediately, they call PERFORM ON COMMIT.

The delete programs must use function module ARCHIVE\_GET\_CUSTOMIZING\_DATA to get the value of the object counter, which controls after how many data objects a COMMIT WORK is called by the delete program.

## Process Flow

### Function module call sequence

#### 1. Initialize delete – ARCHIVE\_OPEN\_FOR\_DELETE

This function module is called only once, at the start of the delete program. It performs the following tasks:

- Opens the archive file to read
- Provides a handle for archive access
- Includes all archiving classes listed in the archive file
- Sets the status information in archive management
- Passes the archive files to the Content Management Service (CMS)

## Developing Delete Programs

### 2. Read next data object from archive file – **ARCHIVE\_GET\_NEXT\_OBJECT**

As for all archive read accesses, you read the next archived data object using this function module. It performs the following tasks:

- Reads an archived data objects from the archive file
- Supplies the archiving classes with data packets
- Provides the data container for the standard class



This function module must be called in a loop, until no more data objects can be provided.

### 3. Delete archived data from the database

The function module **ARCHIVE\_DELETE\_OBJECT\_DATA** deletes the data from the database for all archiving classes of the last data object to be read using **ARCHIVE\_GET\_NEXT\_OBJECT**, if the **OBJECT\_DELETE\_FLAG** was set to 'X' in the archiving program when the data was written. This function module must therefore be called in the delete program only once per data object read (**ARCHIVE\_GET\_NEXT\_OBJECT**). You must read the standard class data from the data object yourself and delete it from the database. Put the standard class read function module (**ARCHIVE\_GET\_NEXT\_RECORD**) in a loop until the data object can provide no more records. If your archiving program has stored information in the **RECORD\_FLAGS** field, you can use this information in the delete program to determine which data should actually be deleted from the database.

### 4. End delete – **ARCHIVE\_CLOSE\_FILE**

This function module performs the following tasks:

- Closes the archive file
- Releases the included archiving classes
- Discards the current handle

## Developing Reload Programs

### Purpose

A reload program must be provided for archiving objects whose archived data need to be reloaded into the database. The program must always process all the archived data of an archiving session.

You can again use the ADK function modules for this purpose. They allow selective reloading of the data objects. Data objects not to be reloaded are stored in new archive files.

Reload programs must be written with great care, because the archive of the reloaded data objects can no longer be accessed from archive management. This prevents duplicate archiving of objects, and guarantees that the R/3 System can be revised.

ADK creates a new archive file during reloading, into which the data objects not reloaded must be copied. The old archive file is retained, although access through archive management is no longer possible meaning the reloaded data must be re-archived.

### Process Flow

#### Function module call sequence

##### 1. Open reload – ARCHIVE\_OPEN\_FOR\_MOVE

This function module passes two handles as parameters:

- ARCHIVE\_READ\_HANDLE  
This corresponds to the handle returned by function module ARCHIVE\_OPEN\_FOR\_READ. You can perform all read operations with it.
- ARCHIVE\_WRITE\_HANDLE  
This handle enables you to write data objects that are not to be written back to the database, into a new archive. You can therefore only call function module ARCHIVE\_SAVE\_OBJECT using this handle.

This function module:

- Opens the existing archive for reading
- Opens a new archive for writing
- Creates a header entry in the archive management
- Includes the archiving classes of the current archive

To use the same commit counter for the reload program as for the delete program, use function module ARCHIVE\_GET\_CUSTOMIZING\_DATA.

Both function modules are called only once, at the start of the reload program.

**The steps 2 and 3 must be called in a loop for all archived data objects.**

##### 2. Read archived data object from archive file – ARCHIVE\_GET\_NEXT\_OBJECT

As for all archive read accesses, you use this function module to read the next archived data object (with the handle ARCHIVE\_READ\_HANDLE).

This function module:

- Reads an archived data object from the archive file

## Developing Reload Programs

- Supplies the archiving classes with data packets
- Provides the data container for the standard class
- Passes the archived data objects to the WRITE\_HANDLE data container



This function module must be called in a loop, until the archive file cannot provide any more data objects.

### 3. Reload archived data into the database

The data in all archiving classes from the last data object read in by ARCHIVE\_GET\_NEXT\_OBJECT is reloaded into the database by function module ARCHIVE\_RELOAD\_OBJECT\_DATA. This function module need only be called once for each data object (ARCHIVE\_GET\_NEXT\_OBJECT) in the reload program. You must program the reloading of the standard class data into the database yourself.

If the last data object read is not to be reloaded, transfer it to the new archive file using ARCHIVE\_SAVE\_OBJECT. Otherwise you lose data, as you can no longer access the old archive file through archive management.

**With this call (step 3) you end the loop, after all archive file data objects have been processed**

### 4. End reload – ARCHIVE\_CLOSE\_FILE

You use this function module to end the reload procedure.

This function module:

- Closes the archive file
- Updates the archive management records
- Releases the included archiving classes
- Discards the current handle



You only need to pass one of the two handles with the call. The archiving object automatically determines the second handle.

## Developing Analysis Programs

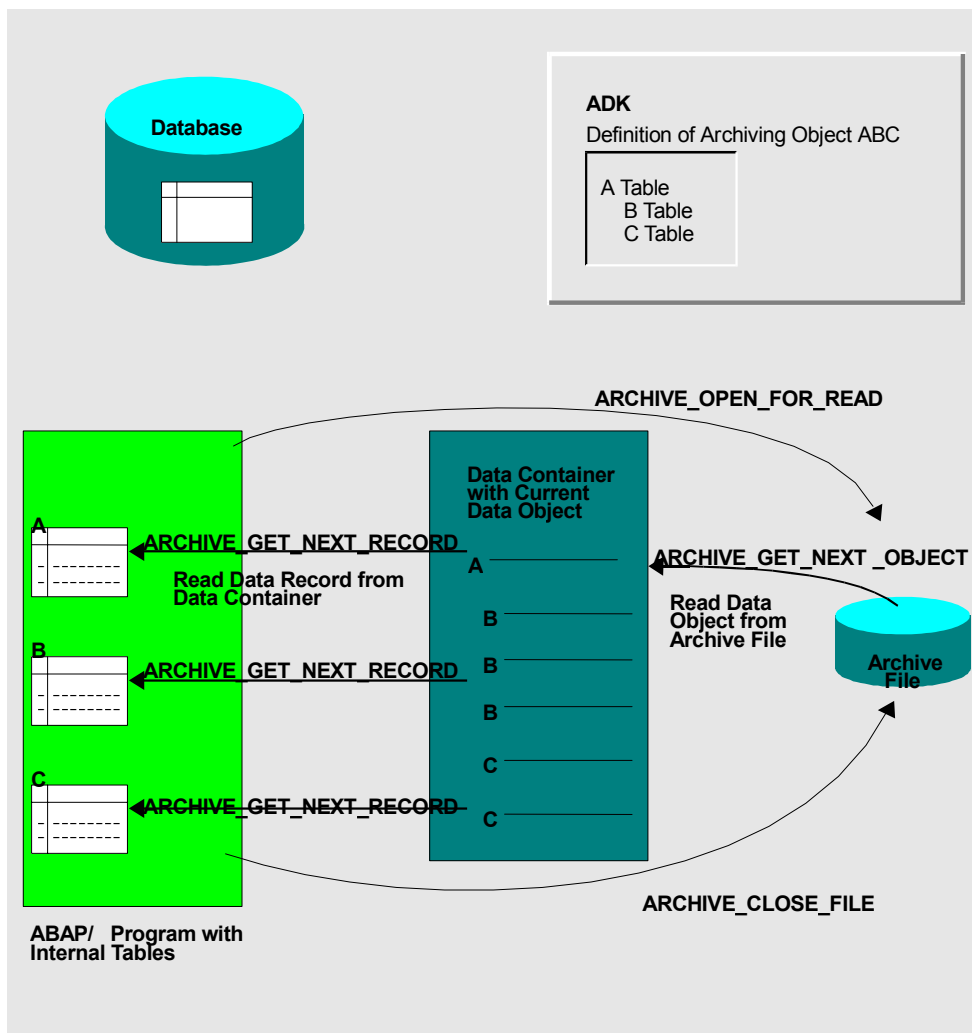
### Purpose

To analyze the data contained in archive files, you can use the function modules to develop an analysis program to read the archive.

You can use the function modules of the archiving classes to access the archiving classes' archived data as if the data was still in the database. There is one restriction: you cannot access all the data in an archiving file at the same time. You can only access the last data object to be read. Interactive reporting using archived data therefore requires some extra effort.

### Process Flow

The following illustration shows how a read or analysis program can access archived data.



The program opens the archive file for reading. Each record in the table is completely read from the archive file into the data container.

## Developing Analysis Programs

From there the program reads the records individually into its internal table.

The analysis is performed when all data objects have been read in from the archive.

The archive file is closed again.

### Function module call sequence

1. **Open archive for read – ARCHIVE\_OPEN\_FOR\_READ**

This function module is only called once at the start of the read or analysis program and returns the handle for read operations.

This function module:

- Opens the existing archive for read
- Includes the archiving classes of the existing archive

2. **Read next data object – ARCHIV\_GET\_NEXT\_OBJECT**

Call this function module in a loop for all data objects in the archive file.

You use this function module to read the opened archive file's data objects sequentially into the data container.

3. **Read the next data object record**

- **ARCHIVE\_GET\_NEXT\_RECORD** (standard class)

Call this function module in a loop for all the standard class records of the last data object to be read.

You use this function module to read the current data object's next standard class data record into the data container for further processing in the program. The first call for a data object automatically reads the first record.

- **Archiving class function modules**

You use these function modules to read the archiving class data into the data container for further processing in the program. For more information, see [Developing Function Modules \[Page 158\]](#).

4. **End analysis – ARCHIVE\_CLOSE\_FILE**

You use this function module to end the analysis.

This function module:

- Closes the archive files
- Releases the included archiving classes
- Discards the current handle

## Maintaining Network Graphic

### Procedure

If your archiving object depends on other applications, you must include all necessary data in the network graphic. Proceed as follows:

Call transaction AOBJ. A list of the existing archiving objects in the system appears.

If your archiving object has predecessors that must be archived first, proceed as follows:

1. Select your archiving object and choose the action *Maintain network graphic* under *Dialog Structure*.  
You may have to scroll in the navigation area to find the desired action.
2. Choose *New entries*.
3. Enter the objects whose data must be archived immediately before those of your archiving object.
4. Save your entries and return to the initial screen of transaction AOBJ.

If the archiving of data for your archiving object is a prerequisite for the archiving of other data in the system, that is, when your archiving object is the predecessor of other objects:

1. Select the successor archiving object in the object list in the initial screen of transaction AOBJ, and choose *Maintain network graphic* under *Dialog Structure*.  
You may have to scroll in the navigation area to find the desired action.
2. Choose *New entries*.
3. Enter the name of your archiving object.
4. Save your entries, and return to the initial screen of transaction AOBJ.
5. Continue for all archiving objects for which your object is the predecessor.

## Creating ADK Indexes and Using Them to Access Archive

## Creating ADK Indexes and Using Them to Access Archive

### Purpose

The index is usually maintained by the delete program during the archiving processes, but it can also be maintained for already existing archive files.



If you are using the [Archive Information System \(SAP AS\) \[Ext.\]](#), you do not need to nor does it make sense to use the ADK index.

### Prerequisites

- The *Create index* indicator is set in the [Definition of the Archiving Object \[Page 131\]](#).
- A unique value was passed for the function module ARCHIVE\_NEW\_OBJECT parameter OBJECT\_ID when writing the archive. This value is used for creating the index.

### Process Flow

#### Create index in delete program

The index entry is written by calling function module ARCHIVE\_ADMIN\_SAVE\_INDEX after you have read the data object from the archive file for deletion. You pass the name of the archiving object, the index entry and the archive file key, which you have previously read using function module ARCHIVE\_GET\_INFORMATION. For examples of how to implement this, see the sample programs RSARCH06 and SBOOKD.

The index entry can be in any format. You pass a character string constructed according to your requirements. Always use the same character string format.

The value of this character string can be passed to the archive file using parameter OBJECT\_ID when the archive is written and copied from the archive file when the function module ARCHIVE\_GET\_NEXT\_OBJECT is called.



You should get the value for MAINTAIN\_INDEX using function module ARCHIVE\_GET\_CUSTOMIZING\_DATA before calling the function module for saving the index entry (ARCHIVE\_ADMIN\_SAVE\_INDEX) so that index creation can be controlled externally. This prevents unnecessary entries.

This value is maintained for the corresponding archiving object [Archive Customizing \[Page 135\]](#) before every archiving session.

#### Creating index after archiving

1. You use program RSARCH15 to add the archived data objects for individual archiving sessions for an archiving object to the index after archiving.



---

**Creating ADK Indexes and Using Them to Access Archive****Accessing individual data objects using an index**

Function module `ARCHIVE_READ_OBJECT` is used to access individual data objects in an archive file using an index. You pass the archiving object name and the index entry. ADK opens the file containing the relevant data object and reads the data object into the data container.



The index must have been created either when writing the archive or afterwards.

You receive the archive handle for the opened archiving file and can now access the individual datasets in the data container using either the standard class methods (`ARCHIVE_GET_NEXT_RECORD`) or, if the archiving object uses archiving classes, using the methods of the archiving class. `RSARCH13` and `SBOOKS` are sample programs that illustrate the use of the ADK index.

## Archiving Classes

# Archiving Classes

## Definition

Archiving classes should be used when complex logical objects are to be archived as service functions during the archiving of application data. This ensures data is consistent. Archiving class are required, for example, when a logical object is to be archived but processing logic cannot be covered by the standard class methods.

For data that represents a business object, you should create an archiving class when you can answer one of the following questions with “Yes”:

- Is the data used repeatedly in the system in the same form, and is it also archived in the various contexts?
- Will other applications use and archive my data structures?

## Structure

### Archiving class services

Archiving classes offer the following services:

- Function module for writing archives
- Function module for generic reading of archived data

A function module with standard interface is also required for classes that already have a class-specific read module, so that external programs and tools, such as SAP AS, can use them. For classes that do not have a function module for reading, assign a module with standard interface.

- Automatic conversion when reading archives
  - Code page adjustment
  - Number format adjustment
  - Changes in record structure
- Optional: Function modules for converting archives



All function module interfaces for archiving classes contain the parameter `ARCHIVE_HANDLE`.

## Function modules

Every archiving class must provide a function module, with which the archiving program informs the system which data is to be archived for a class. This function module is subject to a naming convention; it must begin with the name of the archiving class and end with “\_ARCHIVE\_OBJECT”.



Archiving class	Function module
-----------------	-----------------

Archiving Classes

CHANGEDOCU (change documents)	CHANGEDOCU_ARCHIVE_OBJECT
TEXT (SAPscript texts)	TEXT_ARCHIVE_OBJECT

A function module for reading archived data is usually also provided for analysis programs. This function module is not subject to a naming convention; it is never called directly by the ADK. This function module may be able to read in the database and in the archive file.



This is controlled by parameter ARCHIVE\_HANDLE. If the value "0" is passed for the ARCHIVE\_HANDLE, the function module reads in the database, otherwise it reads in the archive file belonging to the archive handle.

The archiving class receives the data to read through a subprogram called when function module ARCHIVE\_GET\_NEXT\_OBJECT is called by the ADK.

## ADK interface

The archiving class function group contains standardized subprograms that ADK always needs when calling function modules of the archiving object.

When adding an existing archiving class to an archiving object, you do not need to be familiar with the subprograms as they are called in the background by the ADK function modules.

## Developing Archiving Classes

### Procedure

To create an archiving class, proceed as follows:

1. [Specify Function Group \[Page 157\]](#)
2. [Develop Function Modules \[Page 158\]](#)
3. [Develop Subprograms \[Page 161\]](#)
4. [Assign Archiving Class \[Page 138\]](#)
5. [Define Archiving Class \[Page 174\]](#)

Function group SFIL is an example of an archiving class you can find in the system.



As an archiving class can be included in various archiving objects, and several archiving sessions could be processed in one archiving program, several data packets (of an archiving class) must be able to be processed at the same time. At the same time, this places higher demands on the internal memory organization of the function group.

## Specifying Function Groups

### Use

You can use an existing function group for your archiving class. It does not matter which function modules are included in addition to the archiving classes. ADK recognizes the archiving class subprograms from the naming convention.

You can, however, also create your own function group for your archiving class, if this, for example, improves transparency.

You can also use an existing function module, adjusted as necessary, for reading in the database.

Whether you put the archiving class function modules in the same function group as the read function module is up to you. The interfaces of the existing function modules for reading the change documents were extended, and they communicate internally with the new function group to read the desired data. If the parameter `ARCHIVE_HANDLE` is passed with the initial value "0", these function modules read directly from the database.

## Developing Function Modules

### Process Flow

Specify which services you want to provide using the archiving class. A write function module must always be provided (class\_ARCHIVE\_OBJECT). It is also useful to provide a read function module for analysis programs.

### Declaring global data

First describe the required data and structures in INCLUDE LxxxxTOP. Bear in mind that the archiving class can be called by several programs at the same time and you must therefore be able to administer instances. The instances are distinguished by the identifying number (archive handle) and the object key.

The following internal tables are required:

- Table containing the key fields used for archiving
- Table for registering handles and access modes
- Table containing all the table entries in a data object to be archived from the class. This table must contain the STRUCTURE field for the table name of the entry and a SEGMENT field for the data.

### Function module class\_ARCHIVE\_OBJECT

This function module is subject to a naming convention where “class” is the archiving class name in the object definition. The interface of a function module is tailored to the requirements of the data to be archived. There are two parameters, however, that are in every function module of this type:

- ARCHIVE\_HANDLE: Handle for the archiving session
- OBJECT\_DELETE\_FLAG: Delete indicator for the data to be archived from the archiving class. The calling delete program uses this parameter to control whether the class data is to be deleted later by the delete program. The class is responsible (implementation of class\_ARCHIVE\_OBJECT) for determining whether a set delete indicator actually triggers deletion of data. For example, the function module could ignore a “delete request” from a “non-authorized” calling program. However, the function module must adhere to a delete indicator that is *not* set (data is not deleted). The interface must accept the key of the data to be archived.

The class is responsible for how it “remembers” the data. It is advisable to only “remember” the data key initially. The key could, for example, be saved in an internal table defined in the main program. You only need to provide the data to the data object when the subprogram class\_GET\_DATA is called by the ADK function module ARCHIVE\_SAVE\_OBJECT. We recommend you use array operations for this.

The class must “remember” the data until the subprogram class\_FREE\_DATA (by ARCHIVE\_CLOSE\_FILE) or class\_RENEW\_DATA (by ARCHIVE\_NEW\_OBJECT) is called by the ADK function modules.

## Read function module (class-specific, standard method)

This function module is not subject to a naming convention, as it is only called directly from the program (not from ADK). If you have already developed a database read function module, you can use it here. You must just extend it by one parameter used to pass the handle. This parameter must always be called `ARCHIVE_HANDLE`. You use this to specify whether the data is to be read from the database or from the archive. If the handle is passed with the initial value "0", the function module reads directly in the database, otherwise it takes the data from the data packet in its local memory.

For examples, see the function modules `READ_TEXT` (archiving class `TEXT`) and `MEAS_DOCUM_READ_ARCHIVE_OBJECT` (archiving class `MEAS_DOCUM`).

## Read function module: class\_ARCHIVE\_GET\_TABLE (recommended method)

In order for generic tools, such as SAP AS, to be able to access data from an archiving class, a read function module must exist for this archiving class. The function module must follow certain conventions (Example: function module `TEXT_ARCHIVE_GET_TABLE`):

- The name of the function module must be constructed as follows:  
<CLASS>\_ARCHIVE\_GET\_TABLE, where <CLASS> is the name of the archiving class.  
This name was chosen to be similar to the function module `ARCHIVE_GET_TABLE`.
- The interface must import `ARCHIVE_HANDLE` and `RECORD_STRUCTURE` and output the data in `T_DATA` (compare interface in `TEXT_ARCHIVE_GET_TABLE`). In addition, the function module can generate the exception `INVALID_STRUCTURE`.



The returned data must have the structure `RECORD_STRUCTURE`. Otherwise this will not work.

- The function module must be able to read all "direct data" from its archiving class (that is, not the data the archiving class saves using other classes).

When called, all data must be returned to `T_DATA` in the structure as specified in `RECORD_STRUCTURE`. Only the data for the current data object is to be returned. If the function module is unable to return any data to the passed structure, the exception `INVALID_STRUCTURE` can be triggered.

For data belonging to subordinate archiving classes, SAP AS can call the relevant function module itself.

SAP AS attempts to read all the tables of a class that are entered in transaction `ACLA` for an archiving class. This means that the read function module must return exactly these tables.

### Additional information:

SAP AS reads a data object as follows:

1. It reads all the data of an archiving object archived through the standard class. The exact structures do not need to be determined first. SAP AS reads all records using `ARCHIVE_GET_NEXT_RECORD`.
2. Using function module `ARCHIVE_GET_INFORMATION`, SAP AS determines to which classes the data in the archive file are assigned.

---

**Developing Function Modules**

3. The name of the read function module is determined for each class. Then SAP AS checks whether the function module exists (if not, the following does not work).
4. SAP AS then uses table CLASS\_DEF to determine which structures the classes are aware of.
5. The corresponding read function module is then called for every class and its structures.



## Developing Subprograms

### Process Flow

ADK requires class-specific subprograms for controlling the communication between the archiving class and the archive. You must provide these subprograms in the archiving class (LxxxxFnn) function group.

ADK needs these subprograms whenever an archiving object method is called by the program.

- **ARCHIVE\_OPEN\_FOR\_READ or WRITE**
  - **class\_INIT\_WRITE**  
Initializes the archiving class for writing (passes the archive handle)
  - **class\_INIT\_READ**  
Initializes the archiving class for reading
- **ARCHIVE\_NEW\_OBJECT**
  - **class\_RENEW\_DATA**  
Initializes the archiving class for a new data object  
  
This subprogram discards the current archiving class key when writing, or all the data if reading the archive.
- **ARCHIVE\_SAVE\_OBJECT**
  - **class\_GET\_DATA** (“GET” from the ADK perspective)  
Provides data in data container for archiving (the class passes the archived data to the ADK).
- **ARCHIVE\_CLOSE\_FILE**
  - **class\_FREE\_DATA**  
Declares an archive handle invalid when closing an archive
- **ARCHIVE\_GET\_NEXT\_OBJECT**
  - **class\_PUT\_DATA** (“PUT” from the ADK perspective)  
Copies data from the data container after reading the archive (the ADK passes the data read to the class).
- **ARCHIVE\_DELETE\_OBJECT\_DATA**
  - **class\_DELETE\_ARCHIVE\_OBJ**  
Deletes the data contained in the data container from the database  
  
To avoid data inconsistencies, this subprogram uses **PERFORM ON COMMIT** to call another subprogram, in which the data is actually deleted from the database according to global internal tables.
- **ARCHIVE\_ROLLBACK\_WORK**
  - **class\_ROLLBACK\_WORK**  
Discards the data marked for deletion from the database by the archiving class
- **ARCHIVE\_RELOAD\_OBJECT\_DATA**

---

**Developing Subprograms**

- class\_ARCHIVE\_RELOAD\_OBJ  
Reloads the data in the data container into the database

# Initializing the Archiving Classes for Writing

## Use

The name of this subprogram starts with the name of the archiving class and ends with "\_INIT\_WRITE".

## Tasks

This subprogram initializes the archiving class for archiving data.

The classes are informed by this subprogram that archiving methods must be provided for this ARCHIVE\_HANDLE. The archiving classes can pass a data packet to the ADK through an internal table. The data packet is returned when the archive is read by the class\_INIT\_READ subprogram. This table must have been compressed using function module TABLE\_COMPRESS, or be empty.

As explained in conjunction with the information about the global data declaration, several archives may have been opened for an archiving class. The archiving class receives a unique ARCHIVE\_HANDLE for each opened archive.

## When called

This subprogram is called by the ADK whenever a new archive file is opened for writing. As several archive files can be created in one archiving session, this subprogram is called for each archive file. The ARCHIVE\_HANDLE remains the same.

## Interface

<i>FORM</i>	class_INIT_WRITE	<i>TABLES</i>	INIT_TAB
		<i>USING</i>	HANDLE
		<i>CHANGING</i>	LEN

## Parameters

The parameter **INIT\_TAB** indicates the table in which the archiving class can save data it needs for initialization when it reads the archive, and which should be valid for all the data objects in an archive file. Control data can, for example, be stored there so that the appropriate control data is used when the archive is subsequently analyzed, rather than the current values at that time.

The parameter **HANDLE** contains the ARCHIVE\_HANDLE for initialization. The archiving class function modules should note that only write operations are allowed with this handle and must refuse all other calls by raising the exception WRONG\_ACCESS\_TO\_ARCHIVE.

The result of the compression by function module TABLE\_COMPRESS is passed to the archiving through the parameter **LEN**. The function module parameter is called COMPRESSED\_SIZE.

You must register the additional archiving classes that you want to use in your archiving class in this subprogram (ARCHIVE\_REGISTER\_CLASS). Registration ensures that the classes are called in the correct order.



## Getting Data

The name of this subprogram starts with the name of the archiving class, and ends with "\_GET\_DATA".

### Tasks

This subprogram gathers the data the archiving program has requested for archiving a data object. This data is passed using a table, which must be in compressed form.

We recommend you get the data using this subprogram rather than in the function module class\_ARCHIVE\_OBJECT, as the archiving program requests can be collected by the function module and be efficiently processed in this subprogram. An archiving program can also request data for archiving, but discard them later by calling the function module ARCHIVE\_NEW\_OBJECT before they are saved in the archive.

### When called

The ADK calls this subprogram whenever the function module ARCHIVE\_SAVE\_OBJECT is called in the archiving program for an ARCHIVE\_HANDLE.

### Interface

<i>FORM</i>	class_GET_DATA	<i>TABLES</i>	DATA_TABLE	<i>STRUCTURE</i>	ARCH_PACKA
		<i>USING</i>	HANDLE		
		<i>CHANGING</i>	LEN		

### Parameters

The data packet of the data object which is to be written is passed to the data object in the form of a compressed table using the parameter **DATA\_TABLE**. The compression must have been performed by the function module TABLE\_COMPRESS.

The parameter **HANDLE** contains the **ARCHIVE\_HANDLE**, for which the data packet was requested.

The result of the compression by the function module TABLE\_COMPRESS is passed to the archiving via the parameter **LEN**. The function module parameter is called COMPRESSED\_SIZE.

---

**Deleting Local Memory of Archiving Class**

## Deleting Local Memory of Archiving Class

The name of this subprogram starts with the name of the archiving class, and ends with "\_RENEW\_DATA".

### Tasks

This subprogram discards all the data of an archiving class for the current data object by deleting it from the archiving class memory. This task must be performed when data for a new data object are expected.

### When called

This subprogram is called at two different events:

- The archiving program has called the function module ARCHIVE\_NEW\_OBJECT.
- The program which reads an archive file has called the function module ARCHIVE\_GET\_NEXT\_OBJECT. This subprogram is then called before the subprogram class\_PUT\_DATA.

### Interface

FORM class\_RENEW\_DATA USING HANDLE.

### Parameters

The parameter **HANDLE** contains the ARCHIVE\_HANDLE for which the subprogram was called.

## Declaring an Archive Handle Invalid

The name of this subprogram starts with the name of the archiving class, and ends with "\_FREE\_DATA".

### Tasks

This is a clean-up subprogram. All handle information which is passed via the interface, can be deleted from the function group memory. No more calls are made to this handle. If a program calls the function modules of the archiving class with this handle, after this subprogram has been called by the archiving, the function module is to raise the exception

**WRONG\_ACCESS\_TO\_ARCHIVE.**

### When called

This subprogram is called whenever an **ARCHIVE\_HANDLE** becomes invalid. An **ARCHIVE\_HANDLE** becomes invalid whenever the function module **ARCHIVE\_CLOSE\_FILE** is called for this handle.

### Interface

<i>FORM</i>	class_FREE_DATA	<i>USING</i>	HANDLE
-------------	-----------------	--------------	--------

### Parameters

The parameter **HANDLE** contains the **ARCHIVE\_HANDLE** that is to be declared invalid.

## Initializing Archiving Classes for Reading

## Initializing Archiving Classes for Reading

### Use

The name of this subprogram starts with the name of the archiving class and ends with "\_INIT\_READ".

### Tasks

This subprogram initializes the archiving class for reading from archives.

The classes are informed that methods must be provided for reading archived data for this `ARCHIVE_HANDLE`.

Several different archives can be opened for an archiving class. The archiving class receives a unique `ARCHIVE_HANDLE` for each `ARCHIVE_OPEN_FOR_*`.

### When called

The ADK calls this subprogram whenever a new archive file is opened for reading. As several archive files can be read for one `ARCHIVE_HANDLE`, this subprogram is called once per archive file, but then always only for the archive file which is about to be read.

### Interface

<i>FORM</i>	<code>class_INIT_READ</code>	<i>TABLES</i>	<code>INIT_TAB</code>
		<i>USING</i>	<code>HANDLE</code>
			<code>RELEASE_NUMBER</code>
			<code>CODE_PAGE</code>
			<code>NUMBER_FORMAT</code>
			<code>DATE</code>

### Parameters

The parameter **INIT\_TAB** indicates the table in which the archiving class stored data during `class_INIT_WRITE`, which it needs for the read initialization of the archive, and which should be valid for all the data objects in an archive file. Control data can, for example be stored there, so that the appropriate control data is used when the archive is subsequently analyzed, rather than the current values at that time.

The parameter **HANDLE** contains the `ARCHIVE_HANDLE` for which the initialization should run. The archiving class function modules should note that only write operations are allowed with this handle and must refuse all other calls by raising the exception `WRONG_ACCESS_TO_ARCHIVE`.

The parameter **RELEASE\_NUMBER** contains the R/3 System release number at the time when the archive was written.

The parameter **CODE\_PAGE** contains the name of the code page which was active at the time that the archive was written. You can use this parameter, to adjust the data to the current code page with the ABAP language element `TRANSLATE`.



### Initializing Archiving Classes for Reading

You get the number format which was valid at the time of archiving via the parameter **NUMBER\_FORMAT**. You can also use this parameter for the ABAP language element **TRANSLATE**, to adjust the data to the current number format.

The parameter **DATE** contains the date on which the archive was written.

## Copying Data From the Data Container

# Copying Data From the Data Container

The name of this subprogram starts with the name of the archiving class and ends with "\_PUT\_DATA".

## Tasks

This subprogram copies data from the data object, so that they can be read by the archiving class function modules.

## When called

This subprogram is called whenever the function module ARCHIVE\_GET\_NEXT\_OBJECT is called in the archiving program and data for this archiving class exist in the data object.

## Interface

<i>FORM</i>	class_PUT_DATA	<i>TABLES</i>	DATA_TABLE	<i>STRUCTURE</i>	ARCH_PACKA
		<i>USING</i>	HANDLE		

## Parameters

The data packet read from the data object is passed to the archiving class in the form of a compressed table of line type ARCH\_PACKA, using the parameter **DATA\_TABLE**. Compression must be though function module TABLE\_COMPRESS. The parameter HANDLE contains the ARCHIVE\_HANDLE for which the data packet is requested. The result of the compression is passed to the ADK though the function module TABLE\_COMPRESS through parameter LEN. The corresponding parameter of the function module TABLE\_COMPRESS is COMPRESSED\_SIZE.

The parameter HANDLE contains the ARCHIVE\_HANDLE to which the data is passed.



"class\_PUT\_DATA" is the earliest point the conversion routines for structure, code page, and number format adjustment can be called. This is because this is the point where all required nametab information is available.

## Deleting Archived Data

The name of this subprogram begins with the name of the archiving class and ends with "\_DELETE\_ARCHIVE\_OBJ".

### Function

Every archiving class must provide a subprogram that deletes the archived data from the database. The archiving class decides whether this data is actually deleted, or whether only a reference may be deleted, thus avoiding inconsistencies.

The archiving classes automatically know which data were archived, so they do not need to be given this information. The data which were most recently read by the function module `ARCHIVE_GET_NEXT_OBJECT`, and were marked for deletion when the archive was written, are automatically deleted.

The class receives the data to be deleted via the call of the subprogram `class_PUT_DATA` by the archiving. The data format corresponds exactly to the format which was passed to the archiving with the subprogram `class_GET_DATA` when archiving.

### Call

The subprogram is called automatically by the `ARCHIVE_DELETE_OBJECT_DATA` function module. This function module is called only once for each data object which is read by `ARCHIVE_GET_NEXT_OBJECT`.

### Interface

<i>FORM</i>	<code>class_DELETE_ARCHIVE_OBJ</code>	<i>USING</i>	<code>HANDLE</code>
-------------	---------------------------------------	--------------	---------------------

### Parameters

The interface of this subprogram consists of only the parameter **ARCHIVE\_HANDLE**, which specifies the associated archiving object.



The subprogram must save the data to be deleted in global tables, and call an additional subprogram with `PERFORM ON COMMIT`. This subprogram is then processed by the R/3 System when the delete program calls `COMMIT WORK`. The actual delete operations should then be passed to the database in the subprogram called.

The R/3 System calls the subprogram only once each `COMMIT WORK`, however often this `PERFORM ON COMMIT` is called. If `ROLLBACK WORK` is called by the function module `ARCHIVE_ROLLBACK_WORK`, the `PERFORM` command is not executed.

---

**Discarding the Data Selected for Deletion**

## Discarding the Data Selected for Deletion

The name of this subprogram begins with the name of the archiving class, and always ends with "\_ROLLBACK\_WORK".

### Function

The class data selected by the subprogram class\_DELETE\_ARCHIVE\_OBJ and prepared for the delete program in the global internal tables subprogram, is discarded.

### Call

This subprogram is called by the ARCHIVE\_ROLLBACK\_WORK function module.

### Interface

FORM	class_ROLLBACK_WORK	USING	HANDLE
------	---------------------	-------	--------

### Parameters

The parameter HANDLE contains the ARCHIVE\_HANDLE for which the subprogram was called. The function module class\_ROLLBACK\_WORK calls the subprogram for all handles.

## Reloading Archived Data

The name of this subprogram begins with the name of the archiving class, and ends with "\_RELOAD\_ARCHIVE\_OBJ".

### Function

This subprogram reloads archived data from the archive into the database. The archiving class knows automatically, which data were archived, and which data were deleted from the database by the archiving. Data can not be reloaded selectively in the archiving classes. All the data in an archiving class, for the most recently read data object, are always reloaded.

### Call

The subprogram is called automatically by the ARCHIVE\_RELOAD\_OBJECT\_DATA function module. this function module is called only once for each data object read by ARCHIVE\_GET\_NEXT\_OBJECT.

### Interface

<i>FORM</i>	class_RELOAD_ARCHIVE_OBJ	<i>USING</i>	HANDLE
-------------	--------------------------	--------------	--------

### Parameters

The interface of this subprogram consists only of the parameter **ARCHIVE\_HANDLE**, which specifies the associated archiving object.



When programming the subprogram, you can specify which data from the archiving class can be reloaded, and which not. This can be useful if reloading certain data would cause database collisions, for example, if the database already contains data with the same key, which would be overwritten by the reload.

## Defining Archiving Classes

### Defining Archiving Classes

You must specify which function group contains the archiving class, so that ADK knows where it can find the subprograms belonging to the archiving class.

Proceed as follows:

1. Call the transaction `ACLA`. A list of the existing archiving classes in the system appears.
2. Choose *New entries*.
3. Enter the archiving class name, a short text, and the associated function group.
4. Save your entries.

The following must be specified to ensure that the ADK uses the structures specified for automatic conversion of code page, number format, and structure changes:

- All structures or tables to be archived where the additional *Do not delete data* option is set.

To ensure Repository information is complete, including the display of the tables of an archive object to be archived (transaction `DB15`), the following additional information must be specified in transaction `ACLA`:

- Tables from which you only delete data
- The archiving classes used by archiving classes

Proceed as follows:

1. Select the archiving class and choose *Tables to be Archived*.
2. Choose *New entries*.
3. Enter all structures to be archived. If you do not want data to be deleted during archiving, select the *Do not delete* indicator.
4. Choose *Tables from Which You Only Delete Entries*.
5. Enter all tables from which entries are to be deleted but not archived.
6. Choose *Archiving Classes Used*.
7. Enter all classes used in the class you created.

### Conversion routines

If structures, code pages, or number formats need to be converted for reading, the ADK gets the data from class (packed) and generates conversion routines, which are then called by the class. The prerequisite for this is that all nametab information has been read. This is not the case when `class_init_read` is run.

The ADK reads an archive file in the following sequence:

1. HeadA (%A)
2. HeadB (%B)
3. HeadN (%N) (Nametab)
4. %Class (for example, CHANGEDOCU)
5. %Package

6. %N
7. %N\_\_\_\_ \* class\_init\_read (\* corresponds to the time at which all nametab information is available)
8. %Class (Text)
9. %N (table to archive, ACLA)
10. %N\_\_\_\_ \* class\_init\_read (\*corresponds to the time at which all nametab information is available)
11. %Start (Start of a data object)
12. %Class (Start of class data)
13. %Class (Start of class data)

The ADK function module ARCHIVE\_CONVERSION\_FORMROUTINE checks whether a conversion was successful.

This is only possible if the archiving class is correctly defined in transaction ACLA. Each structure is checked separately using the name of the structure as the passed value.



All structures should be carefully and correctly registered in transaction ACLA to ensure the ADK can generate the conversion routine.

## Optimizing performance

Class data is always archived in the context of a data object, such as SAPscript texts in an FI document. Each text from a document (data object) is read separately. This should be taken into account when developing archiving classes to prevent performance problems.

Take the following into account:

- Data should first be read in the first data access (subprogram "\_GET\_DATA") and not in function module class\_ARCHIVE\_OBJECT as this function module collects the requests of the archiving program, but the requests are efficiently (in terms of performance) fulfilled by the subprogram. An archiving program may also request data that it later discards before archiving the data, using ARCHIVE\_NEW\_OBJECT.
- A "pre-get" should be provided. This additional function module reports the keys to be used for reading, such as in archiving class TEXT or MEAS\_DOCUM).
- Function module ARCHIVE\_DELETE\_OBJECT\_DATA calls the subprogram for deleting class data. This function module gets the corresponding key. The delete phase should not be started immediately, but instead at PERFORM ON COMMIT (for example, TEXT\_DELETE\_ARCHIVE\_OBJECT). COMMIT WORK is sent by the delete program after a set number of data objects <commit counter>. If an error occurs during the delete phase, all deletions are reset. The internal tables containing the keys marked for deletion are reset by the subprogram "\_ROLLBACK\_WORK".

## Archiving Functions

## Archiving Functions

### Function group ARCH

- **ARCHIVE\_CLOSE\_FILE**  
With this function module, you close all archive files gathered under one handle, independently of whether they were opened for reading, writing or reloading.
- **ARCHIVE\_DELETE\_OBJECT\_DATA**  
With this function module, you call the delete subprograms of the archive classes for the current data object.
- **ARCHIVE\_GET\_CUSTOMIZING\_DATA**  
This function module returns the Commit counter and the create index indicator. The Commit counter determines, after how many data objects a COMMIT WORK is issued. The create index indicator specifies whether the delete program should insert the archived and deleted data objects in the index.
- **ARCHIVE\_GET\_FIRST\_RECORD** (standard class only)  
With this function module you set the record pointer to the first record in the data container in the data object which was previously read by ARCHIVE\_GET\_NEXT\_OBJECT, and read this record.  
This function module combines the functions of the ARCHIVE\_SET\_RECORD\_CURSOR and ARCHIVE\_GET\_NEXT\_RECORD function modules.  
If you only need the fields RECORD\_FLAGS and RECORD\_STRUCTURE, you can use the function module ARCHIVE\_GET\_RECORD\_INFO.
- **ARCHIVE\_GET\_INFORMATION**  
With this function module you get current information for a handle, such as date, release, SAP System, and archive name.
- **ARCHIVE\_GET\_OPEN\_FILES**  
With this function module, you fill a table with the file names of all archive files which are currently being processed by the ADK function modules.
- **ARCHIVE\_GET\_NEXT\_OBJECT**  
With this function module, you read the next data object for a handle, from an archive which is open for reading in the data container. This call is a prerequisite for the function module ARCHIVE\_GET\_NEXT\_RECORD or ARCHIVE\_GET\_FIRST\_RECORD and the archiving classes call.  
If you use archiving classes, their data is accessible through their function modules after they have been called.
- **ARCHIVE\_GET\_NEXT\_RECORD** (standard class only)  
With this function module, you sequentially read the next record in a data container in a data object which was read by ARCHIVE\_GET\_NEXT\_OBJECT. The first call automatically reads the first record.



The following function modules (ARCHIVE\_GET\_NEXT\_STRUCT\_SPECIF and ARCHIVE\_GET\_RECORD\_INFO) were developed for the easy implementation of logical database archive read operations. They can, of course, also be used in other programs.



- **ARCHIVE\_GET\_NEXT\_STRUCT\_SPECIF** (standard class only)  
With this function module you read archives with specified structures with logical databases. The group change logic is integrated.  
You can, however, also use the function module ARCHIVE\_GET\_NEXT\_RECORD, but in this case you must program the hierarchy step group change logic yourself.
- **ARCHIVE\_GET\_RECORD\_INFO** (standard class only)  
With this function module, you get information in logical databases about archived datasets.
- **ARCHIVE\_GET\_TABLE**  
With this function module, you read several records from a data object which was read by ARCHIVE\_GET\_NEXT\_OBJECT, into an internal table.
- **ARCHIVE\_GET\_WITH\_CURSOR** (standard class only)  
With this function module, you can directly read standard class datasets in a data object. You get the necessary record pointer during sequential access to a data object record via the parameter RECORD\_CURSOR.  
This function module is useful for "remembering" records via the record pointer for later further processing.
- **ARCHIVE\_NEW\_OBJECT**  
With this function module you request a new data container to write for a handle.
- **ARCHIVE\_OPEN\_FOR\_MOVE**  
With this function module you open one or several archive files for reloading archived data. You receive an archive handle for the archive file to be read and another archive handle for writing those data objects, which are not to be reloaded. This facilitates the selective reloading of individual data objects from archives into the R/3 System.
- **ARCHIVE\_OPEN\_FOR\_READ**  
With this function module you open an existing archive file for reading. A handle is created, via which this file can be read. You can also open several archive files at the same time. They all share one handle.  
Function modules which read using this handle treat all the files with this handle like a single file.
- **ARCHIVE\_OPEN\_FOR\_WRITE**  
With this function module you create a new archive file and a handle, with which you have write access to this file.  
If you have not specified a file name in the [Archiving-Object Specific Customizing \[Page 135\]](#), the platform independent logical file name ARCHIVE\_DATA\_FILE is automatically used, to determine a valid, platform-independent file name.  
You can also specify via control parameters, whether the delete program for the archived data should be automatically called after writing.
- **ARCHIVE\_PUT\_RECORD** (standard class only)  
With this function module you pass a data set to the data container which was previously requested with the function module ARCHIVE\_NEW\_OBJECT.  
All data sets which you pass to the data container are written to the archive file together by the function module ARCHIVE\_SAVE\_OBJECT.
- **ARCHIVE\_PUT\_TABLE** (standard class only)  
With this function module, you pass an internal table to the data container which was previously requested by ARCHIVE\_NEW\_OBJECT. The internal table records are entered in the data container as single records.

## Archiving Functions

- **ARCHIVE\_REGISTER\_CLASS**  
With this function module, you dynamically assign archiving classes to an archiving object.
- **ARCHIVE\_RELOAD\_OBJECT\_DATA** (only for archiving classes)  
With this function module, you call the reload subprograms of the archiving classes.
- **ARCHIVE\_ROLLBACK\_WORK**  
If a ROLLBACK WORK has to be carried out in a delete or reload program, it should be done by calling this function module, not by the ABAP command ROLLBACK WORK. This function module guarantees the correct resetting of the data for all archiving classes used.
- **ARCHIVE\_SAVE\_OBJECT**  
With this function module, you write a data object into the archive file. As well as the data passed by ARCHIVE\_PUT\_RECORD, the data which were passed via the archiving classes are taken into account.
- **ARCHIVE\_SET\_RECORD\_CURSOR** (standard class only)  
With this function module you set the standard class record pointer of the last data object to be read. You can then read the next record with ARCHIVE\_GET\_NEXT\_RECORD.
- **ARCHIVE\_WRITE\_STATISTICS** (standard class only)  
With this function module, you create a statistics print-out for the data object which you have written to the archive files with ARCHIVE\_SAVE\_OBJECT.