

BC ABAP User's Guide



Release 4.0B



Copyright

© Copyright 1998 SAP AG. All rights reserved.

No part of this brochure 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.

SAP AG further does not warrant the accuracy or completeness of the information, text, graphics, links, or other items contained within these materials. SAP AG shall not be liable for any special, indirect, incidental, or consequential damages, including without limitation, lost revenues or lost profits, which may result from the use of these materials. The information in this documentation is subject to change without notice and does not represent a commitment on the part of SAP AG for the future.

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

Microsoft®, WINDOWS®, NT®, EXCEL® 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.

OSF/Motif® is a registered trademark of Open Software Foundation.

ORACLE® is a registered trademark of ORACLE Corporation, California, USA.




INFORMIX®-OnLine *for SAP* is a registered trademark of Informix Software Incorporated.

UNIX® and X/Open® are registered trademarks of SCO Santa Cruz Operation.

ADABAS® is a registered trademark of Software AG.

SAP®, R/2®, R/3®, RIVA®, ABAP/4®, SAP ArchiveLink®, SAPaccess®, SAPmail®, SAPoffice®, SAP-EDI®, R/3 Retail®, SAP EarlyWatch®, SAP Business Workflow®, ALE/WEB™, Team SAP™, BAPI™, Management Cockpit™ are registered or unregistered trademarks of SAP AG.

Icons

Icon	Meaning
	Caution
	Example
	Note

Contents

BC ABAP User's Guide	21
ABAP Basics	22
The R/3 Basis System: Overview	23
Overview of the Components of Application Programs	38
The ABAP Programming Language	59
Creating and Changing ABAP Programs	60
Opening Programs in the Repository Browser	61
Entering the Program Name Directly	62
Creating a New Program	63
Displaying or Changing an Existing Program	64
Display Programs of a Development Class	65
Open Programs Using the ABAP Editor	67
Opening Programs With Forward Navigation	69
Maintaining Program Attributes	74
Program Attributes	78
Editing Your Program	79
Assigning Transaction Codes to Executable Programs (Reports)	81
Naming Conventions	83
Basic Statements	84
ABAP Program Syntax and Layout	85
Syntax Elements	86
Statements	87
Keywords	88
Comments	90
Syntax Structure	91
Structure of Statements	92
Structure of Comments	93
Concatenating Similar Statements	94
ABAP Program Layout	95
Indenting Blocks of Statements	96
Using Modularization Tools	97
Inserting Program Comments Correctly	98
Pretty Printer	99
Inserting Predefined Structures	100
Inserting Predefined Keyword Structures	101
Inserting Predefined Comment Lines	102
Declaring Data	103
Introduction to Data Types and Objects	104
Data Types	105
Elementary Data Types - Predefined	106
Numeric Data Types	107
Elementary Data Types - User-Defined	108
Structured Data Types	109

Compatibility of Data Types	111
Data Objects	112
Literals	113
Variables.....	115
Constants	116
System-Defined Data Objects	117
Creating Data Objects and Data Types.....	118
The DATA Statement	119
Basic Form of the DATA Statement.....	120
Naming a Variable	121
Specifying the Data Type and the Length of the Variable	122
Specifying a Start Value.....	124
Specifying the Number of Digits after the Decimal Point.....	125
DATA Statement for Structures	126
The CONSTANTS Statement.....	127
The STATICS Statement.....	128
The TABLES Statement	129
The TYPES Statement	130
Summarizing Examples.....	132
Example of Predefined Elementary Data Types and Objects.....	133
Example of User-Defined Elementary Data Types and Objects	134
Example of Structures	135
Example of Internal Tables.....	136
Type Groups	137
Determining the Attributes of Data Objects	139
Determining the Field Length	140
Determining the Data Type	141
Determining the Output Length	143
Determining the Decimal Places	144
Determining the Conversion Routine	145
Working with Text Elements	146
Text Elements - Concept	147
Creating and Changing Text Elements	148
Titles and Headers	150
Changing the Title of a Program	151
Creating and Changing List and Column Headers	152
Selection Texts.....	154
Text Symbols.....	157
Comparing Text Elements	159
Comparing Selection Texts	160
Comparing Text Symbols.....	164
Copying Text Elements	167
Translating Text Elements.....	168
Processing Data.....	171
Assigning Values	172
Assigning Values with MOVE.....	173

Basic Assignment Operations	174
Assigning Values with Offset Specifications	176
Copying Values between Components of Structures	178
Assigning Values with WRITE TO	179
Basic Form of the WRITE TO Statement	180
Specifying the Source Field at Runtime	182
Writing Values with Offset Specifications	183
Resetting Values to Initial Values	184
Numerical Operations	185
Performing Arithmetic Operations	187
Basic Arithmetic Operations	188
Performing Arithmetic Operations on Structures	190
Adding Sequences of Fields	191
Using Mathematical Functions	192
Functions for all numeric data types	193
Floating-Point Functions	195
Processing Packed Numbers	196
Processing Date and Time Fields	197
Processing Character Strings	199
Shifting Field Contents	200
Shifting a Structure by a Given Number of Positions	201
Shifting a Structure up to a Given String	202
Shifting a Structure According to the First or Last Character	203
Replacing Field Contents	204
Converting to Upper/Lower Case and Substituting Characters	206
Converting into a Sortable Format	207
Overlaying Character Fields	208
Searching for Character Strings	209
Obtaining the Length of a Character String	211
Condensing Field Contents	212
Concatenating Character Strings	213
Splitting Character Strings	214
Assigning Parts of Character Strings	215
Specifying Offset Values for Data Objects	216
Type Conversions	218
Convertibility of Elementary Data Types	219
Source Type Character	220
Source Type Date	221
Source Type Floating Point Number	222
Source Type Numeric Text	223
Source Type Packed Number	224
Source Type Time	225
Source Type Hexadecimal	226
Convertibility of Structures	227
Compatible Structures	228
Incompatible Structures and Elementary Fields	229

Structures with Internal Tables as Components	231
Convertibility of Internal Tables	232
Alignment of Data Objects.....	233
Controlling the Flow of an ABAP Program	234
Programming Logical Expressions	235
Comparisons with All Field Types	236
Comparisons with Character Strings and Numeric Strings.....	238
Comparisons of Bit Structures.....	241
Checking Whether a Field Belongs to a Range	243
Checking for the Initial Value.....	244
Checking Selection Criteria	245
Combining Several Logical Expressions.....	246
Programming Branches and Loops.....	247
Conditional Branching using IF	248
Conditional Branching with CASE	249
Unconditional Looping using DO.....	251
Conditional Loops using WHILE	254
Terminating Loops.....	256
Terminating a Loop Pass Unconditionally.....	257
Terminating a Loop Pass Conditionally	258
Terminating a Loop Entirely	259
Creating and Processing Internal Tables	260
What are Internal Tables?	261
Internal Tables as Data Types	262
Internal Tables as Dynamic Data Objects.....	263
Using Internal Tables	264
Accessing Internal Tables	265
Operations on Internal Tables	266
Choosing a Table Type	267
Declaring Internal Tables	268
Internal Tables as Parameters for Routines.....	269
Creating Internal Tables	270
Creating Internal Table Data Types	271
Creating Internal Table Data Objects.....	273
Creating Internal Tables by Referring to Another Table	274
Creating Internal Tables by Referring to a Structure	275
Creating Internal Tables with a New Structure	276
Working with Internal Tables	277
Filling Internal Tables	278
Appending Lines	279
Appending Lines Depending on the Standard Key.....	281
Inserting Lines.....	283
Appending Lines of an Internal Table	286
Inserting Lines of an Internal Table.....	287
Copying Internal Tables	289
Reading Internal Tables	292

Reading Internal Tables Line by Line	293
Reading Single Lines Using the Index	295
Reading Single Lines Using a Key	296
Reading Single Lines With User-Defined Keys	297
Reading Single Lines with the Standard Key	300
Binary Search	302
Comparing the Contents of Single Lines	303
Reading Parts of Single Lines	305
Searching Internal Tables for Character Strings	306
Determining the Attributes of Internal Tables	308
Changing and Deleting Lines of Internal Tables	309
Changing Lines with MODIFY	310
Changing Lines with WRITE TO	313
Deleting Lines in a Loop	314
Deleting Lines Using the Index	315
Deleting Adjacent Duplicate Entries	316
Deleting Selected Lines	318
Sorting Internal Tables	319
Creating Ranked Lists	323
Loop Processing	324
Calculating Totals	325
Using Control Levels for Groups of Lines	327
Comparing Internal Tables	332
Initializing Internal Tables	334
Working with Field Symbols	336
Field Symbols - Concept	337
Defining Field Symbols	338
Defining Field Symbols for Internal Fields	339
Field Symbols Without Type Specifications	340
Typing Field Symbols	341
Defining Structured Field Symbols	343
Defining Local Field Symbols	345
Assigning Data Objects to Field Symbols	346
Basic Form of the ASSIGN Statement	347
Static ASSIGN	348
Static ASSIGN with Offset Specifications	349
Dynamic ASSIGN	351
Dynamic ASSIGN of Table Work Areas	353
Assigning Field Symbols to Other Field Symbols	354
Assigning Components of Field Strings	355
Defining the Data Type of a Field Symbol	356
Changing the Number of Decimal Places	358
Assigning a Local Copy of a Global Field	359
Runtime Checks	360
Saving and Reading Data	361
Storing Data Objects as Clusters	362

Data Clusters in ABAP Memory	363
Storing Data Objects in ABAP Memory.....	364
Reading Data Objects from Memory.....	365
Deleting Data Clusters in Memory.....	367
Data Clusters in Databases.....	368
Cluster Databases.....	369
Structure of Cluster Databases.....	370
Example of a Cluster Database	371
Storing Data Objects in Cluster Databases.....	372
Creating a Table of Contents for a Data Cluster	374
Reading Data Objects from Cluster Databases	376
Deleting Data Clusters from Cluster Databases	378
Accessing Cluster Databases with Open SQL statements.....	379
Working with Files.....	381
Working with Files on the Application Server.....	382
File Handling in ABAP.....	383
Opening a File.....	384
Basic Form of the OPEN DATASET Statement	385
Opening a File for Reading.....	386
Opening a File for Writing	387
Opening a File for Appending	390
Specifying Binary Mode	392
Specifying Text Mode	394
Opening a File at a Specific Position	396
Sending Operating System Commands	398
Receiving the Operating System Message.....	399
Closing a File	400
Deleting a File	401
Writing Data to Files	402
Reading Data from Files.....	404
Automatic Checks before File Operations.....	406
Authorization Check for particular Programs and Files	407
General Check before File Access	409
Working with Files on the Presentation Server	412
Writing Data to the Presentation Server With User Dialog	413
Writing Data to the Presentation Server Without User Dialog	416
Reading Data from the Presentation Server With User Dialog.....	419
Reading Data from the Presentation Server Without User Dialog	422
Checking Files on the Presentation Server.....	425
Using Platform-Independent File Names	428
Maintaining Syntax Groups	429
Assigning Operating Systems to Syntax Groups	430
Creating and Defining Logical Paths.....	431
Creating and Defining Logical File Names.....	432
Using Logical Files in ABAP Programs	433
Modularization.....	436

Modularizing ABAP Programs	437
Source Code Modules	439
Defining and Calling Macros	440
Include Programs	441
Creating Include Programs	442
Using Include Programs	443
Subroutines	444
Defining Subroutines	445
Calling Subroutines	446
Calling Internal Subroutines	447
Calling External Subroutines	448
Specifying the Subroutine Name at Runtime	449
Calling Subroutines from a List	450
Passing Data Between Calling Programs and Subroutines	451
Declaring Data as Common Part	452
Passing Data by Parameters	454
Passing by Reference	456
Passing by Value	458
Passing by Value and Result	459
Typing Formal Parameters	461
Passing Structures to Subroutines	463
Passing Internal Tables to Subroutines	466
Defining Local Data Types and Objects in Subroutines	469
Defining Dynamic Local Data Types and Objects	470
Defining Static Local Data Objects	471
Defining Local Data Objects Explicitly	472
Terminating Subroutines	474
Terminating Subroutines Unconditionally	475
Terminating Subroutines Conditionally	476
Function Modules	477
Working with Existing Function Modules	478
Calling a List of Available Function Modules	479
Displaying Attributes of Function Modules	481
Documentation	482
Interfaces	483
Source Code	486
Testing Function Modules	487
Calling Function Modules	488
Creating and Programming Function Modules	491
Creating Function Modules	492
Programming Function Modules	494
Function Groups	497
Special Techniques	499
Checking the Runtime of Program Segments	500
GET RUN TIME FIELD	501
Runtime Measurement of Database Accesses	503

Generating and Running Programs Dynamically	505
Creating a New Program Dynamically.....	506
Changing Existing Programs Dynamically	508
Running Programs Created Dynamically	509
Creating and Starting Temporary Subroutines.....	512
ABAP Objects.....	515
What are ABAP Objects?	516
What is Object Orientation?.....	517
From Function Groups to Objects	519
Classes and Class Components.....	522
Reference Variables and Object Instances	526
Interfaces	532
Triggering and Handling Events	534
Inheritance	535
Further Reading.....	536
ABAP Database Access	537
Reading and Processing Database Tables	538
Database Tables and SQL Concepts	539
Reading Data from Database Tables	542
Defining the Result of a Selection	543
Selecting All Data from Several Lines	544
Selecting All Data from a Single Line.....	545
Selecting and Processing Data from Specific Columns.....	546
Specifying the Database Table to be Read.....	550
Specifying the Database Table Name in the Program.....	551
Specifying the Database Table Name at Runtime	552
Specifying the Target Area for the Selected Data	553
Reading Data into a Work Area	554
Reading Data into an Internal Table	555
Reading Data Component by Component	557
Choosing the Lines to be Read	559
Specifying Conditions for Line Selection in the Program	560
Specifying Conditions for Line Selection at Runtime	563
Grouping Lines	567
Specifying the Order of Lines	568
Changing the Contents of Database Tables.....	570
Adding Lines to Database Tables	571
Adding a Single Line	572
Adding Several Lines from an Internal Table	574
Changing Lines in Database Tables	576
Changing a Single Line	577
Changing Several Lines	579
Changing Several Lines Using an Internal Table.....	580
Adding or changing lines	581
Inserting a Single Line.....	582
Inserting Several Lines.....	583
Deleting Lines from Database Tables	584

Deleting a Single Line	585
Deleting Several Lines	586
Deleting Several Lines Using an Internal Table	587
Reading Lines of Database Tables Using a Cursor	588
Opening a Cursor	589
Using a Cursor to Read Data	590
Closing the Cursor	591
Example of Reading Data Using a Cursor	592
Writing or Undoing Changes to Database Tables	594
Specifying Clients for Processing Database Tables	596
Performance Notes	597
Using Native SQL Statements in an ABAP Program	599
Native SQL for Informix	601
Native SQL for DB2 Common Server	616
Locking Database Objects During Program Execution	628
Checking the Authorization of Program Users	629
Using Contexts	630
What are Contexts?	631
The Context Builder in the ABAP Workbench	632
Creating and Editing a Context	633
Using Tables as Modules	636
Using Function Modules as Modules	639
Using Contexts as Modules	642
Testing a Context	644
Buffering Contexts	646
Fields	649
Modules	651
Interfaces	653
Using Contexts in ABAP Programs	654
Finding and Displaying a Context	655
Creating an Instance of a Context	657
Supplying Context Instances with Key Values	658
Querying Data from Context Instances	659
Message Handling in Contexts	661
Message Handling in Table Modules	662
Message Handling in Function Module Modules	664
Working With Contexts - Hints	666
Programming Database Updates	667
Transactions and Logical Units of Work	668
Database Logical Unit of Work (LUW)	669
SAP LUW	672
SAP Transactions	676
The R/3 Lock Concept	677
Example Transaction: SAP Locking	681
Update Techniques	683
Asynchronous Update	684
Updating Asynchronously in Steps	687
Synchronous Update	688

Local Update.....	689
Creating Update Function Modules	690
Calling Update Functions.....	691
Calling Update Functions Directly	692
Adding Update Task Calls to a Subroutine	693
Special LUW Considerations	694
Transactions that Call Update-Task Functions	695
Dialog Modules that Call Update-Task Functions	696
Error Handling for Bundled Updates.....	697
ABAP User Interfaces	699
Screens	700
Processing User Input on a Screen	701
Programming with Function codes	702
Setting up Function codes.....	703
Handling Function codes.....	706
Handling Field Selections.....	708
Sharing GUI Statuses.....	709
Programming with Radio Buttons	710
Programming with Check Boxes	711
Controlling the Screen Flow.....	712
Introduction to Screen Flow Control	713
Setting the Next Screen.....	716
Calling a New Screen Sequence.....	717
Leaving the Current Screen.....	719
Example Transaction: Setting and Calling Screens	720
Processing Screens in the Background	722
Modifying the Screen	723
Setting Screen Field Attributes	724
Changing Screen Field Attributes with the Function Field Selection.....	727
Field Selection - Overview.....	728
Calling Field Selection.....	729
Combination Rules For Attributes	732
Screen Painter Attributes	734
Generating the Field Selection	735
Function Modules for Field Selection	736
Linking Fields	739
The Display Attribute 'Active'	741
Authorization for Field Selection	742
Using Subscreens	743
Manipulating the Cursor	745
Tab strips	746
Using a tab strip.....	747
Tab strip components	750
Programming tab strips	752
Defining a tab strip in the Screen Painter	755
Further Notes.....	759
Using Tables in a Screen.....	760

Introduction	761
Using the LOOP Statement	763
Looping Directly Through a Screen Table	764
Looping Through an Internal Table	766
How the System Transfers Data Values	767
Scrolling and the Scroll Variables	768
Using Table Controls	770
Declaring Table Controls in ABAP	771
Setting Table Control Attributes	773
Example Transaction: Table Controls	775
Using Step Loops	779
Checking User Authorizations	781
Defining an Authority Check	782
Defining Authorization Objects	785
Defining Authorization Fields	786
Programming Field- and Value-Help	787
User-specific F1 Help	789
Context-sensitive Value Help	790
Selection Screens	794
Working with Selection Screens	795
What are Selection Screens?	796
Standard Selection Screens and Logical Databases	797
Defining Selection Screens	800
Standard and User-defined Selection Screens	801
PARAMETERS - Defining Input Fields for Variables	803
Basic Form of the PARAMETERS Statement	804
Assigning Default Values to Parameters	806
Suppressing Display of Parameters	807
Allowing Parameters to Accept Upper and Lower Case	808
Making Parameters Required Input Fields	809
Creating Checkboxes on the Selection Screen	810
Creating Radio Button Groups on the Selection Screen	811
Using Default Values from SAP Memory	812
Assigning Matchcode Objects to Parameters	813
Assigning Parameters to a Modification Group	814
SELECT-OPTIONS - Defining Selection Criteria	815
Selection Tables	816
RANGES	818
Program-specific Selection Criteria and Logical Databases	820
Basic Form of the SELECT-OPTIONS Statement	821
Assigning Default Values to Selection Criteria	825
Restricting the Selection Table to One Line	827
Restricting the Selection Table to Single Value Selection	828
Preventing the Transfer of Selection Criteria to Logical Databases	829
Further Options for Selection Criteria	831
SELECTION-SCREEN - Formatting	832

Specifying Blank Lines, Underlines, and Comments	833
Blank Lines	834
Underlines	835
Comments	836
Example of Blank Lines, Underlines, and Comments	837
Placing Several Elements On a Single Line	839
Positioning an Element	840
Creating Blocks of Elements	841
Creating Pushbuttons in the Application Toolbar	842
Creating Pushbuttons on the Selection Screen	844
Calling Selection Screens	846
Calling Standard Selection Screens	847
Calling User-defined Selection Screens	848
Call From a Program	849
Call as a Report Transaction	853
Call as a Dialog Transaction	855
Using Selection Criteria in Programs	857
Using Selection Tables in the WHERE Clause	858
Using Selection Tables in Logical Expressions	859
Using Selection Tables with the CHECK Statement in GET Events	862
Pre-Setting Selections Using Variants	864
What is a Variant?	865
Creating and Changing Variants	866
Displaying a List of all Variants	867
Creating Variants	869
Attributes of a Variant	870
Changing Variants	872
Deleting Variants	873
Printing Variants	874
Using Variables with Variants	875
Using Variable Date Calculations	876
Using User-Specific Values	877
Changing Values of User-Specific Variables Interactively	879
Changing Values of User-Specific Variables from the Program	881
Using Table TVARV	882
Creating New Entries in Table TVARV	883
Changing Entries in Table TVARV	885
Running an Executable Program (Report) with a Variant	887
Lists	889
Creating Simple Lists with the WRITE Statement	890
The WRITE Statement	891
Positioning WRITE Output on the Screen	894
Formatting Options	896
Outputting Symbols and Icons on the Screen	898
Lines and Blank Lines on the Output Screen	899
Outputting Field Contents as Checkboxes	900

Using WRITE via a Statement Structure	901
Formatting Data	905
Example for Refined Data	906
Refining Data During Reading	909
Refining Data Using Internal Tables	911
Refining Data Using Flat Internal Tables	912
Refining Data Using Nested Internal Tables	914
Refining Data Using Extract Datasets	916
Creating and Filling Extract Datasets	917
Defining Extract Records as Field Groups	918
Assigning Fields to a Field Group	919
Extracting a Dataset	920
Processing Extract Datasets	922
Processing Control Levels	923
Calculating Numbers and Totals	927
Reading Extract Datasets	930
Sorting Extract Datasets	933
Example for Refining Data Using Extract Datasets	936
Creating Complex Lists	938
The Standard List	939
Example for a Standard List	940
Structure of the Standard List	942
Standard Page Header	943
Standard Page	944
Width of the Standard List	945
User Interface of the Standard List	946
Printing the Output List	947
Saving a List	948
Saving the List in SAPoffice	949
Saving the List in the Reporting Tree	950
Saving the List as Local File on the Presentation Server	951
Modifying List and Column Headers	952
The Self-Defined List	953
Individual Page Header	954
Determining the List Width	956
Determining the Page Length	958
Defining a Page Footer	960
Lists with Several Pages	962
Programming Page Breaks	963
Unconditional Page Break	964
Conditional Page Break- Defining a Block of Lines	966
Standard Page Headers of Individual Pages	968
Page Lengths of Individual Pages	970
Page Width of List Levels	974
Scrolling from within the Program	975
Scrolling Window by Window	976

Scrolling by Pages	977
Scrolling to the List's Margins	979
Scrolling by Columns	980
Laying Out List Pages	982
Positioning the Output	983
Absolute Positioning.....	984
Horizontal Positioning	985
Vertical Positioning	986
Positioning Output Beneath the Page Header.....	987
Example for Absolute Positioning	988
Relative Positioning.....	989
Producing a Line Feed.....	990
Positioning Output Underneath Other Output.....	991
Positioning Output in the First Line of a Line Block	992
Examples for Relative Positioning	993
Formatting Output	995
The FORMAT Statement	996
Colors in Lists	997
Demonstrating the Colors Available in Lists.....	999
Example for Using Colors in Lists.....	1001
Enabling Fields for Input	1003
Outputting Fields as Hotspots.....	1005
Special Output Formats	1007
Country-specific and User-specific Output Formats	1008
Currency-specific Output Formats	1010
Unit-specific Output Formats	1011
Creating Blank Lines	1012
Drawing Lines, Frames, and Grids	1014
Straight Lines	1015
Corners	1016
T Sections	1017
Crosses	1018
Using Special Lines	1019
Programming Frames	1022
Programming Grids.....	1023
Determining Which Part of a Page to Scroll Horizontally.....	1025
Excluding Lines from Horizontal Scrolling	1026
Left Boundary for Horizontal Scrolling	1028
Interactive Lists	1030
What is Interactive Reporting?	1031
Event Control for Interactive Lists	1033
Basic Lists and Secondary Lists.....	1035
Creating the Basic List	1036
Creating Secondary Lists	1037
Maintaining Lists.....	1039
System Fields for Secondary Lists.....	1040

Page Headers for Secondary Lists	1041
Messages in Lists.....	1043
User Interfaces of Interactive Lists	1046
Allowing Line Selection	1047
Allowing Function Key Selection	1049
Defining Individual User Interfaces	1051
Defining a Status for Interactive Lists	1052
Starting the Menu Painter Tool for Interactive Lists.....	1053
Using the Menu Painter for Interactive Lists	1055
Defining Titles for Interactive Lists	1061
Using Your Own Function Codes in the Program.....	1063
Setting a Status.....	1064
The AT USER-COMMAND Event.....	1067
Displaying Lists in Dialog Windows.....	1070
Triggering Events from within the Program.....	1074
Passing Data from List to Report.....	1076
Passing Data Automatically.....	1077
Data from System Fields of Interactive Lists.....	1078
Using SY-LISEL	1080
Passing Data by Program Statements	1082
The HIDE Technique.....	1083
Reading Lines from Lists	1087
Reading Lists at the Cursor Position.....	1091
Passing List Attributes.....	1094
Manipulating Interactive Lists	1097
Scrolling through Interactive Lists	1098
Setting the Cursor from within the Program	1100
Setting the Cursor Explicitly	1101
Setting the Cursor to a Field	1103
Setting the Cursor to a Line	1105
Modifying List Lines	1107
Modifying Line Formatting.....	1108
Modifying Field Contents.....	1109
Modifying Field Formatting.....	1110
Calling Programs	1112
Calling Executable Programs (Reports).....	1113
Exiting the Executable Program Called from Within the Program	1114
Manipulating the List Structure of the Called Executable Program (Report)	1116
Filling the Selection Screen of the Called Executable Program (Report)	1118
Calling Transactions.....	1122
Transferring SPA/GPA Parameters to Transactions	1123
Printing Lists	1126
Printing a List after Creating it	1127
Printing a List while Creating it	1129
Print Parameters	1130
Print Parameters- Overview	1131

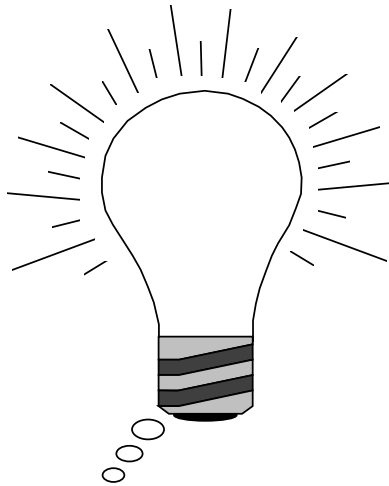
Print Parameters -Pre-setting Values	1135
Setting Print Parameters from within the Program	1137
GET_PRINT_PARAMETERS - Overview.....	1138
Import Parameters of GET_PRINT_PARAMETERS.....	1139
Export Parameters of GET_PRINT_PARAMETERS.....	1140
Exception Parameters of GET_PRINT_PARAMETERS	1141
How to Use GET_PRINT_PARAMETERS	1142
Executing and Printing	1143
Printing from within the Program	1146
Printing Lists of Called Executable Programs (Reports).....	1150
Print Control.....	1152
Determining Left and Upper Margins	1153
Determining the Print Format	1155
Indexing Print Lists for Optical Archiving	1159
Switching Between Dialog Screens and List Display	1162
Using LEAVE TO LIST-PROCESSING	1163
Using GUI Statuses in List-Mode.....	1165
Returning to Dialog Mode	1167
Returning to a Different Screen	1168
Example Transaction: Branching to List-Mode.....	1169
Messages	1171
Handling Errors and Messages	1172
Introduction to Error Processing.....	1173
Checking Screen Fields for Validity.....	1176
Understanding Automatic Field Checks	1177
Checking Fields in the Screen Flow Logic	1178
Checking Fields in ABAP	1179
Checking a Single Field	1180
Checking Multiple Fields	1181
Making Module Calls Conditionally.....	1182
Conditional FIELD Statements	1183
Conditional CHAIN Statements	1184
Avoiding Automatic Field Checks.....	1185
Issuing Messages	1186
Sending a Message.....	1187
Message Processing in Dialog-Mode	1188
Restarting PAI after an error dialog	1189
Creating a Message Class	1190
Creating Messages	1191
Example Transaction: Checking Field Input.....	1192
Messages on Selection Screens	1194
Messages in Lists	1195
Running ABAP Programs	1198
Directly Executable Programs	1199
Executable Programs (Reports) and Logical Databases	1200
Accessing Data with SELECT	1201
Accessing Data Using Logical Databases.....	1202

Comparison of Access Methods.....	1204
Advantages of Logical Databases.....	1206
Controlling the Database Accesses from the Executable program (Report)	1207
Controlling the Flow of ABAP Programs by Events.....	1208
Defining Processing Blocks.....	1209
Events and their Event Keywords	1211
INITIALIZATION.....	1213
AT SELECTION-SCREEN	1216
Processing a Particular Input Field	1218
Processing Multiple Selection	1219
Creating a List of Input Values.....	1220
Creating Help for Input Fields	1221
Processing a Group of Radio Buttons.....	1222
Processing a Block of Input Fields.....	1223
PBO of the Selection Screen	1224
START-OF-SELECTION.....	1225
GET <table>	1226
Specifying Fields of the Database Table Explicitly.....	1229
GET <table> LATE	1230
END-OF-SELECTION	1232
Terminating Processing Blocks	1233
Leaving Processing Blocks Unconditionally.....	1234
Branching to END-OF-SELECTION.....	1235
Branching to the Output Screen.....	1236
Leaving AT Events	1237
Leaving Processing Blocks Conditionally.....	1238
Leaving GET Events Unconditionally.....	1240
Branching to the Next Line of the Current Database Table	1241
Branching to the Next Line of a Superior Database Table.....	1243
Leaving GET Events Conditionally.....	1244
Features and Maintenance of Logical Databases	1246
Features of Logical Databases.....	1247
Tasks of Logical Databases	1248
Basic Features of Logical Databases.....	1249
Structure of Logical Databases	1251
Logical Database Selections.....	1252
Database Program of a Logical Database	1254
Logical Databases and Reports	1258
Authorization Checks with Logical Databases	1262
Performance Aspects of Logical Databases	1263
Example of a Logical Database	1265
Creating and Maintaining Logical Databases.....	1269
Creating a Logical Database	1270
Processing the Structure	1273
Displaying Structures	1274
Changing Structures.....	1276

Editing Selections.....	1278
Editing the Database Program	1281
Working with Dynamic Selections.....	1287
Working with Field Selections	1291
Editing Selection Texts.....	1294
Editing Matchcode Selections	1297
Editing Documentation	1301
Further Editing Options	1302
Editing the Data Model.....	1303
Checking Logical Databases.....	1305
Copying Logical Databases.....	1306
Deleting Logical Databases.....	1307
Dialog Mode.....	1308
Dialog Mode.....	1309
Dialog Mode Programs - Transactions.....	1310
A Sample Program	1313
Screen	1315
ABAP Module Pool.....	1317
Interaction between Dialog Screen and ABAP Module Pool	1320
Calling Programs Externally	1322
Calling External Program Components	1323
Embedding Program Calls.....	1324
Calling Function Modules	1325
Handling Exceptions.....	1328
Calling Other Transactions.....	1330
Calling Dialog Modules.....	1331
Using Transactions as Dialog Modules	1332
Submitting Executable Programs (Reports).....	1334
Passing Data to the Called Program	1335
Saving or Printing the List	1337
Passing Data Between Programs	1338
Passing Data with SPA/GPA Parameters.....	1339
Syntax Conventions	1340

BC ABAP User's Guide

ABAP Basics



The R/3 Basis System: Overview

The R/3 Basis System: Overview

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

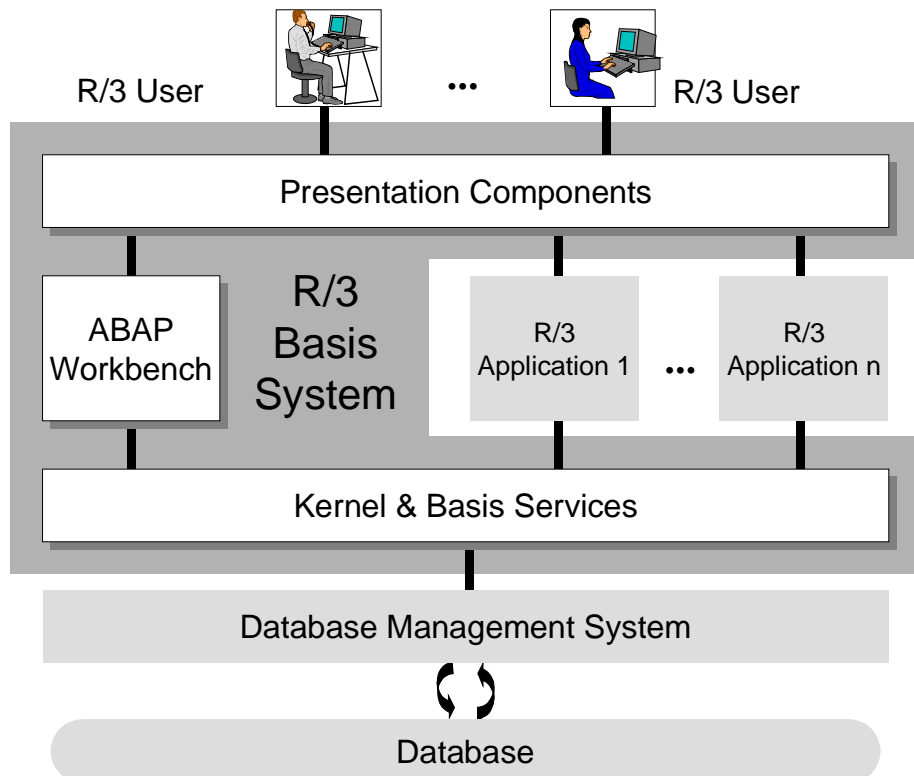
This documentation explains just what the Basis system is, and how it ties in with the R/3 System as a whole. It starts by introducing the Basis system in general. The second part concentrates on one central component - the application server. Finally, it will explain about work processes, which are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

Position of the Basis System Within the R/3 System

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

Logical View

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



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

Kernel and Basis Services

The kernel and basis services component is a runtime environment for all R/3 applications that is hardware-, operating system- and database-specific. It is written principally in C and C++,

The R/3 Basis System: Overview

although some low-level parts are written in SAP's own programming language. The tasks of the kernel and basis services component are as follows:

- **Running applications**
All R/3 applications run on software processors (virtual machines) within this component.
- **User and process administration**
An R/3 System is a multi-user environment, and each user can run several independent applications. In short, this component is responsible for the tasks that usually belong to an operating system. Users log onto the R/3 System and run applications without ever coming into contact with the operating system of the host on which it is running. The R/3 System is the only user of the host operating system.
- **Database access**
Each R/3 System is linked to a database system, consisting of a database management system (DBMS) and the database itself. The applications do not communicate directly with the database. Instead, they use Basis services.
- **Communication**
R/3 applications can communicate with other R/3 Systems and with non-SAP systems. It is also possible to access R/3 applications from external systems using a BAPI interface. The services required for communication are all part of the kernel and basis services component.
- **System Monitoring and Administration**
The component contains programs that allow you to monitor and control the R/3 System while it is running, and to change its runtime parameters.

ABAP Workbench

The ABAP Workbench component is a fully-fledged **development environment** for creating, enhancing, testing, and organizing applications in the ABAP language. It is fully integrated in the R/3 Basis system and, like other R/3 applications, is itself written in ABAP.

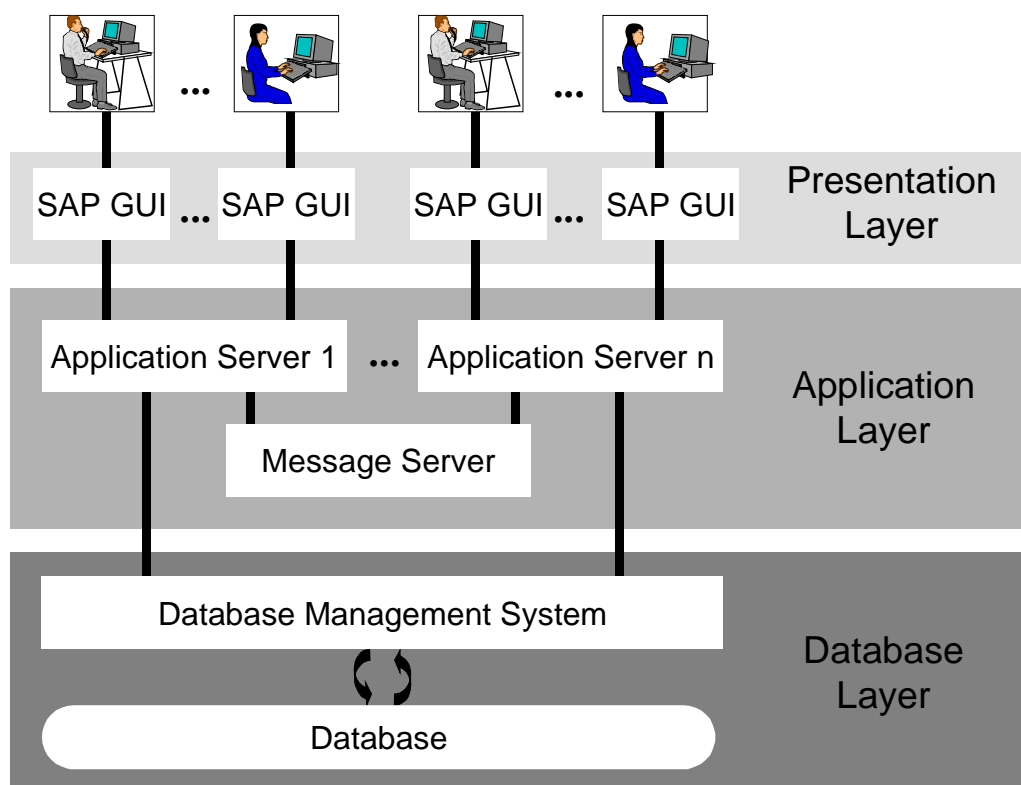
Presentation Components

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

Software-oriented View

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

The R/3 Basis System: Overview



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

Database Layer

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

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

Application Layer

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

The R/3 Basis System: Overview

servers. It passes requests from one application server to another within the system. It also contains information about application server groups and the current load balancing within them. It uses this information to choose an appropriate server when a user logs onto the system.

Presentation Layer

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

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

Software-oriented and Hardware-oriented View

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

Advantages of the Multi-tier Architecture

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

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

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

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

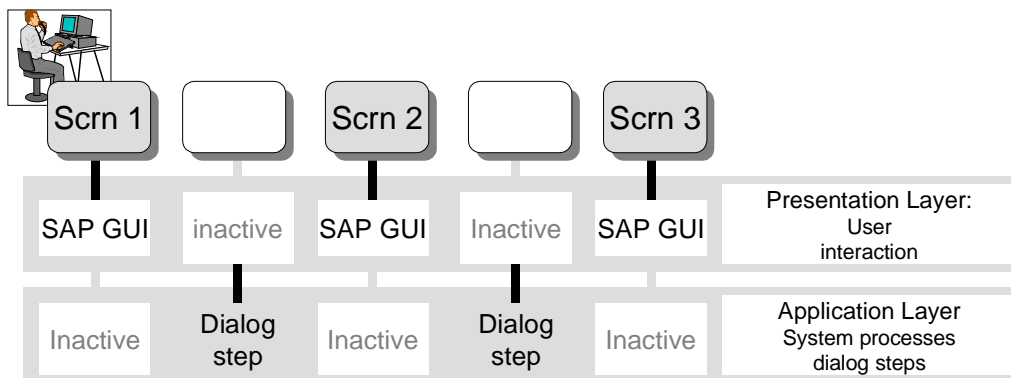
Consequences for Application Programming

The fact that the application and presentation layers are separate carries an important consequence for application programmers. When you run an application program that requires user interaction, control of the program is continually passed backwards and forwards between the layers. When a screen is ready for user input, the presentation layer is active, and the

The R/3 Basis System: Overview

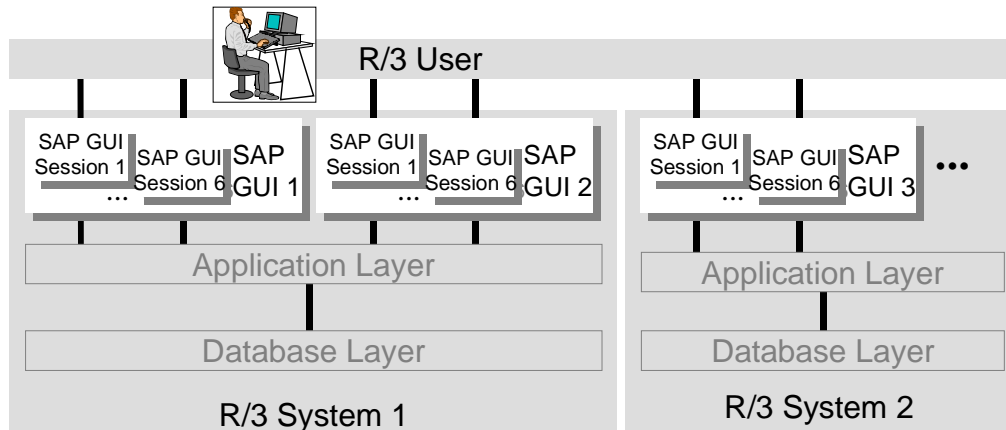
application server is inactive with regard to that particular program, but free for other tasks. Once the user has entered data on the screen, program control passes back to the application layer. Now, the presentation layer is inactive. The SAPgui is still visible to the user during this time, and it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

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



User-oriented View

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



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

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

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

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

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

Application Servers

R/3 programs run on application servers. This section takes a look at these important software components within the R/3 System.

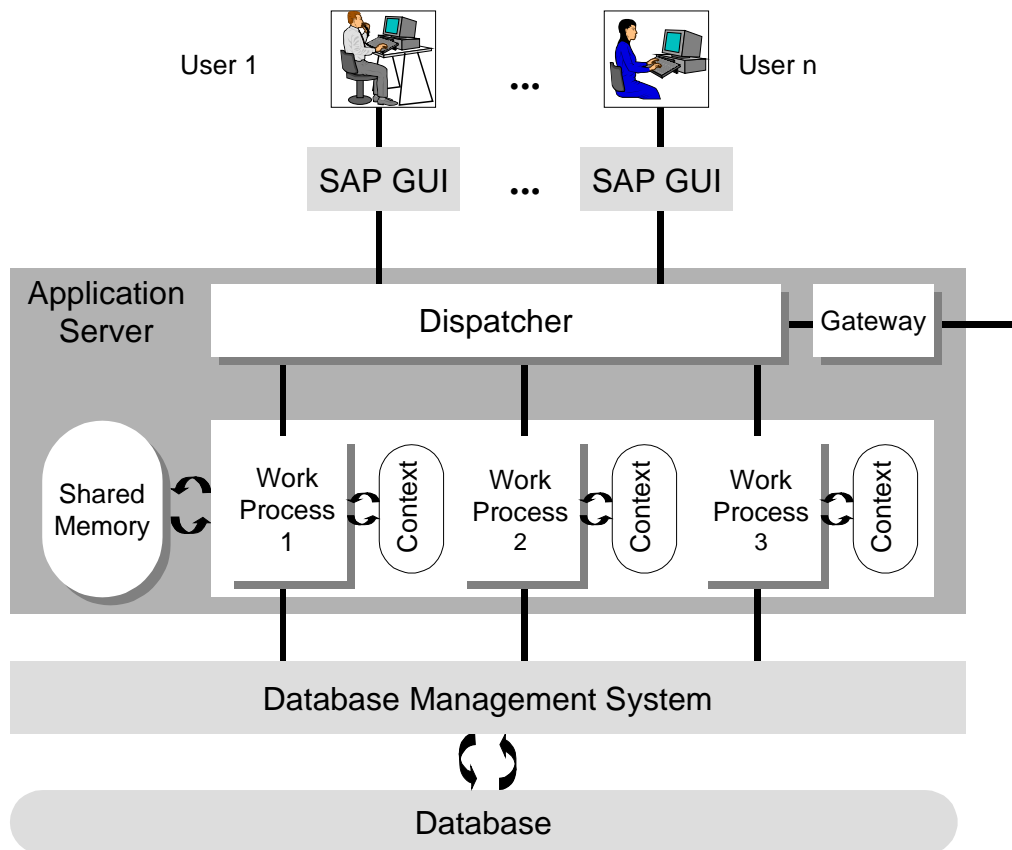
Structure of an Application Server

The application layer of an R/3 System is made up of the application servers and the message server. Application programs in an R/3 System are run on application servers. The application

User-oriented View

servers communicate with the presentation components, the database, and also with each other, using the message server.

The following diagram shows the structure of an application server:



Work Processes

An application server contains work processes, which are components that can run an application (or a dialog step). The numbers and types of work processes are set when the R/3 System is started. Each work process is linked to a memory area containing the context of the application being run. The context contains the current data for the application program. This needs to be available in each dialog step. Further information about the different types of work process is contained later on in this documentation.

Dispatcher

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

Gateway

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

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

Shared Memory

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

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

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

Database Connection

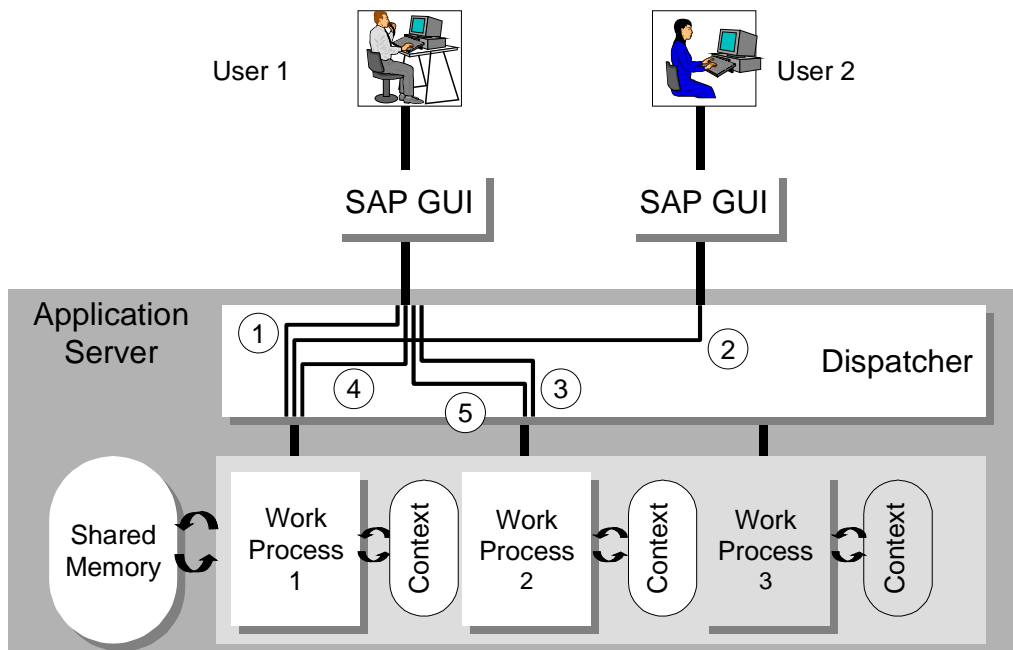
When you start up an R/3 System, each application server registers its work processes with the database layer, and receives a single dedicated channel for each. While the system is running, each work process is a user (client) of the database system (server). You cannot change the work process registration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

Dispatching Dialog Steps

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

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

User-oriented View



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

From this example, we can see that:

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

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

Dispatching and the Programming Model

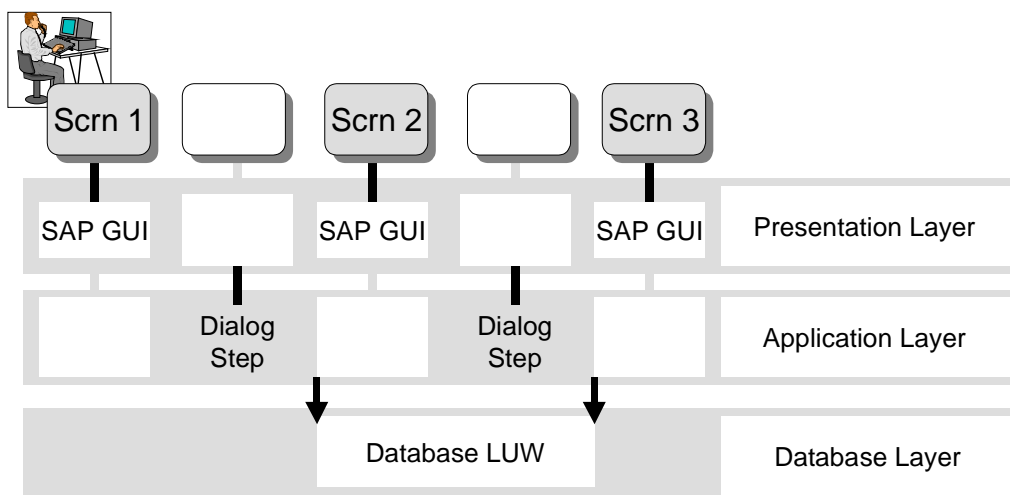
The separation of application and presentation layer made it necessary to split up application programs into dialog steps. This, and the fact that dialog steps are dispatched to individual work processes, has had important consequences for the programming model.

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

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

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

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



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

User-oriented View

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

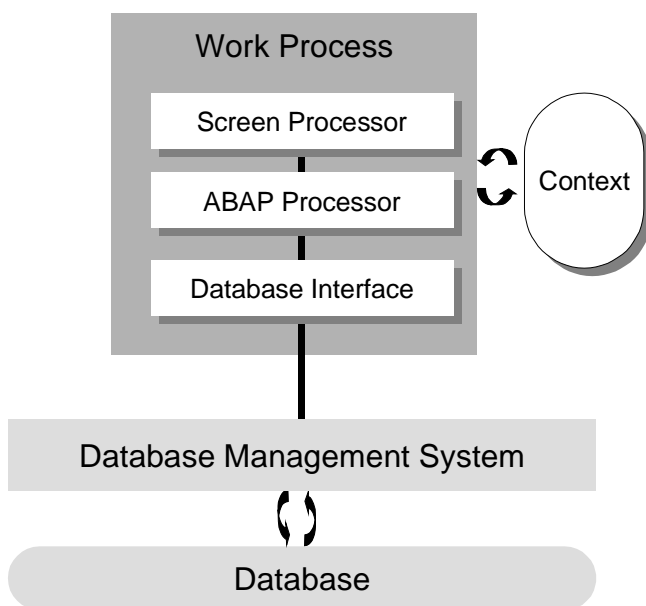
The SAP programming model contains a series of bundling techniques that allow you to group database updates together in logical units. The section of an R/3 application program that bundles a set of logically-associated database operations is called an **SAP LUW**. Unlike a database LUW, a SAP LUW includes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

Work Processes

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

Structure of a Work Process

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



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

Screen Processor

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

ABAP-Processor

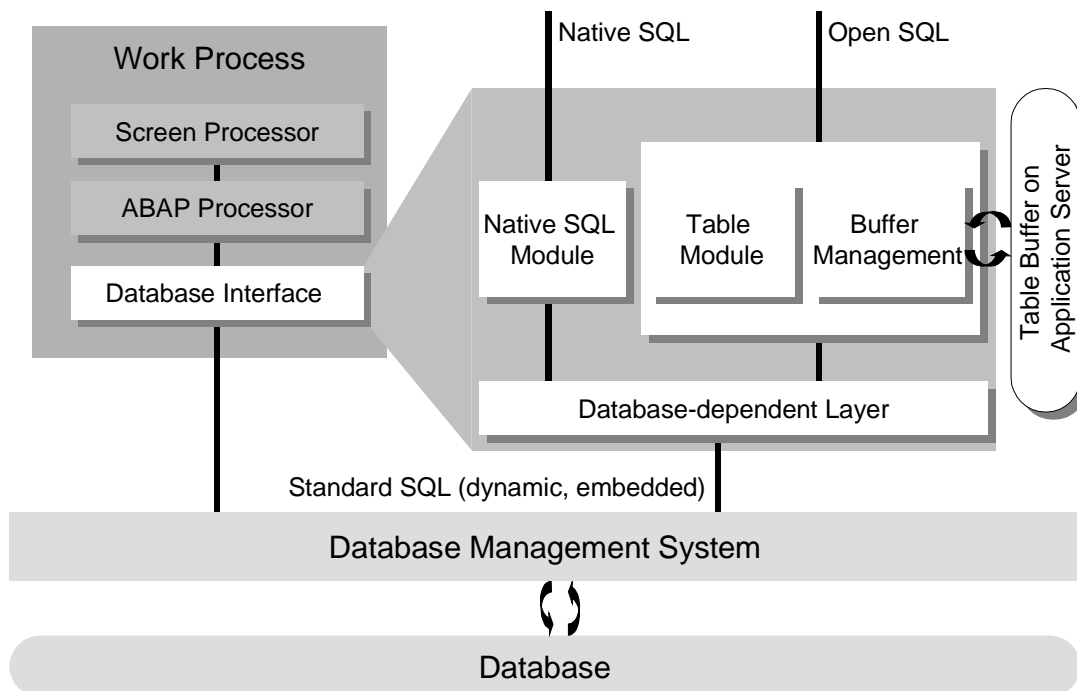
The actual processing logic of an application program is written in **ABAP** - SAP's own programming language. The ABAP processor executes the **processing logic** of the application program, and communicates with the database interface. The screen processor tells the ABAP processor which module of the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

Database Interface

The database interface provides the following services:

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

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



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

Open SQL statements are a subset of Standard SQL that is fully integrated in ABAP. They allow you to access data irrespective of the database system that the R/3 installation is using. Open SQL consists of the Data Manipulation Language (DML) part of Standard SQL; in other words, it allows you to read (SELECT) and change (INSERT, UPDATE, DELETE) data. The tasks of the

User-oriented View

Data Definition Language (DDL) and Data Control Language (DCL) parts of Standard SQL are performed in the R/3 System by the ABAP Dictionary and the authorization system. These provide a unified range of functions, irrespective of database, and also contain functions beyond those offered by the various database systems.

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

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

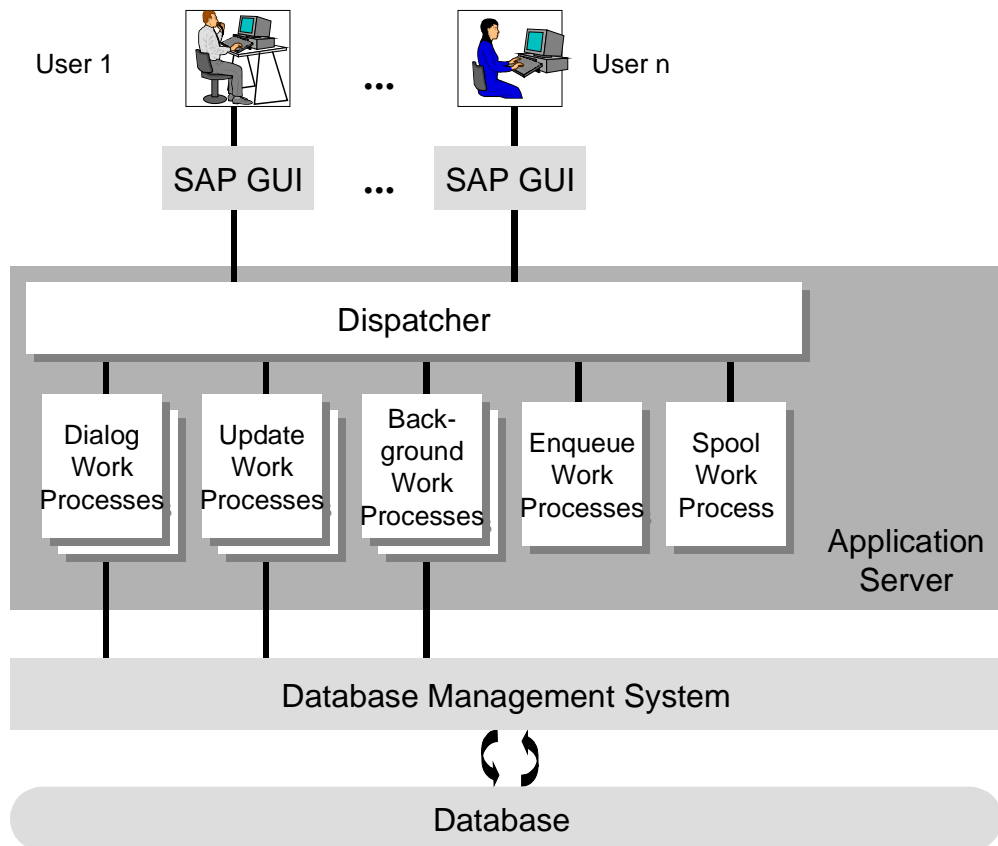
The database-dependent layer in the diagram serves to hide the differences between database systems from the rest of the database interface. You choose the appropriate layer when you install the Basis system. Thanks to the standardization of SQL, the differences in the syntax of statements are very slight.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

Types of Work Process

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

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

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



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

Dialog Work Process

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

Update Work Process

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

Background Work Process

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

Enqueue Work Process

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

Spool Work Process

User-oriented View

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

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

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

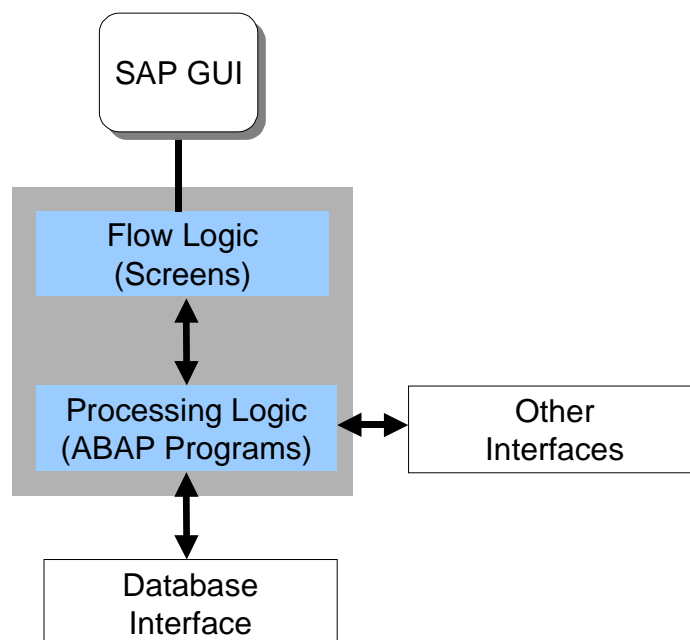
Overview of the Components of Application Programs

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

Structure of an Application Program

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

As described in the [Overview of the R/3 Basis System \[Page 23\]](#), a work process contains a screen processor for processing user input, an ABAP processor for processing the program logic, and a database interface for communicating with the database. These components of a work process determine the following structure of an application program:



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

Flow Logic

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

Processing logic

Overview of the Components of Application Programs

The components of application programs that are responsible for data processing in the R/3 System are ABAP programs. ABAP stands for 'Advanced Business Application Programming'. ABAP programs run on the ABAP processor of a work process. They receive screen input from the screen processor and send it to the screen processor. You access the database using the database interface. ABAP contains a special set of commands called OPEN SQL. This allows you to read from and write to the database regardless of the database you are using. The database interface converts the OPEN SQL commands into commands of the relevant database. You can also use native SQL commands, which are passed to the database without first being converted. There is a range of further interfaces such as memory, sequential files and external interfaces. These provide other means of sending data to and from ABAP programs. When working together with screens, ABAP programs play a more passive role, acting as a container for a set of modules that can be called from the flow logic.

Screens

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

Screens

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

Selection Screens and Lists

There are two special kinds of screen in the R/3 System that you will often use - selection screens and lists. In these cases, you do not create the screen or its flow logic using the **Screen Painter**. Instead you use ABAP statements in the **processing logic**.

Screens in the R/3 System also contain a menu bar, a standard toolbar, and an application toolbar. Together, these three objects form the **status** of the screen. A status is a development object belonging to its corresponding application program. It is independent of the screen, and can therefore be used in conjunction with several different screens. You define statuses using the **Menu Painter** in the ABAP Workbench. You can define statuses yourself for screens and lists, but there is a predefined status that the system always uses for selection screens.

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

Screens

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

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

The screen language is a special subset of ABAP, and contains only a few keywords. The statements are syntactically similar to the other ABAP statements, but you may not use screen statements in ABAP programs or ABAP statements in the screen flow logic. The most important screen keywords are MODULE, FIELD, CHAIN, and LOOP. Their only function is to link the

Overview of the Components of Application Programs

processing logic to the flow logic, that is, to call modules in the processing logic, and control data transfer between the screen and the ABAP program, for example, by checking fields.

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

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

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

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

Dialog screens enable the user and application programs to communicate with each other. They are used in dialog-oriented programs such as transactions, where the program consists mainly of processing a sequence of screens, and are particularly useful when specialized screens (selection screens and lists) are insufficient for your requirements.

Selection Screens

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

Overview of the Components of Application Programs

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

The screenshot shows a selection screen with the following components:

- Selection options** and **Free selections** tabs.
- Airline carrier**: A dropdown menu followed by a "to" field.
- Search help selection**: A section containing three input fields: "Search help Id", "Search string", and "Complex Search help".
- From** and **To**: Two date input fields.
- Departure date**: An input field followed by a "to" field.
- Booking number**: An input field followed by a "to" field.

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

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

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

Lists

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

Overview of the Components of Application Programs

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

Open items as of 12/07/95					
Open Due					
CC	BA	Total	up to 8 days	p to 30 days	than 30 days
0001	0000	4.368,00	4.192,00	169,00	7,00
0001	0001	3.528,00	3.473,00	29,00	26,00
0001	0002	1.235,00	982,00	234,00	19,00
0001	0003	128,00	67,00	52,00	9,00
0001	****	9.259,00	8.714,00	484,00	61,00
0002	0000	922,00	909,00	4,00	9,00
0002	0001	2.098,00	1.746,00	325,00	27,00
0002	0002	5.076,00	4.608,00	434,00	34,00
0002	****	8.096,00	7.263,00	763,00	70,00
0003	0000	4.041,00	3.789,00	209,00	43,00
0003	0001	2.196,00	1.867,00	279,00	50,00
0003	0002	466,00	175,00	258,00	33,00
0003	0003	2.484,00	2.404,00	42,00	38,00

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

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

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

Processing logic

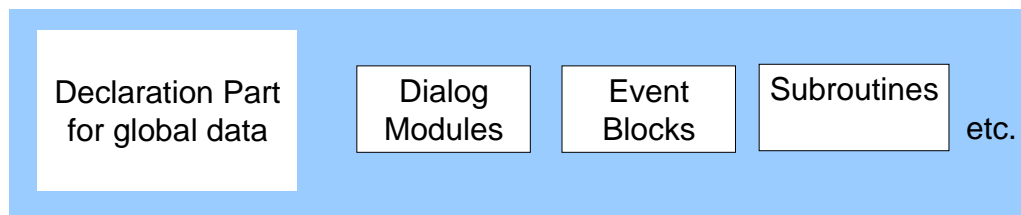
ABAP processing logic is responsible for processing data in R/3 application programs. ABAP was designed specifically for dialog-oriented database applications. The following sections deal with how an ABAP program is structured and executed.

Program Structure

Overview of the Components of Application Programs

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

The following diagram shows the structure of an ABAP program:



Each ABAP program consists of the following two parts:

Declaration Part for Global Data, Classes and Selection Screens

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

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

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

Container for Processing Blocks

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

- Dialog modules (no local data area)
- Event blocks (no local data area)
- Routines (methods, subroutines and function modules with their own local data area).

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

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

Overview of the Components of Application Programs

Calling Processing Blocks

You can call processing blocks either from outside the ABAP program or using ABAP commands which are themselves part of a processing block. Calling event blocks is different from calling other processing blocks for the following reasons:

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

Program Types and Execution

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

When you run an application program, you must call at least the first processing block from outside the program, either using screen flow logic or an external programming interface (like RFC). This processing block can then either call further processing blocks or return control to outside the program.

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

The following program types are relevant to application programming:

Type 1

Type 1 programs have the important characteristic that they do not have to be controlled using screens. Instead, they are controlled by an invisible system program that calls a series of processing blocks (and selection screens and lists where necessary) in a fixed sequence. Further processing blocks can be triggered by user actions on the screen.

You can start a type 1 program and the invisible system program using the SUBMIT statement in another ABAP program. There are also various ways of starting a type 1 program by entering its program name. This is why we refer to type 1 programs as executable programs.

The invisible system program provides a predefined program flow, which starts with a selection screen (for data entry), processes the data, and then displays the results in a list without the programmer having to define any screens his- or herself. The system program also allows you to work with a logical database. A logical database is a special ABAP program which combines the contents of certain database tables. The flow of the system program is oriented towards reporting, whose main tasks are to read data from the database, process it, and display the results. This is why executable programs (type 1) in the R/3 System are often referred to as **reports**, and why running an executable program is often called **reporting**.

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

Overview of the Components of Application Programs

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

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

It is appropriate to use executable programs (reports) when the flow of your program corresponds either wholly or in part to the pre-defined flow of the system program. If you want to use a logical database, you must use an executable program (report).

Type M

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

When you start a program using its transaction code, the initial screen is called. This calls a dialog module belonging to the associated ABAP program. The remainder of the program flow can take any form. For example, the dialog module can:

- return control to the screen, after which, the processing passes to a subsequent screen in a fixed sequence.
- call other dialog screens, selection screens or lists, from which further processing blocks in the ABAP program are started.
- call other processing blocks itself, either internally or externally.
- call other application programs using CALL TRANSACTION (type M program) or SUBMIT (type 1 program).

ABAP programs with type M contain the dialog modules belonging to the various screens. They are therefore known as **module pools**. It is appropriate to use module pools when you write dialog-oriented programs using a large number of screens whose flow logic largely determines the program flow.

Type F

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

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

Type I

Type I programs - called **includes** - are a means of dividing up program code into smaller, more manageable units. You can insert the coding of an include program at any point in another ABAP program using the INCLUDE statement. There is no technical relationship between include programs and processing blocks. Includes are more suitable for logical programming units, such

Overview of the Components of Application Programs

as data declarations, or sets of similar processing blocks. The ABAP Workbench has a mechanism for automatically dividing up module pools and function groups into include programs.

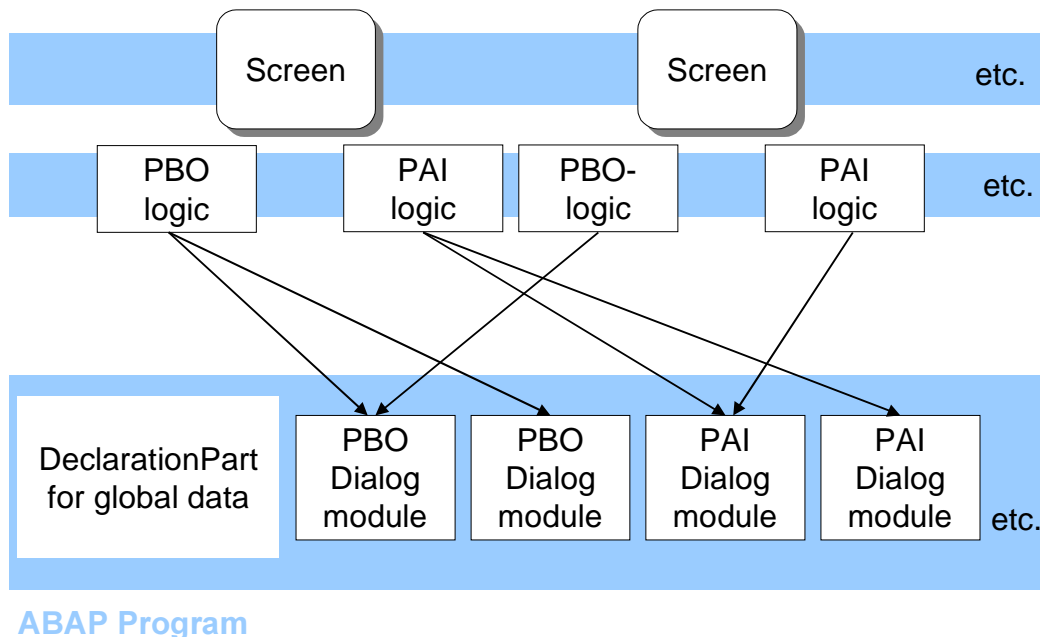
Processing Blocks

The following sections describe the various processing blocks that you can use in ABAP programs. There are also further processing blocks that occur in ABAP Objects, the object-oriented extension of ABAP.

Dialog Modules

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

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

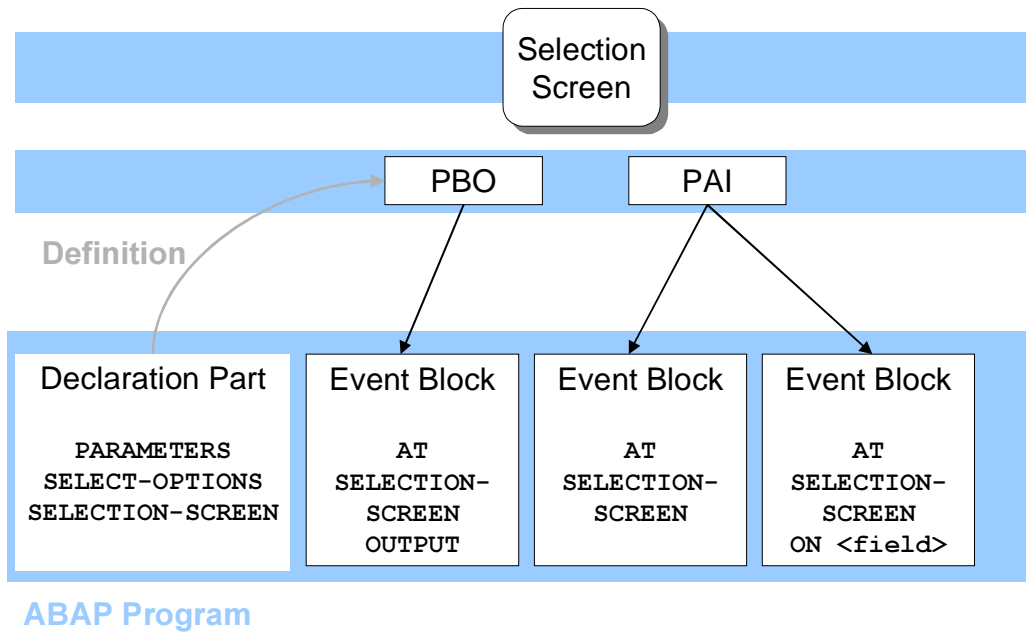


Fields on a dialog screen have the same name as a corresponding field in the ABAP program. Data is passed between identically-named fields in the program. You do not define dialog screens in the ABAP programs.

Event Blocks for Selection Screens

A selection screen is a special kind of dialog screen that you create using ABAP commands. Particular events correspond to the different states of the selection screen (PAI, PBO, user input), and processing blocks can be assigned to them in the ABAP program.

Overview of the Components of Application Programs

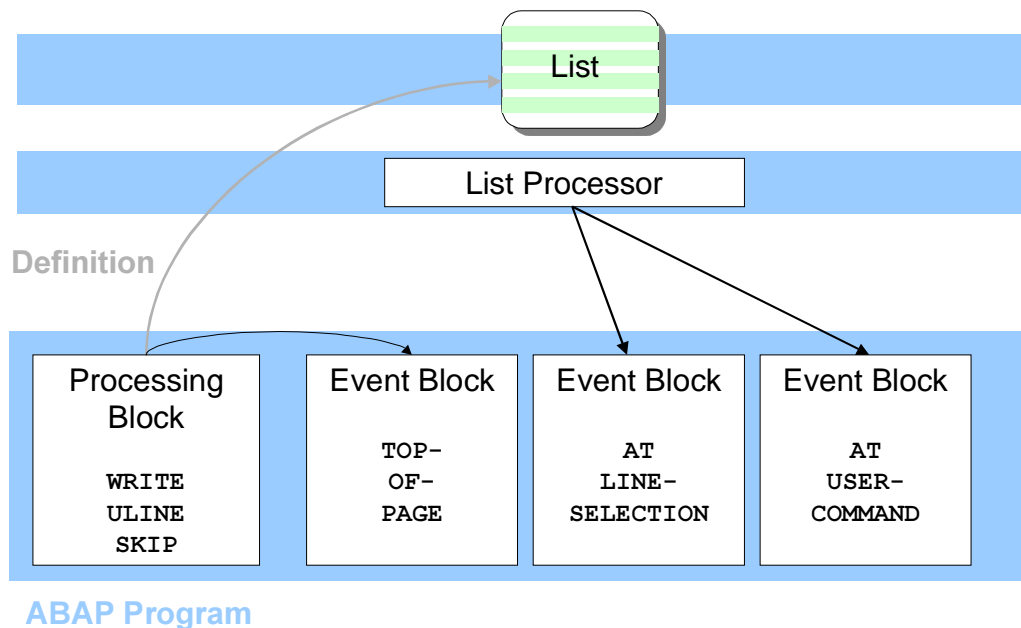


You therefore do not need to create dialog modules for the PBO and PAI events of a selection screen. Data is passed between selection screen and ABAP program using the fields (parameters and selection tables) which you create in the selection screen definition in the declaration part of the ABAP program.

Event Blocks for Lists

Lists are special screens which output formatted data. You can create them in any processing block in an ABAP program using a special set of commands (such as WRITE, NEW-PAGE and so on). An internal program called the list processor displays the list on the screen and handles user actions within lists.

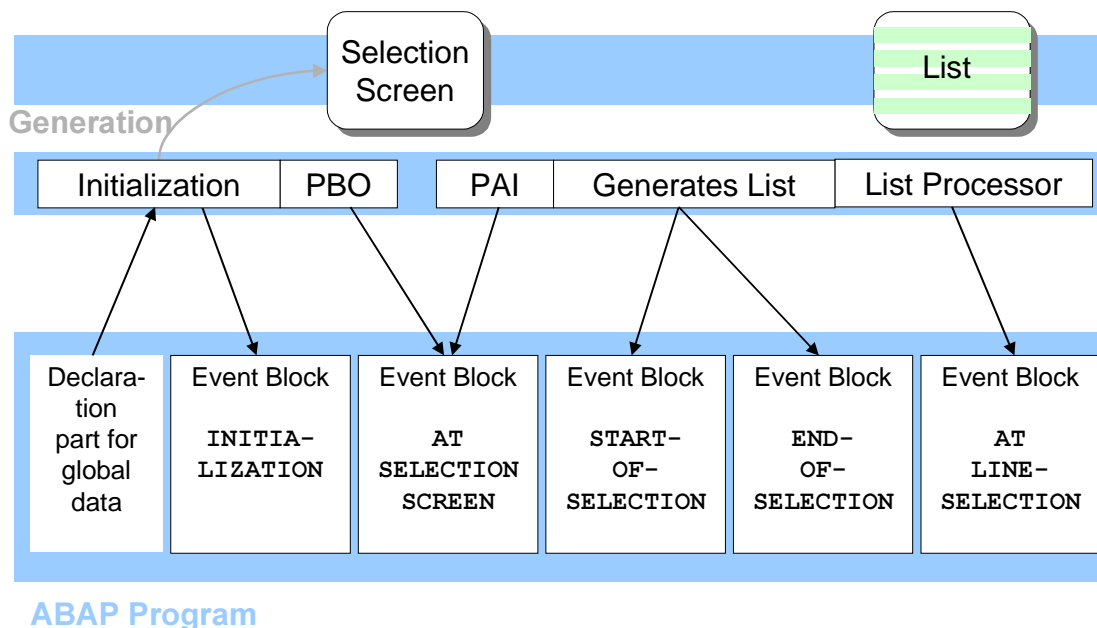
Overview of the Components of Application Programs



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

Event Blocks for Executable Programs (Reports)

When you run an executable (type 1) program, an invisible system program calls a series of event blocks in a fixed order:



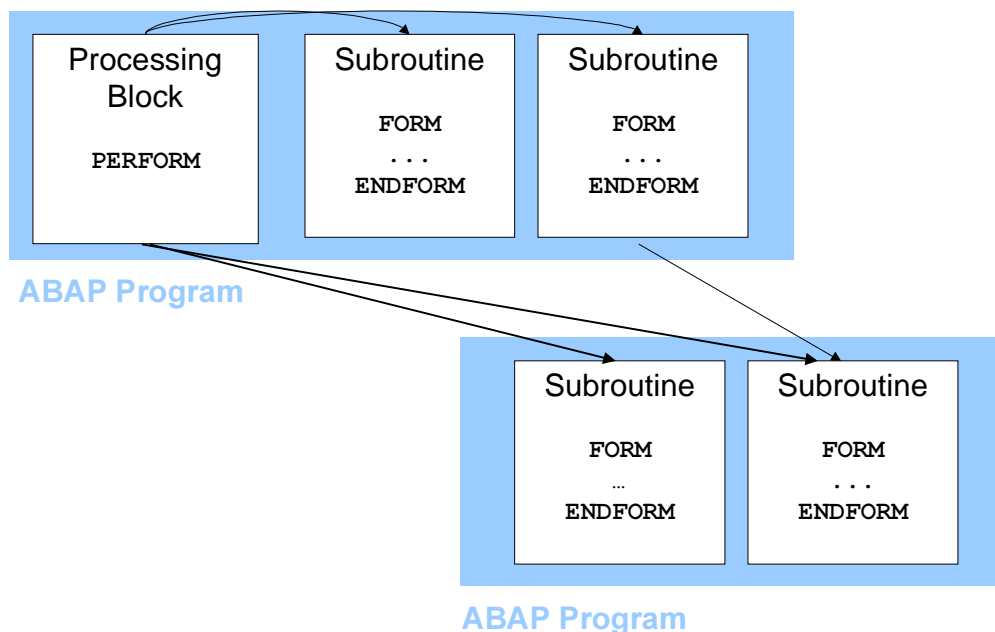
Overview of the Components of Application Programs

1. ? If you have defined a selection screen in the declaration part of the ABAP program, the system program generates it in an initialization part of the program.
2. ? It creates the INITIALIZATION event and calls the corresponding event block (if it has been defined in the ABAP program).
3. ? It passes control to the selection screen. The selection screen generates its own events, and calls the corresponding event blocks. At the end of the PAI of the selection screen, the list generation part of the system program takes control.
4. ? It creates the START-OF-SELECTION event and calls the corresponding event block (if it has been defined in the ABAP program).
5. ? The logical database, if you are using one, calls further event blocks at this point.
6. ? It creates the END-OF-SELECTION event and calls the corresponding event block (if it has been defined in the ABAP program).
7. ? It then passes control to an internal system program called the list processor. The list processor displays the list defined in the ABAP program. User actions in the list generate events themselves, and the corresponding event blocks are called.

Subroutines

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

You call them from ABAP programs using the PERFORM statement.



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

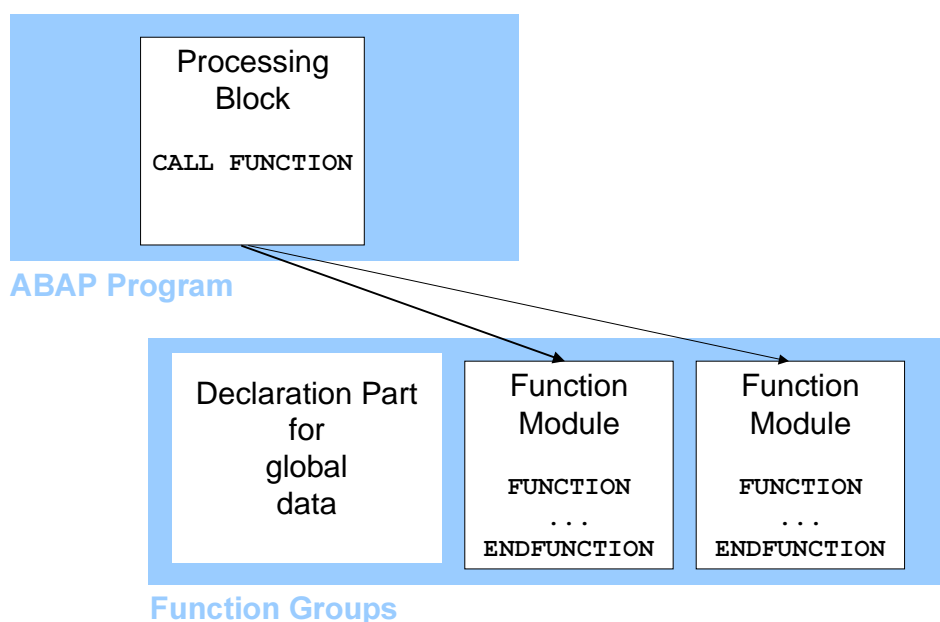
Overview of the Components of Application Programs

program and the subroutine. When you call an external subroutine, you must pass actual parameters from the main program to formal parameters in the subroutine.

Function Modules

Function modules are special external functions with a fixed interface. Function modules are introduced with the `FUNCTION` statement and concluded with the `ENDFUNCTION` statement. You define them in the ABAP Workbench using the Function Builder. Each function module belongs to a function group.

You call function modules from ABAP programs using the `CALL FUNCTION` statement.



Unlike subroutines, function modules have a pre-defined parameter interface. You can **only** call function modules externally using `CALL FUNCTION`.

Logical Databases

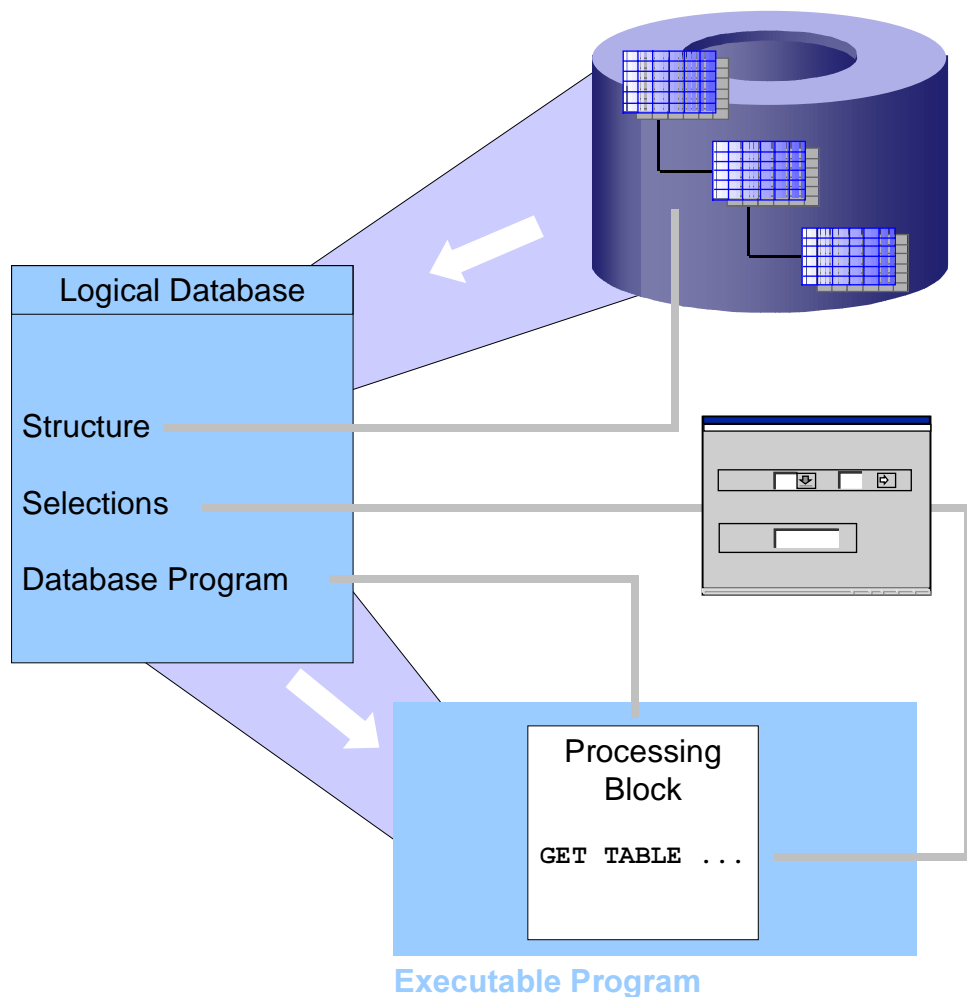
Logical databases are special ABAP programs that read data from database tables. They are used by executable (type 1) programs. At runtime, you can regard the logical database and the executable program (reports) as a single ABAP program, whose processing blocks are called by the system program in a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

You edit logical databases using a tool within the ABAP Workbench, and link them to executable programs (reports) when you enter the program attributes. You can use a logical database with any number of executable programs (reports). However, they are only auxiliary programs, and can only be used in conjunction with executable programs.

Structure of a Logical Database

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

Overview of the Components of Application Programs



Structure

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

Selection Part

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

Database Program

The database program of a logical database is a container for special subroutines, in which the data is read from the database tables. The system program, which controls the flow of an executable program, calls these subroutines in the order determined by the structure of the logical database.

Overview of the Components of Application Programs

Running Programs with a Logical Database

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

The system program calls depend both on the structure of the logical database and on the definition of the executable program. The structure of the logical database determines the sequence in which the processing blocks of the logical database are called. These in turn call GET event blocks in the executable program. These GET event blocks determine the read depth in the structure of the logical database. The TABLES statement in the declaration part of the executable program determines which of the input fields defined in the logical database will be displayed on the screen.

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

Uses of Logical Databases

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

ABAP Statements

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

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

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

ABAP statements use ABAP data types and objects.

Statements and Keywords

The first element of an ABAP statement is the ABAP keyword. This determines the statement category, which is one of the following:

Declarative Statements

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

Examples of declarative keywords:

TYPES, DATA, TABLES

Modularization Statements

These statements define the processing blocks in an ABAP program.

Overview of the Components of Application Programs

The modularization keywords can be further divided into:

- **Event Keywords**

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

Examples of event keywords are:

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

- **Defining keywords**

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

Examples of definitive keywords:

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

Control Statements

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

Examples of control keywords:

IF, WHILE, CASE

Call Statements

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

Examples of call keywords:

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

Operational Statements

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

Examples of operational keywords:

WRITE, MOVE, ADD

Database Statements

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

Open SQL

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

Overview of the Components of Application Programs

access the database are fully portable. The database interface converts the OPEN SQL commands into commands of the relevant database.

Native SQL

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

Data Types and Objects

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

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

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

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

Predefined and User-defined Elementary Types

There are five predefined non-numeric data types:

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

and three predefined numeric types:

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

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

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

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

Overview of the Components of Application Programs

You use elementary data types to define individual elementary data objects. You use these object to transfer input and output values, as auxiliary fields in calculations, to store intermediate results, and so on. As well as their function as individual objects, data types are also the smallest components of structured data types.

A structured data type can be a **structure** or an **internal table**.

Structures

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

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

Internal Tables

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

Internal tables are characterized by:

- Their line type.

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

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

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

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

Declaring Data Objects

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

When you declare an internal table, you specify the above details. However, you do not need to specify the overall size of the data object. Only the length of a row in an internal table is fixed.

Overview of the Components of Application Programs

The number of rows (the actual length of the data object in memory) is adapted dynamically at runtime. In short, internal tables can be extended dynamically while retaining a fixed structure.

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

Memory Structures of an ABAP Program

In the [Overview of the R/3 Basis System \[Page 23\]](#) you have seen that each user can open up to six R/3 windows in a single SAPgui session. Each of these windows corresponds to a session on the application server with its own area of shared memory.

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

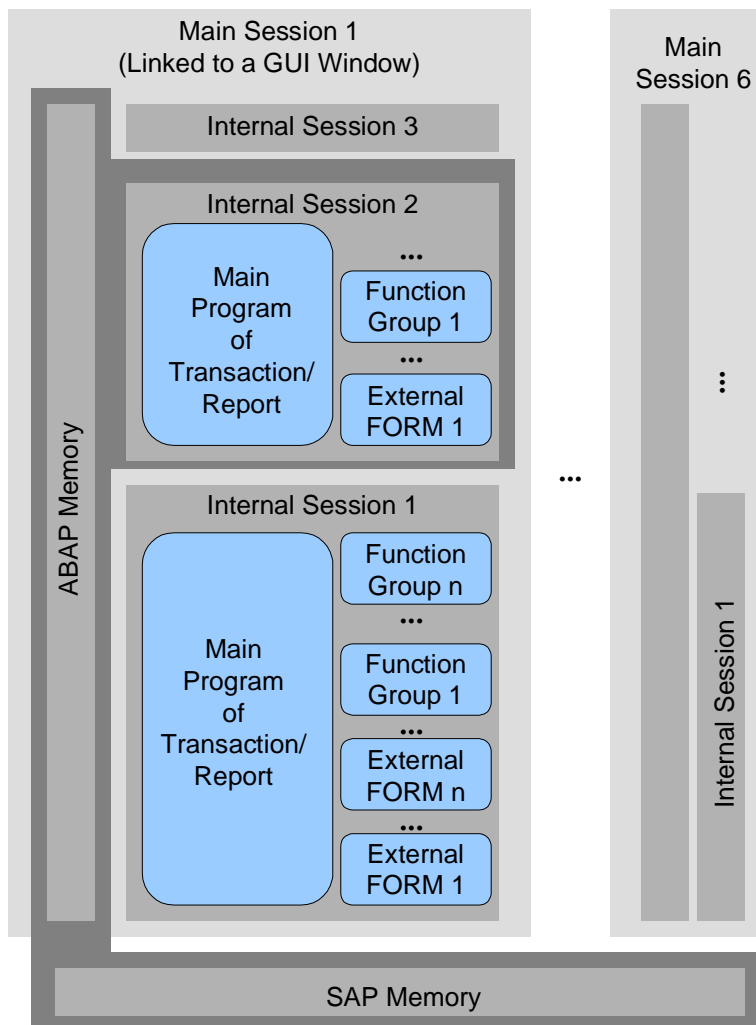
Only one internal session is ever active. If the active application program calls a further application program, the system opens another internal session. Here, there are two possible cases: If the second program does not return control to the calling program when it has finished running, the called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

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

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

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

Overview of the Components of Application Programs



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

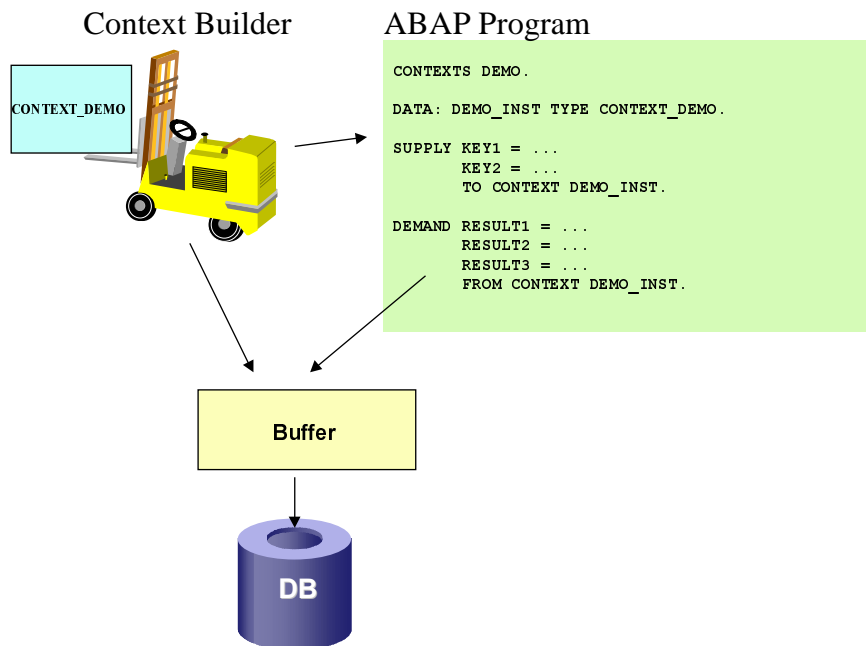
Contexts

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

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

Overview of the Components of Application Programs

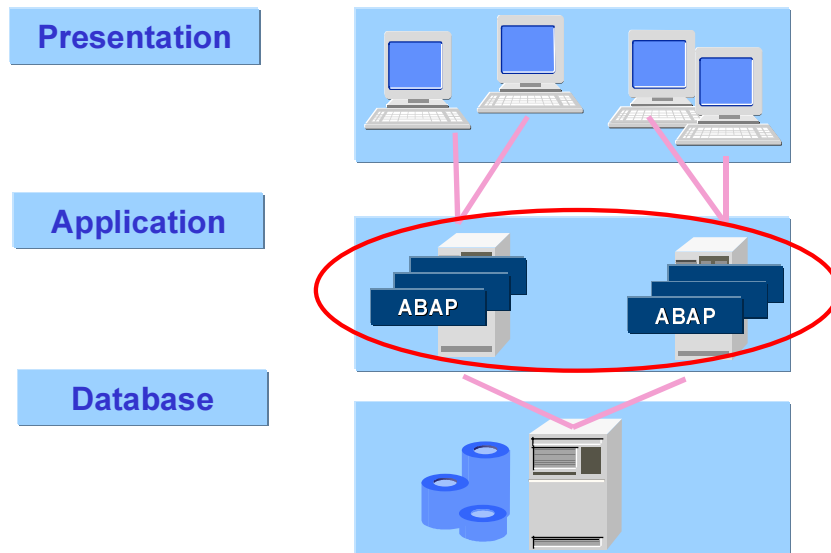
You define contexts in the Context Builder, which is part of the ABAP Workbench. They consist of key input fields, the relationships between the fields, and the other fields and values that you can derive from them. Contexts can link these derived fields by foreign key relationships between tables, by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



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

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

The ABAP Programming Language



Creating and Changing ABAP Programs

ABAP programs are objects of the R/3 Repository. Therefore, you maintain ABAP programs just like any other Repository objects (for example, ABAP Dictionary tables or user interfaces of screens) using a tool of the ABAP Development Workbench. The tool for maintaining ABAP programs is the ABAP Editor.

This section gives an overview of how to create new ABAP programs and change existing ones. It describes the possibilities the R/3 system offers to start the ABAP Editor. In the text below, 'open a program' always means 'call the ABAP Editor for a program'.

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

- [Opening programs in the Repository Browser \[Page 61\]](#)

The Object Browser of the ABAP Development Workbench (transaction SE80) offers a hierarchical overview of all R/3 Repository objects, ordered by development classes, user name of the programmers, object types, and so on. By selecting a program, the Object Browser supplies direct access to all components of a program, such as main program, subroutines, or global data. By selecting a program object in the Object Browser and calling a maintenance transaction, you directly open the appropriate tool for this object, in this case the ABAP Editor.

This procedure is suited for complex (interactive) reports or module pools for transactions, since in the Object Browser you always have an overview of all components of a program, such as user interfaces of screens or dynpros.

- [Open Programs Using the ABAP Editor \[Page 67\]](#)

To open a program object directly using the corresponding tool, on the initial screen of the ABAP Development Workbench choose *ABAP Editor* (or start transaction SE38). To change a program, you must know exactly the name of the requested program and its environment.

This procedure is suited for maintaining or creating relatively simple reports or short test programs, which have only few or even no components.

- [Open Programs via Forward Navigation \[Page 69\]](#)

Whenever you work with a tool of the ABAP Development Workbench and you position the cursor on a name of an R/3 Repository object and select the object (for example, by double-clicking the mouse), the system opens the object together with the corresponding tool. This also applies for editing ABAP programs.

This procedure is suited whenever ABAP programs are called from within objects such as screen flow logic or from within another ABAP program.

For detailed information on the Object Browser, the ABAP Editor, and the other tools of the ABAP Development Workbench, see the [ABAP Workbench Tools \[Ext.\]](#) documentation.

Opening Programs in the Repository Browser

Opening Programs in the Repository Browser

To open ABAP programs in the Repository Browser of the ABAP Development Workbench, choose *Repository Browser* on the screen *ABAP Development Workbench* or start transaction SE80. The *Repository Browser: Initial Screen* appears:

Object list

☒ Development class SAB1 Display

☐ Program SAPMTALE

☐ Function group [Dropdown Arrow]

☐ Local priv. objects KELLERH

Single object

☒ Program objects Edit

☐ Function group objects

☐ Dictionary objects

☐ Modeling objects

☐ Other objects

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

[Entering a Program Name \[Page 62\]](#)

[Displaying Programs of a Development Class \[Page 65\]](#)

Entering the Program Name Directly

Enter a program name according to the [naming conventions \[Page 83\]](#) into the object list and choose *Display*.

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

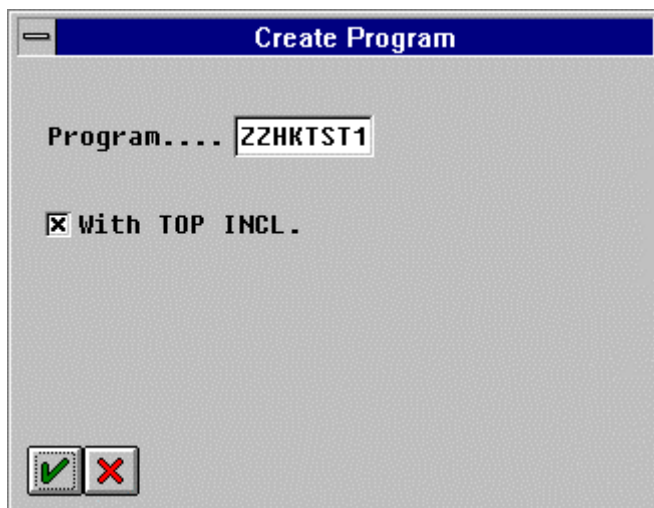
[Creating a New Program \[Page 63\]](#)

[Displaying or Changing an Existing Program \[Page 64\]](#)

Creating a New Program

Creating a New Program

1. The dialog box *Create Program* appears, which allows you to create a *TOP INCL* (top include program).



You need top include programs when programming module pools. In this case, you must adhere to the [naming conventions \[Page 83\]](#) for module pools to enable the system to create all required include programs under their correct names.

When programming stand-alone programs (reports), you need no top include program.

2. You can enter the [program attributes \[Page 74\]](#) on the *ABAP: Program Attributes* screen.

Maintaining the [program attributes \[Page 78\]](#) is an important procedure when creating a program. Fill in the input fields of the *ABAP: Program Attributes* screen carefully. The [Maintaining Program Attributes \[Page 74\]](#) section contains an overview of the input fields.

3. Save the program attributes.

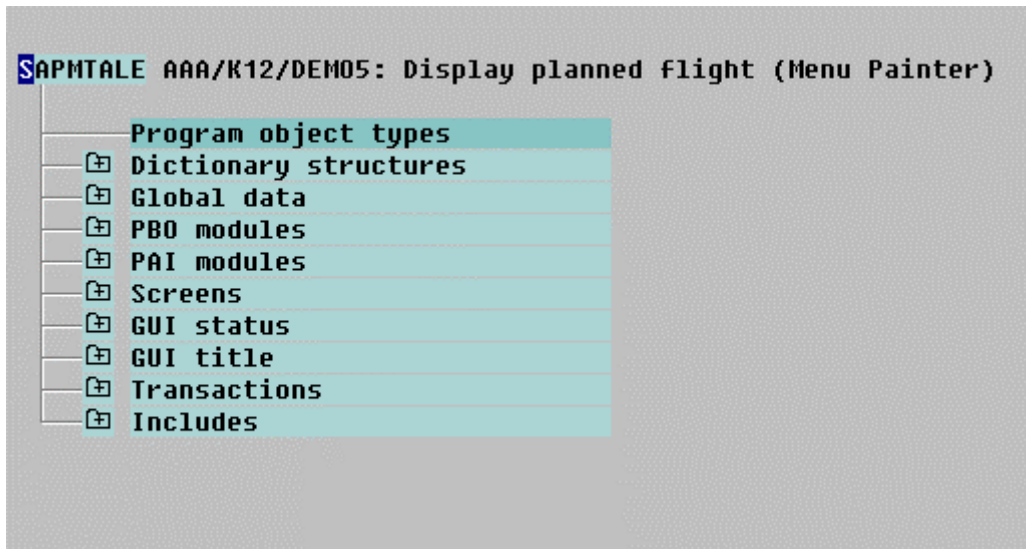
The program is now stored in the R/3 Repository and you can maintain the source code in the ABAP Editor by following, for example, the procedure explained in [Displaying or Changing an Existing Program \[Page 64\]](#), using the Repository Browser.

Or you can go to the ABAP Editor directly by choosing *Source code* on the *ABAP: Program Attributes* screen.

4. You can [edit the program \[Page 79\]](#) in the ABAP Editor.

Displaying or Changing an Existing Program

1. The Object Browser displays the overview of all components (object types) of a program.



For the above screen shot, we used the program SAPMTALE from ABAP training, which also exists in your system. This program is an example from dialog programming.

2. Position the cursor on the program name and double-click or choose *Display* or *Change*.

This opens the ABAP Editor for the program and you can start [editing the program](#) [\[Page 79\]](#).

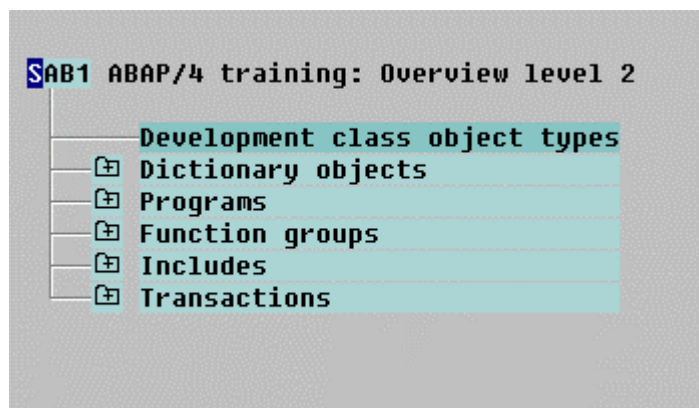
Note that selecting other components of the program does not open the ABAP Editor to display the source code of the program, but always the appropriate tool for the selected component. In special cases, such as for include programs, this can also be the ABAP Editor. However, you then edit only the component and not the source code of the entire program (see also [Opening Programs Using Forwards Navigation](#) [\[Page 69\]](#)).

Display Programs of a Development Class

Display Programs of a Development Class

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

The screen below shows the development class SAB1 from ABAP training.



You now have several possibilities:

- If on this screen the node *Programs* does not exist, you can position the cursor on *Development class object types* and choose *Create* to create this node by creating a program.

A dialog box appears (*Development Objects*). Choose the entry *Program objects*. On the subsequent dialog box (*Program Objects*) choose the entry *Program*. Enter a name according to the [naming conventions \[Page 83\]](#), and choose *Create* again.

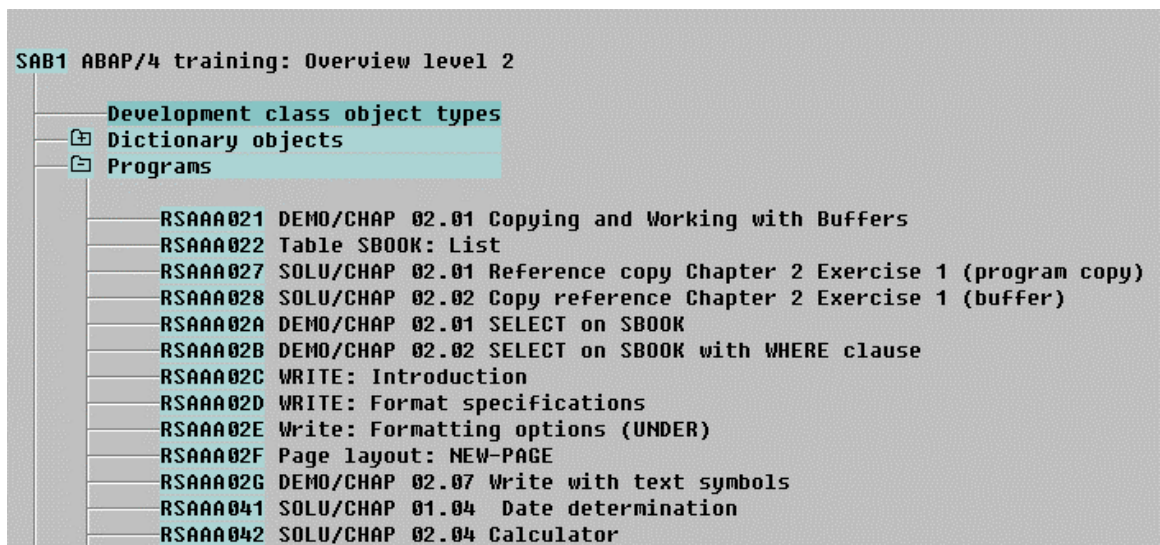
The dialog box *Create Program* appears. From here, carry on as described in [Creating a new Program \[Page 63\]](#).

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

The dialog box *Create Program* appears. Enter a name corresponding to the [naming conventions \[Page 83\]](#) and carry on as described in the section [Creating a new Program \[Page 63\]](#)

- If on this screen the node *Programs* exists, you can expand the node to display all programs of the development class.

Display Programs of a Development Class



Position the cursor on a program name and double-click or choose *Display* or *Change*.

This opens the ABAP Editor for the program and you can start [editing the program](#) [Page 79].

Open Programs Using the ABAP Editor

Open Programs Using the ABAP Editor

To open ABAP programs directly using the ABAP Editor, choose *ABAP Editor* in the *ABAP Development Workbench* screen or start transaction SE38. The *ABAP Editor: Initial Screen* appears.

In the *Program* field, enter a program name according to the [naming conventions \[Page 83\]](#).

Creating a new program

If the program does not exist, choose *Create*. The *ABAP: Program Attributes* screen appears regardless of which *object component* you selected.

Here, the system does **not** offer a top include. This procedure of creating a program is therefore suited for reports and short test programs only.

From here, carry on as described in [Creating a new Program \[Page 63\]](#).

Maintaining an Existing Program

If the program exists, you can mark one of the following *Object components* and choose *Display* or *Change* to edit the component.

- *Source Code*

This opens the ABAP Editor for the program and you can start [editing the program \[Page 79\]](#).

- *Variants*

Mark this to maintain variants. Variants allow you to define fixed values for the input fields on the selection screen of a report. For further details about variants, see [Filling Selection Screens Using Variants \[Page 864\]](#).

- *Attributes*

Open Programs Using the ABAP Editor

Here, you maintain the [program attributes \[Page 78\]](#). The [Maintaining Program Attributes \[Page 74\]](#) section contains an overview of the input fields.

- *Documentation*

Mark this to maintain a program-specific documentation for a report. The system opens the *SAPscript* Editor, where you can enter the documentation according to the predefined template. When executing the report, the user can display this documentation by choosing *System → Services → Reporting* (transaction SA38) and *Goto → Documentation*. If the report is stored as node in a reporting tree (transaction SERP), the user can choose *Goto → Display docu* there to display the documentation.

- *Text elements*

Mark this to maintain text elements. Text elements are all texts that appear on the selection screen or on the output screen of a report. For further information about text elements, see [Working with Text Elements \[Page 146\]](#).

To go to these object components from within the ABAP Editor, choose the menu entry *Goto*.

Opening Programs With Forward Navigation

Opening Programs With Forward Navigation

When you are working with the ABAP Development Workbench and you position the cursor on a program name and select the corresponding program by double-clicking, the system opens the ABAP Editor for this program and you can start [editing the program \[Page 79\]](#).

For more ways of forward navigation, see the menu bars of the ABAP Development Workbench tools, for example *Goto* or *Environment*.

The examples below show some ways of forward navigation.

Suppose, you are editing a program and find the line

SUBMIT ZZHKTST 1.

This statement calls a report from within another ABAP program. For further information, see [Calling Executable Programs \(Reports\) \[Page 1113\]](#)

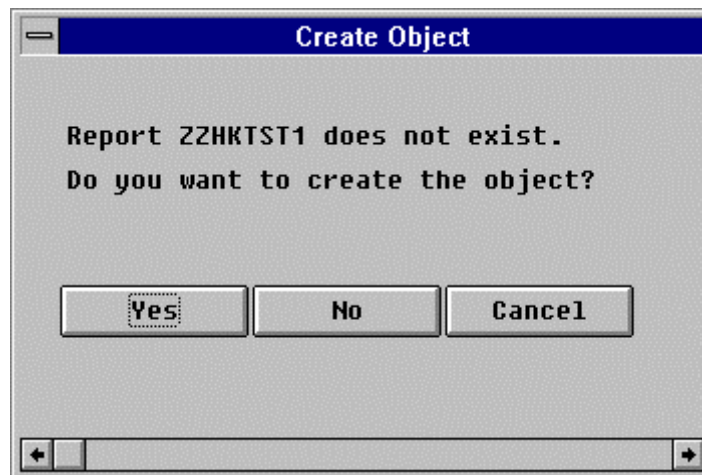
If you position the cursor on the name ZZHKTST1 and double-click, there are two possibilities:

1. The report ZZHKTST1 exists.

The system opens the Editor for ZZHKTST1 and you can start [editing the program \[Page 79\]](#).

2. The report ZZHKTST1 does not exist.

The system displays the following dialog box:

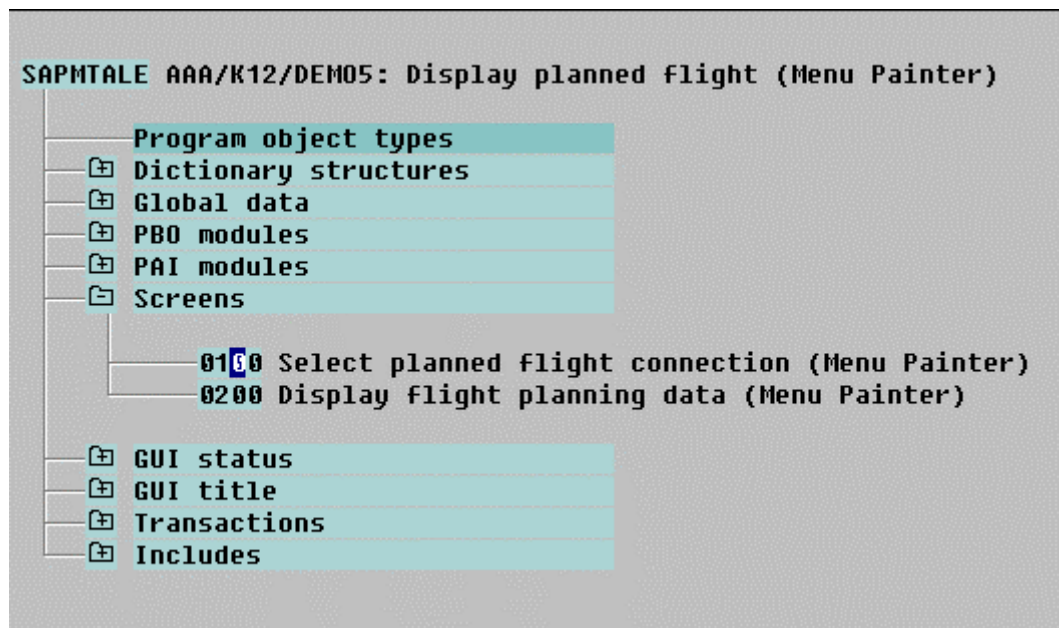


If you choose Yes, the system proceeds as described in [Creating a New Program \[Page 63\]](#).

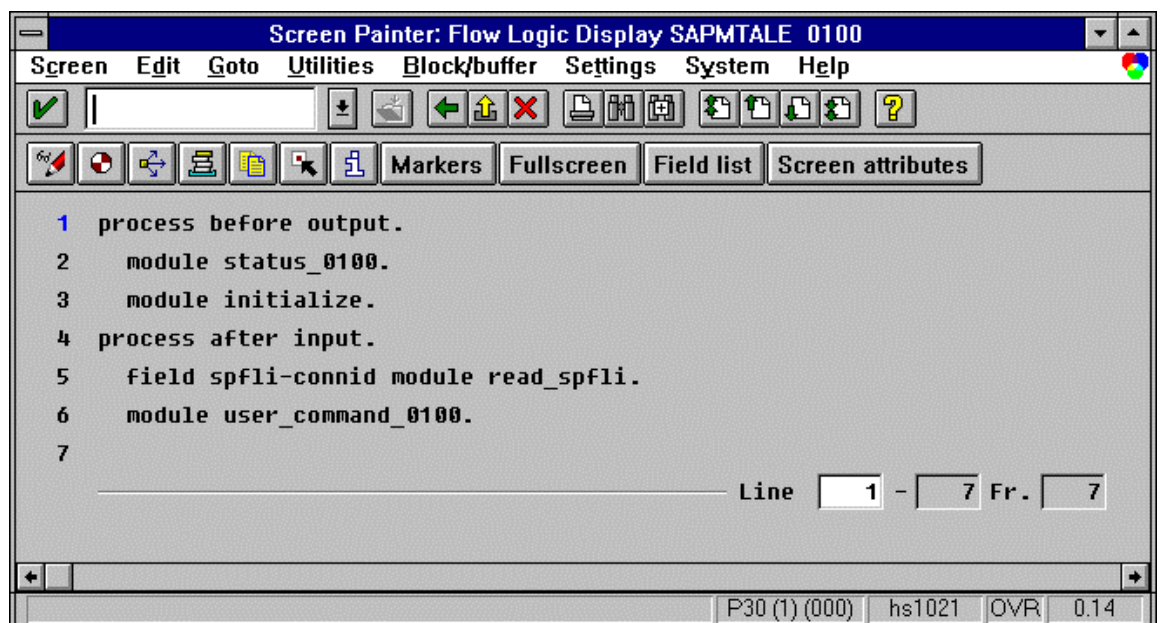
After editing ZZHKTST1, you can choose *Back* to return to the ABAP Editor session of the original program.

In the Object Browser open the dialog program SAPMTALE from the development class SAB1 (ABAP training) and position the cursor on screen 100:

Opening Programs With Forward Navigation



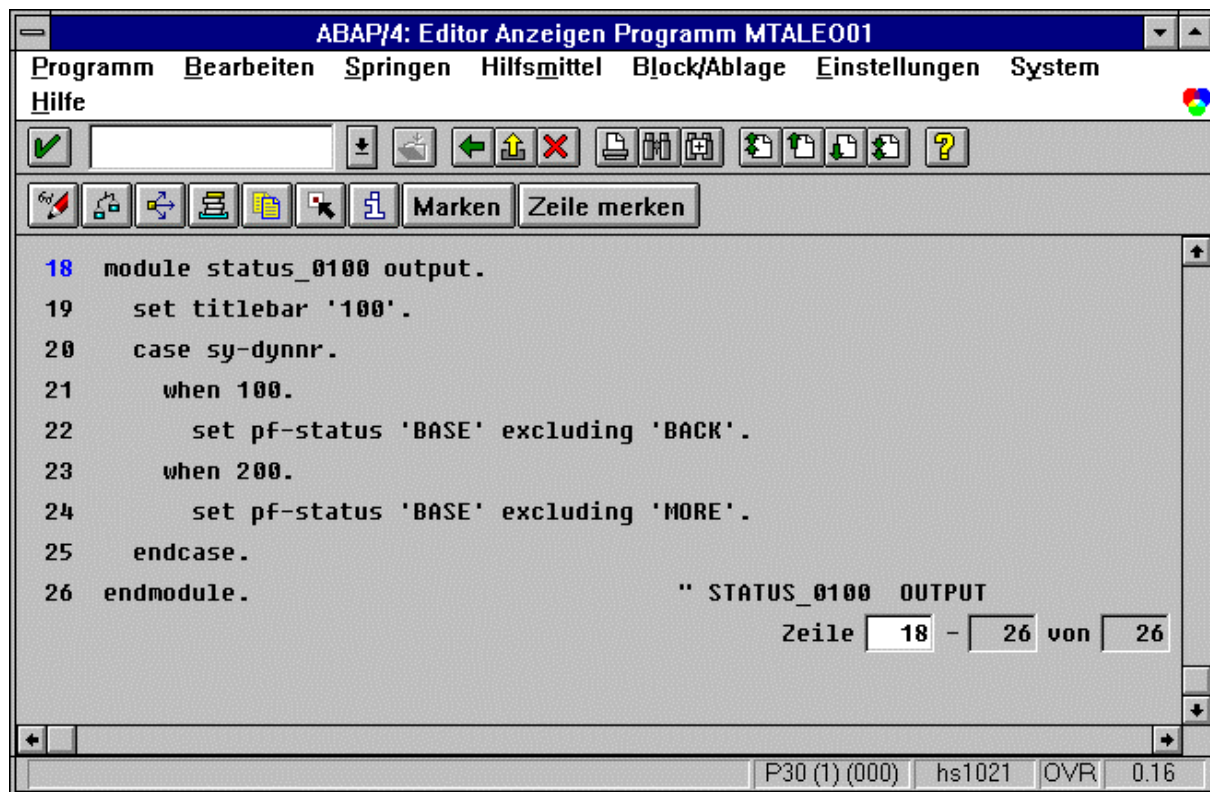
Choose *Display*, *Change*, or double-click on the screen to open the SCREEN PAINTER tool, where the system displays the screen flow logic of the selected screen:



Here, you see the screen statement MODULE, which calls an ABAP module in the ABAP module pool.

Position the cursor, for example, on STATUS_100 and double-click. The system opens the ABAP Editor for the include program MTALEO01 at the position at which the module STATUS_100 is defined.

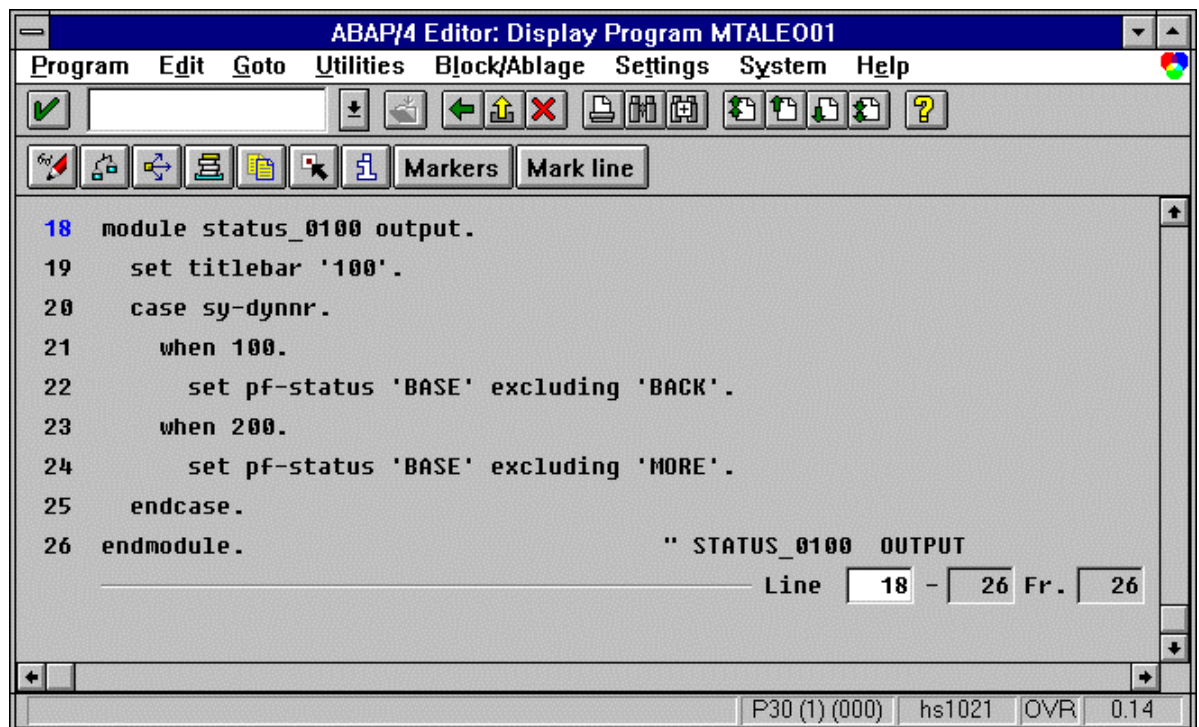
Opening Programs With Forward Navigation



The include program MTALEO01 is used to modularize the source code of the main program SAPMTALE (see [INCLUDE Programs \[Page 441\]](#)).

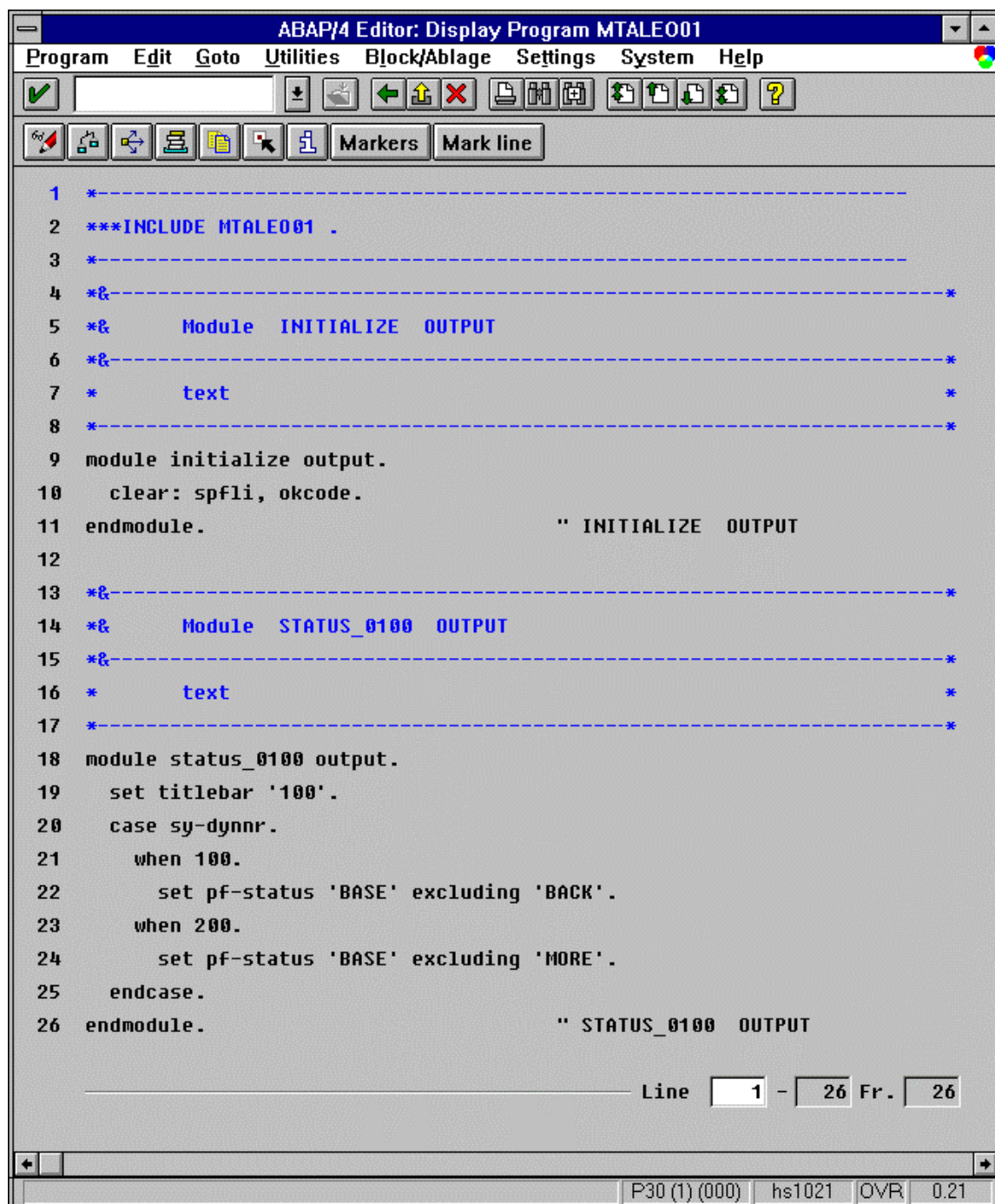
In the Object Browser open the dialog program SAPMTALE from the development class SAB1 (ABAP training) and position the cursor on the program name to open the program. The ABAP Editor displays the following source code:

Opening Programs With Forward Navigation



The INCLUDE statements include the specified include programs into the source text. Position the cursor on MTALEO01 and double-click. The system opens the include program MTALEO01.

Opening Programs With Forward Navigation



In this include program, you find all modules for the PBO event of all screens of the dialog program SAPMTALE. Compare them to the previous example. For more details on dialog programming, see part 3 of the User's Guide.

Maintaining Program Attributes

You maintain the [program attributes \[Page 78\]](#) on the *ABAP: Program Attributes* screen.

If you create a report, enter 1 in the *Type* field. If you create a module pool, enter M in the *Type* field. For a list of other possible types, use the possible entries button.

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

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

Overview of all program attributes

The following section provides information about [program attributes \[Page 78\]](#). Note that some of these attributes only apply to executable programs (reports), and not to other ABAP program types. The field help and possible values help for the fields on the *ABAP: Program Attributes* screen provide further information.

Version	<input type="text" value="0"/>	Created on/by	<input type="text" value="01/02/1997"/>	<input type="text" value="KELLERH"/>
		Last changed on/by	<input type="text"/>	<input type="text"/>
Title				
<input type="text" value="?"/>				
Maintenance language	<input type="text" value="E"/>	English		
Attributes				
Type	<input type="text" value="1"/>	Online program		
Status	<input type="text"/>			
Application	<input type="text" value="*"/>	Not application-specific		
Authorization group	<input type="text"/>			
Development class	<input type="text"/>			
Logical database	<input type="text"/>			
From application	<input type="text"/>			
Selection screen	<input type="text"/>			
<input type="checkbox"/> Upper/lower case		<input type="checkbox"/> Fixed pt.arithmetic		
<input type="checkbox"/> Editor lock		<input type="checkbox"/> Start via variant		

Version

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

Title

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

Maintaining Program Attributes

Maintenance Language

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

Type

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

Use **Type 1** to declare your program as executable.. This means that the program can run on its own, and that you can start it in the R/3 system without a transaction code. You can also run executable programs (reports) in the background. You can also [assign a transaction code to an executable program \(report\) \[Page 81\]](#)

Use **Type M** to declare your program as a module pool. This means that your program **cannot** run on its own, but serves as a frame for program modules used for dialog programming. These program modules contain the application logic of a transaction and are called by a separately programmed screen flow logic (programming screens using the Screen Painter tool). The screen flow logic itself can be called via a transaction code only. See also [Dialog Mode \[Page 1309\]](#)

Apart from type 1 (for executable programs (reports)) and type M (for module pools), you should also know **Type I** for include programs. An include program is an independent program with two main functions: On one hand, it contains program code that can be used by different programs. On the other hand, it modularizes source code, which consists of several different, logically related parts. Each of these parts is stored in a different include program. Thus, include programs enhance readability of the source code and facilitate its maintenance (for more information, see [INCLUDE Programs \[Page 441\]](#)).

Status

This entry has protocol character and describes the status of the program development, such as T for test program.

Application

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

Authorization Group

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

Development Class

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

If you are working in a team, you may have to assign your program to an existing development class, or you may be free to create a new class. All programs assigned to the development class \$TMP are private objects and cannot be transported into other systems.

You can enter the development class directly into this field. Otherwise, the system prompts for it when you save the attributes:

Create object catalog entry

Object

Attributes

Development class

Author

Repaired ☐

Original system

Originalsprache ☐

Generated object ☐

Choosing *Local object* is equivalent to entering \$TMP in the field *Development class*.

You can change the development class of a program at a later time by choosing, for example, *Program* → *Reassign* in the *ABAP: Program Attributes* screen.

Logical Database from Application

Only for Executable Programs (Reports)

These attributes determine the logical database used by the executable program (report) to read data, and the application to which it belongs. Logical databases have unique names within their application. However, systemwide, you can have more than one logical database with the same name. This is why you also need to specify the application. For information about logical databases, refer to [Attributes and Maintenance of Logical Databases \[Page 1246\]](#)

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

Selection Screen Version

Only for Executable Programs (Reports)

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

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

Maintaining Program Attributes

logical database (program DBxxxSEL, where xxx is the name of the logical database). For further information, refer to [Logical Database Selections \[Page 1252\]](#).

Upper/Lower Case

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

Editor Lock

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

Fixed Point Arithmetic

If you set this attribute, the system rounds fields with type P according to the number of decimal places, or fills them with zeros. For further information about type P fields, see [Numeric Data Types \[Page 107\]](#). The decimal sign in this case is always the period (.), regardless of the user's personal settings.

We recommend that you **always** set the fixed point arithmetic attribute.

Start Using Variant

Only for Executable Programs (Reports)

If you set this attribute, other users can only start your program using a variant. Before running the program, you must create at least one variant (for information about creating variants, see [Filling Selection Screens Using Variants \[Page 864\]](#).)

Program Attributes

In the program attributes, you determine the runtime environment of a program and thus determine how it runs. The most important program attribute in this respect is the type. This determines whether the program can be started directly, or only in conjunction with a particular screen.

You can also [assign a transaction code to an executable program \(report\) \[Page 81\]](#)

The program attributes also tell you the application to which the program belongs, and, in the case of executable programs (reports), the name of any associated logical database. Logical databases are special ABAP programs that you use to read data from a variety of tables.

Take care to enter the correct program attributes, otherwise the system will not be able to run the program properly.

Editing Your Program

Editing Your Program

To edit programs, you use the ABAP Editor tool. For more information about creating function modules, refer to the [ABAP Workbench Tools \[Ext.\]](#) documentation.

Writing programs

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

1. The first program statement

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

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

The statements REPORT or PROGRAM can have several options, such as LINE-SIZE, LINE-COUNT, or NO STANDARD PAGE HEADING. You use these options mainly in programs that which evaluate data and display the results in a list. For more information about designing lists, see [Creating Complex Lists \[Page 938\]](#) For other options, such as the definition of a message class, see the key word documentation.

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

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

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

2. Data declaration

Now declare all global data of your program. For further information, see [Declaring Data \[Page 103\]](#).

3. Processing logic

Next comes the processing logic, which consists of a number of processing blocks (see [Program Structure \[Ext.\]](#)).

4. Subroutines

The last unit of a program should be the list of all subroutines of the ABAP program. For more information about subroutines, see [Modularizing ABAP Programs \[Page 437\]](#)

Modularizing programs into source code modules does not change the above structure. If, for example, you follow the forward navigation of the ABAP Development Workbench when creating a dialog program, the system automatically creates a number of include programs, which contain the program parts described above in the correct sequence. The top include program usually

contains the statement `PROGRAM` and the global data declaration. The subsequent include programs contain the individual dialog modules, ordered by PBO and PAI. Further include programs may contain, for example, subroutines. These include programs do not influence the functionality of the program, they only serve to enhance source code readability. See also the last two examples in the section [Opening Programs by Forward Navigation \[Page 69\]](#).

Checking programs

After you finish editing or reach a temporary version, choose *Check* to check the syntax. The system checks the program coding for syntax errors and compatibility problems. If it finds an error, it displays a message that indicates the error and, if possible, offers a solution or correction. The system positions the cursor on the error in the coding.

Saving programs

Choose *Save* to save the source code.

The system stores the source code in the program library. When you execute the program for the first time **outside** the ABAP Editor or by choosing *Generate* on a screen of the ABAP Development Workbench, the system creates a runtime object for this program.

Testing programs in the ABAP Editor

You can test executable programs in the ABAP Editor. Choose *Program* → *Execute* from within the ABAP Editor. The system then creates a temporary runtime object with a name that differs from the program name.

However, the system executes the program as if started outside the ABAP Editor. If, for example, you created a report, you first see the selection screen for entering selection criteria and then the output list.

If you created an ABAP module pool, you cannot test the program in the ABAP Editor. You must create a transaction code and a screen flow logic before you can execute the program (for more information, see [Dialog Mode \[Page 1309\]](#)).

For test purposes, you can even execute a stand-alone program without first saving it. The ABAP Editor stores a temporary version of the source code including all changes. However, after finishing the test, you must return to the editor and save the program to ensure that all changes are stored.

Testing a program often involves a runtime analysis, which shows you the amount of time your program consumes in the client/server environment of the R/3 system and what this time is used for. For a runtime analysis, choose *System* → *Utilities* → *Runtime* → *analysis*. For more information about creating function modules, refer to the [ABAP Workbench Tools \[Ext.\]](#) documentation.

For information on how to measure the runtime of program segments using ABAP statements, see [Checking the Runtime of Program Segments \[Page 500\]](#).

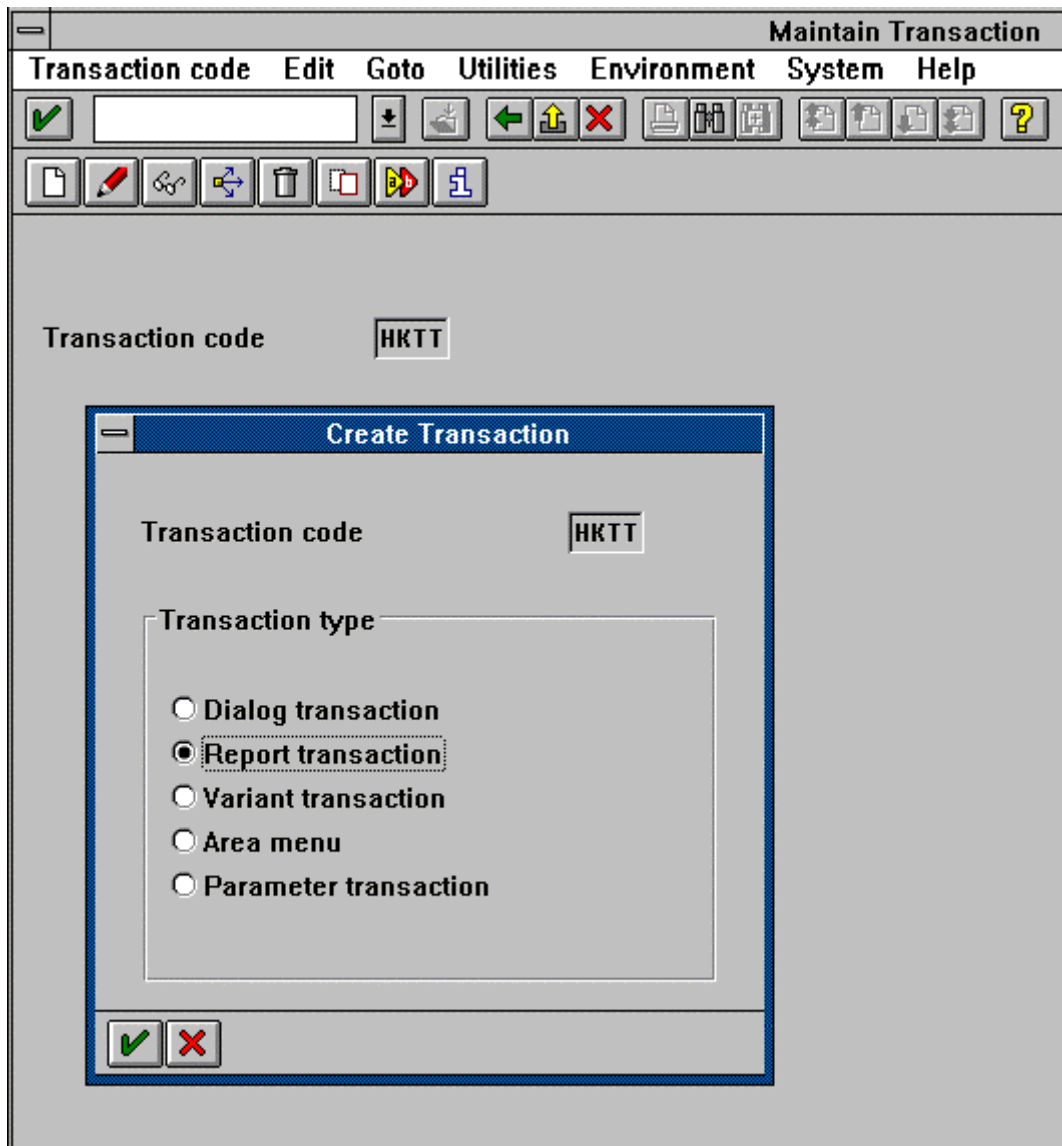
Assigning Transaction Codes to Executable Programs (Reports)

Assigning Transaction Codes to Executable Programs (Reports)

You can assign a transaction code to any executable program (report) with type Users can then start the program as though it were a transaction.

Procedure:

1. Create an executable program (report) with type 1, or choose an existing one.
2. From the ABAP Workbench, choose *Development* → *More tools* → *Transactions*.
3. On the *Maintain Transaction Codes* screen, enter a name for your transaction and choose *Create*.
4. The system displays a dialog box. Choose *Report transaction*.



Assigning Transaction Codes to Executable Programs (Reports)

It is important that you choose this transaction type, to ensure that the executable program is run by the same invisible system program when it is called as a transaction as when it is called directly (see [Starting Programs Directly \[Ext.\]](#)). If you choose Dialog transaction from the above screen, the program will be run from screen flow logic as though it were a dialog transaction (see [Starting a Program Using a Transaction Code \[Ext.\]](#)).

5. On the *Create Report Transaction* screen, enter the *Transaction text*, *Program* and *Selection screen*. These are required fields.
6. Save the transaction code in the required development class.

Naming Conventions

Executable Programs (Reports)

Customer programs should follow the naming convention **Yaxxxxxx** or **Zaxxxxxx**. Replace **a** with the identification code of the appropriate application. Replace **x** with any valid character. SAP programs follow a similar convention, namely **Raxxxxxx**.

Module Pool Programs (for Transactions)

Customer dialog programs should follow the naming convention **SAPMYxxx** or **SAPMZxxx**. Replace **x** with any valid character. SAP standard programs follow a similar convention, namely **SAPMaxxx**, where **a** stands for an application.

Valid Characters

- The program name must be at least one character long, and may be up to eight characters long.
- Do not use the following characters:
 - Period (.)
 - Comma (,)
 - Space ()
 - Parentheses ' (' ') '
 - Inverted comma (')
 - Quotation mark (")
 - Equals sign (=)
 - Asterisk (*)
 - German umlauts (Ä, ä, Ö, ö, Ü, ü)
 - Percentage sign (%) or underscore (_)

Basic Statements

ABAP Program Syntax and Layout

This section describes the ABAP syntax and provides recommendations on how to program in ABAP. It also explains how you can improve the clarity of your program and use ready-made modules of program code to make programming easier. The topics in this section cover:

[Syntax Elements \[Page 86\]](#)

[Syntax Structure \[Page 91\]](#)

[ABAP Program Layout \[Page 95\]](#)

[Inserting Predefined Structures \[Page 100\]](#)

This section is only an overview. For more detailed information about the individual ABAP components, see the corresponding topics in this guide.

Syntax Elements

The ABAP programming language consists of the following element types:

[Statements \[Page 87\]](#)

[Keywords \[Page 88\]](#)

[Comments \[Page 90\]](#)

Statements

Statements

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

```
PROGRAM SAPMTEST.
```

```
WRITE 'First Program'.
```

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

Keywords

A keyword is the first word of a statement. It determines the meaning of the entire statement. There are four different types of keywords:

- **Declarative keywords**

These keywords define data types or declare the data objects which the program can access. Examples of declarative keywords are:

TYPES, DATA, TABLES

The system processes declarative keywords during the generation of a program, not at runtime. They are processed independently of their position in the program code. For the sake of clarity, you should specify all declarative keywords together in a “declaration section” at the beginning of the program.

For more information about declarative keywords, see [Declaring Data \[Page 103\]](#)

- **Modularization keywords**

These keywords define processing blocks in an ABAP program. Processing blocks are groups of statements which are processed during the execution of an ABAP program as soon as they are called from another point.

Modularization keywords comprise:

- **Event keywords**

The respective processing blocks are processed as soon as a particular event occurs. Examples of event keywords are:

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

For more information about event keywords, see [Controlling the Flow of ABAP Programs Using Events \[Page 1208\]](#)

- **Defining keywords**

These keywords define processing blocks that are processed as soon as they are called by an explicit statement in an ABAP program or in a screen flow logic. Examples of defining keywords are:

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

For more information about defining keywords, see [Modularizing ABAP Programs \[Page 437\]](#).

- **Control keywords**

These keywords control the flow of an ABAP program according to certain conditions. Examples of control keywords are:

IF, WHILE, CASE

For more information about control keywords, see [Controlling the Flow of an ABAP Program \[Page 234\]](#).

- **Calling keywords**

Keywords

These keywords call processing blocks (defined by modularization keywords) in the same or other ABAP programs or branch completely to other ABAP programs.

Examples of calling keywords are:

PERFORM, CALL, SUBMIT, LEAVE TO

For more information about calling keywords, see the respective sections of this guide.

- **Operational keywords**

These keywords process the data (defined by declarative keywords) when certain processing blocks (triggered by events or called by calling keywords) are processed and certain conditions (defined by control keywords) occur. Examples of operational keywords are:

WRITE, MOVE, ADD

For more information about operational keywords, see [Processing Data \[Page 171\]](#)

Comments

Comments are text elements which you can write between the statements of your ABAP program to explain its purpose to a reader. Comments are flagged by special characters which cause the system to ignore them. You should use comments to document your program internally. Comments help other users to understand or change the program.

```

*****
* PROGRAM SAPMTZST *
* CREATED BY CARL BYTE, 06/27/1995 *
* LAST CHANGE BY RITA DIGIT, 10/01/1995 *
* PURPOSE: DEMONSTRATION *
*****

PROGRAM SAPMTEST.
*****
* DECLARATION PART *
*****

DATA.....

.....
*****
* OPERATION PART *
*****

.....

```

All lines beginning with an asterisk (*) are comments and are ignored by the system (for further information about comments, see [Structure of Comments \[Page 93\]](#)).

Syntax Structure

An ABAP program is a sequence of different statements which have a particular structure.

[Statement Structure \[Page 92\]](#)

Comments can be inserted between the statements.

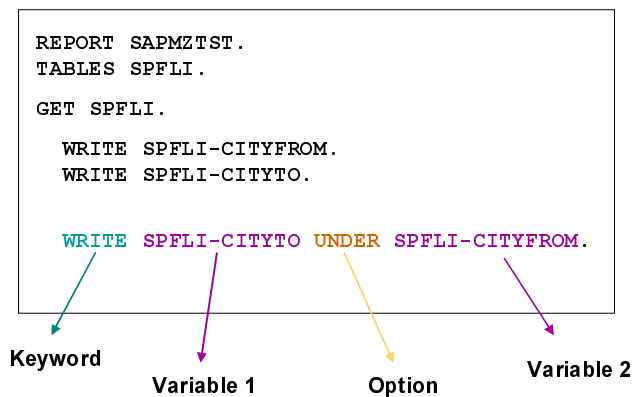
[Comment Structure \[Page 93\]](#)

Sequences of separate, but similar, statements can be combined into chain statements.

[Concatenating Similar Statements \[Page 94\]](#)

Structure of Statements

The following diagram shows the structure of ABAP statements.



ABAP has no format restrictions. You can enter statements in free format. This means that you can indent statements, write several statements on one line, or extend one statement over several lines.

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

The program fragment

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

could also be written as follows:

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

or as follows:

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

You should use free formatting to improve the readability of your program, but you should avoid using complicated layouts.

Structure of Comments

You can insert comment lines anywhere in a program. There are two ways to indicate comments in a program:

- If you want the entire line to be a comment, enter an asterisk (*) at the beginning of the line.
- If you want part of a line to be a comment, enter a double quotation mark (") before the comment. The system interprets comments indicated by double quotation marks as spaces.

```
PROGRAM SAPMTEST.
```

```
* The following line contains a WRITE statement
```

```
WRITE 'First Program'. " Output on List
```

The second line of this program is a comment which is not executed. The comment is indicated by an asterisk (*) at the beginning of the line.

On the third line, everything after the double quotation mark (") is a comment and is not executed.

The rest of the program consists of executable statements with the keywords PROGRAM and WRITE.

Concatenating Similar Statements

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

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

Statement sequence:

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

Chain statement:

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

In the chain, a colon separates the beginning of the statement from the variable parts. You can insert as many spaces as you like before and after the colon (or comma).

For example, you could also write the same statement as follows:

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

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

Statement sequence:

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

Chain statement:

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

ABAP Program Layout

To write a high-quality program, you should not only follow the appropriate [naming conventions \[Page 83\]](#) , but also keep to certain layout standards for ABAP programs.

You should observe these standards as soon as you start defining your data. Note the proposals in the following topics when structuring your program flow, and use as many informative comments as possible. If you follow these suggestions, your programs will be

- more readable
- easier to test and change
- more reliable

To improve the quality of your programs, use the techniques described in the following:

[Indenting Statement Blocks \[Page 96\]](#)

[Using Modularization Tools \[Page 97\]](#)

[Inserting Program Comments Correctly \[Page 98\]](#)

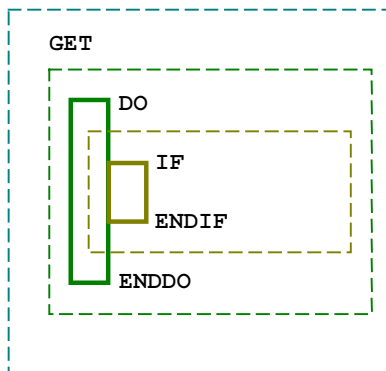
The ABAP Editor includes a tool which helps you to design the layout of your program more easily. This tool is known as the Pretty Printer.

[Pretty Printer \[Page 99\]](#)

Indenting Blocks of Statements

You should combine statements that belong together into a single block. For more information about control keywords, see [Controlling the Flow of an ABAP Program \[Page 234\]](#).

Indent each block by at least two columns, as shown below.



Using Modularization Tools

You can modularize your programs with the help of the ABAP Workbench (see [Modularizing ABAP Programs \[Page 437\]](#)).

If you write larger processing blocks as subroutines, the logical structure of your program becomes easier to identify. It also allows you to sort the subroutines according to the tasks they perform.

Subroutines may increase the overall length of programs, but you will soon find that this method greatly increases clarity, especially in the case of complex programs. If you list the subroutines in the order in which the system executes them, your code will be easy to read.

Inserting Program Comments Correctly

You should avoid placing comments on statement lines. Placing them on separate comment lines improves the readability of the program.

To insert subroutine headings and comments in your program, use the ready-made structures available in the ABAP Editor. In the subroutine heading, explain the purpose of the called program and provide adequate information and references. For further information about predefined structures, see [Inserting Predefined Keyword Structures \[Page 101\]](#) and the ABAP Editor section of the [ABAP Workbench Tools \[Ext.\]](#) documentation.

Pretty Printer

Pretty Printer

Using the Pretty Printer enables you to follow the ABAP layout guidelines more easily.

The Pretty Printer is an optional function of the ABAP Editor (for more information, see the documentation on the ABAP Editor in the [ABAP Workbench Tools \[Ext.\]](#) documentation).

To call the Pretty Printer from the ABAP Editor, choose *Program* → *Pretty Printer*.

Here is an example of how the Pretty Printer works.

The following shows the layout of a program before using the Pretty Printer:

```
PROGRAM SAPMTEST.

      DATA: SUM1 TYPE I, SUM2 TYPE I, SUM3 TYPE I.
      IF SUM1 = SUM2.
        WRITE 'Case 1'.
      ELSEIF SUM1 = SUM3. WRITE 'Case 2'.
      ENDIF.
```

The following is the same program after using the Pretty Printer:

```
PROGRAM SAPMTEST.

DATA: SUM1 TYPE I, SUM2 TYPE I, SUM3 TYPE I.
IF SUM1 = SUM2.
  WRITE 'Case 1'.
ELSEIF SUM1 = SUM3.
  WRITE 'Case 2'.
ENDIF.
```

Inserting Predefined Structures

Predefined structures simplify the coding of ABAP programs. They provide the exact syntax and follow the ABAP layout guidelines.

You can insert two kinds of predefined structures into your program code when using the ABAP Editor:

[Inserting Predefined Structures \[Page 101\]](#)

[Inserting Predefined Comment Lines \[Page 102\]](#)

For detailed information on ready-made structures see the documentation on the ABAP Editor in the [ABAP Workbench Tools \[Ext.\]](#) documentation.

Inserting Predefined Keyword Structures

Inserting Predefined Keyword Structures

1. Place the cursor on the line where you want to insert the structure.
2. Choose *Edit* → *Insert statement* or select *Pattern*.
3. The following dialog box appears: In the dialog box which appears, choose a statement with a radio button or enter it in the *Other pattern* field:

To display a list of all predefined keyword structures, place the cursor in the *Other pattern*. field and click the possible entries button to the right of the input field. All structures with an asterisk (*) as a first character are ready-made comment lines (for more information about ready-made comment lines, see [Inserting Predefined Comment Lines \[Page 102\]](#)).

If you enter CASE in the *Other instruct.* field of the dialog box, the system inserts the following lines into your program:

```
CASE f.
  WHEN w1.
    .....
  WHEN w2.
    .....
  WHEN OTHERS.
    .....
ENDCASE.
```

Inserting Predefined Comment Lines

1. Carry out steps 1 and 2 as described in [Inserting Predefined Keyword Structures \[Page 101\]](#).
2. Select a structure with an asterisk (*) as a first character from the list of possible entries in the *Other instruct.* field in the dialog box.
3. The system inserts comment lines into your program.

If you entered ****3** in the *Other instruct.* field in the dialog box, the system inserts the following lines into your program:

```
*****
*                                     *
*                                     *
*                                     *
*****
```

Declaring Data

Declaring Data

This section describes how to declare data in ABAP programs. To be able to do this, you should be familiar with the ABAP type concept which defines the relationship between data types and data objects.

The ABAP type concept is introduced in the

[Introduction to Data Types and Objects \[Page 104\]](#)

The following topics describe

[Data Types \[Page 105\]](#)

[Data Objects \[Page 112\]](#)

[Creating Data Objects and Data Types \[Page 118\]](#)

[Summarizing Examples \[Page 132\]](#)

[Type Groups \[Page 137\]](#)

[Determining the Attributes of Data Objects \[Page 139\]](#)

Throughout this guide, the term 'field' is used for data objects.

In general, a field can be

- any data object (see [Data Objects \[Page 112\]](#))
- a field symbol or a formal parameter (see [Working with Field Symbols \[Page 336\]](#) and [Passing Data Using Parameters \[Page 454\]](#))

It will always be clear exactly what 'field' means from the context.

Introduction to Data Types and Objects

Data types and data objects are the essential components of the ABAP type concept. Both can be declared and maintained by the user. This contrasts with other programming languages like C, FORTRAN, or PASCAL where the user can declare data objects but is restricted to the predefined data types. In ABAP, you can process data types independently of data objects, using a syntax which is parallel to normal data declaration. It is also possible to store data types centrally outside a specific program.

The following list is an overview of the main features of data types and objects:

- Data types (elementary or structured)
 - Data types are pure descriptions.
 - They do not occupy memory.
 - They describe the technical attributes of data objects.
 - The data type is an attribute of a data object and is closely connected with it.
- Data objects (literals, variables, constants, and so on):
 - Data objects are the physical units a program uses at runtime.
 - Each data object has a specific data type assigned to it.
 - Each data object occupies some space in memory.
 - ABAP processes a data object according to its data type.

In your programs, you must declare all the data objects you want to work with. In the declaration, you must assign certain attributes to a data object. The most important of these attributes is the data type. In ABAP, you can either use predefined data types similar to other program languages, or user-defined data types.

User-defined data types are a powerful programming tool, allowing you a high degree of flexibility. They range from elementary types (such as character strings of a given length) to very complex structures (like nested tables).

Assigning user-defined data types to data objects allow you to work with precisely the data objects you require. User-defined data types can be used in the same way as predefined data types. You can declare them locally within a program or store them globally in the ABAP Dictionary.

There are three hierarchical levels of data types and objects:

- program-independent data, defined in the ABAP Dictionary
- internal data used globally in one program
- data used locally in a procedure (subroutine, function module)

You will learn more about these levels in the corresponding sections of this guide.

Data Types

Data Types

Data types in ABAP are classified by structure and definition. Data types are either:

- elementary (non structured) or structured
- predefined or user-defined

Resulting from this, there are four classes of data types that are listed in the following table. The names of the predefined data types are fixed. The names of the user-defined data types are defined by the programmer in the program.

Data types in ABAP

	Predefined	User-defined
Elementary	C, D, F, I, N, P, T, X: ABAP contains eight predefined elementary data types. Elementary Data Types - Predefined [Page 106]	User-defined elementary data types are based on the predefined elementary data types. Elementary Data Types - User-defined [Page 108]
Structured	TABLE: This predefined structured data type is used only for the typing of formal parameters and field symbols.	Field strings and internal tables: These structured data types can be used for data objects and are user-defined. Structured Data Types [Page 109]

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

[Compatibility of Data Types \[Page 111\]](#).

Elementary Data Types - Predefined

The following table summarizes the elementary data types which are predefined in ABAP (the valid size is given in bytes):

ABAP data Types

Data type	Initial size	Valid size	Initial value	Meaning
C	1	1 - 65535	SPACE	Text, character (alphanumeric characters)
D	8	8	'00000000'	Date (format: YYYYMMDD)
F	8	8	0	Floating point number
I	4	4	0	Integer (whole number)
N	1	1 - 65535	'00...0'	Numeric text
P	8	1 - 16	0	Packed number
T	6	6	'000000'	Time (format: HHMMSS)
X	1	1 - 65535	X'00'	hexadecimal

The data types D, F, I, and T are predefined in all respects, but the data types C, N, P, and X can have additional specifications. For example, you can define the size in the program.

For more information about the **numeric data types** I, F, and P, see [Numeric Data Types \[Page 107\]](#).

Numeric Data Types

Numeric Data Types

ABAP supports three numeric data types. These are:

- integers (whole numbers) of type I
- packed numbers of type P
- floating point numbers of type F

Type I Data

The value range of type I numbers is -2^{31} to $2^{31}-1$ and includes only **whole** numbers.

Non-integer results of arithmetic operations (e.g. fractions) are rounded, not truncated.

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

Type P Data

Type P data allows digits after the decimal point.

The value range of type P data depends on its size and the number of digits after the decimal point. The valid size can be any value from 1 to 16 bytes. Two decimal digits are **packed** into one byte, while the last byte contains one digit and the sign. Up to 14 digits are allowed after the decimal point. For further information about defining decimal places, see the section [Basic Form of the DATA Statement \[Page 120\]](#).

When working with type P data, it is a good idea to set the program attribute *Fixed point arithmetic* (see [Maintaining Program Attributes \[Page 74\]](#)). Otherwise, type P numbers are treated as integers.

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

Type F Data

The value range of type F numbers is 1×10^{-307} to 1×10^{308} for positive and negative numbers, including 0 (zero).

The accuracy range is approximately 15 decimals, depending on the **floating point** arithmetic of the hardware platform.

Since type F data is internally converted to a binary system, rounding errors can occur. Although the ABAP processor tries to minimize these effects, you should not use type F data if high accuracy is required. Instead, use type P data.

If you need large value ranges, and rounding errors are not important, you can use type F data.

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

Type N data does **not** contain numbers. Although it consists of digits, these digits are not used for calculations. Typical type N fields are account numbers and zip codes.

Elementary Data Types - User-Defined

User-defined elementary data types are based entirely on predefined elementary data types. You define your own data types using the [TYPES \[Page 130\]](#) statement.

User-defined elementary data types make your programs easier to read and maintain:

- If, for example, you often need to use a certain set of data types in your programs, you can ensure that you are always working with the same data types by creating an include program for the type definition (see [INCLUDE programs \[Page 441\]](#)) or by defining the data types in a type group in the ABAP Dictionary (see [Type Groups \[Page 137\]](#)).
- To make the data types easier to understand and more recognizable, you can give them descriptive names.
- After assigning a user-defined data type to several data objects, you change the data type of all these data objects in a single operation by changing the definition of the data type in the TYPES statement.

```
TYPES: NUMBER TYPE I,  
      LENGTH TYPE P DECIMALS 2,  
      CODE(3) TYPE C.  
  
.....  
  
DATA: NO_FLIGHTS TYPE NUMBER,  
      NO_PASSENGERS TYPE NUMBER,  
      DISTANCE TYPE LENGTH,  
      HEIGHT TYPE LENGTH,  
      ....  
      CITY_CODE TYPE CODE,  
      COUNTRY_CODE TYPE CODE,  
      .....
```

In this example, a data type called NUMBER is defined. It is the same as the predefined data type I, except it has a different name to make the program easier to read.

Also defined in this example is a data type LENGTH which is based on the predefined elementary data type P. LENGTH is defined with a given number of decimals. If it becomes necessary to change the accuracy of length specifications, for example, you only have to change the TYPES statement in the program.

A third data type, CODE, is also defined. CODE is based on the predefined type C, with a given length of 3.

Structured Data Types

Structured Data Types

In general, structured data types in ABAP programs are user-defined.

There are no predefined structured data types you can use in ABAP with the exception of the following:

You can use the predefined generic table type `TABLE` to pass an internal table with a generic line structure to a subroutine (see [Typing Formal Parameters \[Page 461\]](#)) or to type a field symbol (see [Typing Field Symbols \[Page 341\]](#)).

There are two kinds of structured data types:

- Structures

A structure is a collection of other data types. The components of structures can themselves be structures or internal tables.

You define structures using the `TYPES` or `DATA` statement (see [The DATA Statement for Structures \[Page 126\]](#)).

With the `DATA` statement, you do not define a standalone data type, but a data object with a structured type.

- internal tables

An internal table consists of several lines of the same type. Unlike structures, which extend only 'horizontally', internal tables also extend 'vertically'.

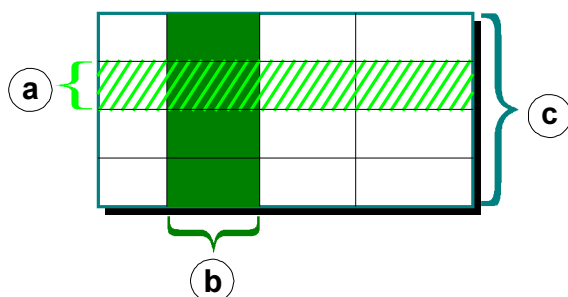
You define internal tables with the `OCCURS` parameter of the `TYPES` or `DATA` statements (see [Creating Internal Tables \[Page 270\]](#)).

Since structures can contain components of any type and internal tables can be defined for any type, user-defined data structures can become very complex.

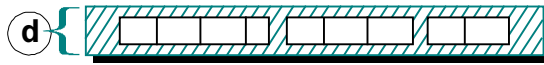
Examples of Structured Data Types

The following are examples of structured data types in ABAP

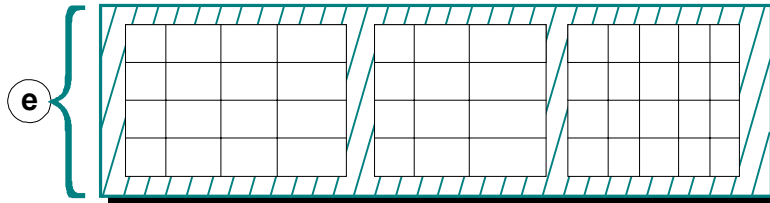
- structures, which consist of a series of elementary data types (flat structures, only 'horizontal')
- internal tables defined for elementary fields (dynamic arrays of elementary data types, only 'vertical')
- internal tables defined for simple structures (tables with lines and columns)



- structures containing sub-structures (nested structures)



e) structures containing internal tables as components (deep structures)



f) internal tables defined for structures which contain structured components (tables of deep line type)

g) internal tables defined for structures which contain internal tables as components

Compatibility of Data Types

Compatibility of Data Types

When working with data, you occasionally convert data from one type to another. To do this, the data types must be convertible (see [Type Conversions \[Page 218\]](#)). An important prerequisite for convertibility is compatibility.

If two data types have the same technical properties, they are compatible.

The consequences for the different ABAP data types are as follows:

Elementary Data Types

Elementary data types are compatible with other elementary data types if they are identical in type, size, and (for type P) the number of digits after the decimal point.

Elementary data types and structured data types are not compatible with each other.

Structured Data Types

With structured data types, you must distinguish between field strings and internal tables.

Structures

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

Structures are not compatible with elementary data types and internal tables.

Internal Tables

Internal tables are compatible with other internal tables if their line types are compatible. The compatibility of internal tables does not depend on the number of lines.

Internal tables are not compatible with structures and elementary data types.

In this context, compatibility between data types always refers to fully specified data types, since every data type which occurs at runtime of an ABAP program is fully specified (e.g. the length of a data type C is always defined and known). This compatibility is symmetrical.

In connection with the typing of formal parameters and field symbols a somewhat different, unsymmetrical compatibility is used (see [Typing Formal Parameters \[Page 461\]](#) and [Typing Field Symbols \[Page 341\]](#)).

Data Objects

In ABAP, you can work with several kinds of data objects, such as:

- Internal data objects

Internal data objects are created for use in one particular program. They have no validity outside that program. Internal data objects include:

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

[Variables \[Page 115\]](#)

[Constants \[Page 116\]](#)

- External data objects

External data objects exist independently of programs. You cannot work with them directly, but you can copy them to internal data objects and write them back when you have finished. External data objects can be used globally throughout the system environment.

ABAP stores external data objects in tables defined in the ABAP Dictionary. To access this data from within a program, you declare the tables in the program with the [TABLES statement \[Page 129\]](#).

- System-Defined Data Objects

Besides user-defined data objects, some data objects are defined automatically by the system. (See

[System-defined Data Objects \[Page 117\].\)](#)

- Special data objects

ABAP also includes some data objects with special features, namely:

- Parameters

Parameters are variables which are linked to a selection screen. They can accept values after a program is started.

- Selection criteria

Selection criteria are special internal tables used to specify value ranges. They are also linked to a selection screen.

For more information about these data objects and their declarations, see [Working with Selection Screens \[Page 795\]](#).

Literals

Literals

A literal is a fixed value. ABAP distinguishes between text literals and number literals.

Text literals

Text literals are sequences of alphanumeric characters enclosed in single quotation marks.

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

Text literals can be up to 254 characters long. If a text literal contains a quotation mark, you must repeat it to enable the system to recognize the contents as a text literal and not as the end of the literal.

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

This statement generates the following output:

This is John's bicycle

To ensure that programs remain easy to maintain and language-independent, do not use text literals explicitly in the final version of a program. Instead, define them externally as text symbols. Text symbols are part of the text element concept in ABAP. For further information on this and an explanation of how to create and use text symbols, see [Working with Text Elements \[Page 146\]](#).

Number literals

Number literals are sequences of digits which may contain a leading sign. They can contain up to 15 digits.

```
123  
-93  
+456
```

If you need a non-integer value or a longer number, you must use a text literal which is converted to the correct type automatically (for more information about this, see [Type Conversions \[Page 218\]](#)).

```
'12345678901234567890'  
'+0.58498'  
'-8473.67'
```

Likewise, use a text literal to represent a floating point value. This must have the following format:

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

```
'-12.34567'  
'-765E-04'
```

'1234E5'

'+12E+23'

'+12.3E-4'

'1E160'

Variables

Variables

A variable is used to store and reference data under a certain name and in a certain format. Variables can differ in

- name
- type
- length
- structure

You declare constants in your program with the CONSTANTS statement (see [The CONSTANTS Statement \[Page 119\]](#)).

```
DATA: S1 TYPE I,  
      S2 TYPE I,  
      SUM TYPE I.
```

```
....
```

```
SUM = S1 + S2.
```

```
....
```

Constants

A constant is a data object containing a value which you fix at initialization. Constants cannot be changed during the execution of the program.

You declare constants in your program with the `CONSTANTS` statement (see [The `CONSTANTS` Statement \[Page 127\]](#)). If you try to change the constant elsewhere in the program, either the syntax check or the runtime system outputs an error message.

You can also declare constants in [type groups \[Page 137\]](#) in the ABAP Dictionary.

Use constants if you need a specific value frequently in your program. In this case, do not use literals. If it becomes necessary to modify this value, you only have to change the declaration.

```
CONSTANTS PI TYPE P DECIMALS 10 VALUE '3.1415926536'.
```

```
....
```

System-Defined Data Objects

System-Defined Data Objects

When you start an ABAP program, some data objects are available automatically and do not need to be declared. System-defined data objects include:

- SPACE

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

- System fields

All system fields are named according to the following convention: SY-<name>, where <name> is the name of the individual field. To display a list of available system fields in the ABAP Editor, type SHOW SY in the command line (choose *Edit* → *More functions* → *Command input*). The list also includes the data types of the system fields.

The following are examples of system fields:

- SY-SUBRC: return code value (zero, if an operation was successful)
- SY-UNAME: logon name of the user
- SY-DATUM: current date
- SY-UZEIT: current time
- SY-TCODE: current transaction

System fields are variables and can be changed by the program. However, SAP does not recommend that you do so, because you will lose the information stored in these fields when the execution of your program continues.

Creating Data Objects and Data Types

This section describes how to create data objects and data types in your program. Apart from literals, you must declare each data object with a declarative statement.

In declarative statements, you must specify the data types of all data objects. To do this, you can use any data type described under [Data Types \[Page 105\]](#).

You define the data type of an object in the declarative statement, either

- directly, using <declaration>...TYPE <datatype>....
- indirectly, using <declaration> LIKE <dataobject>....

TYPE and LIKE are optional additions to most of the data declaration statements listed below.

With the TYPE option, you assign the data type <datatype> directly to the declared data object.

With the LIKE option, you assign the data type of another data object <dataobject> to the declared data object. This means that you reference the data type indirectly.

There are separate name spaces for data objects and data types. This means that a name can at the same time be the name of a data object as well as the name of a data type.

In your program, you can either define data objects statically using data declaration statements or create them dynamically with an operational statement.

ABAP includes the following keywords for creating data objects and data types statically:

The DATA Statement [Page 119]	for creating variables
The CONSTANTS Statement [Page 127]	for creating constants
The STATICS Statement [Page 128]	for creating variables which exist as long as the program runs, but are only visible in a procedure
The TABLES Statement [Page 129]	for creating table work areas
The TYPES Statement [Page 130]	for creating user-defined data types

In the context of internal tables, you use the operational statements APPEND, COLLECT, and INSERT to create lines of an internal tables dynamically (see [Filling Internal Tables \[Page 278\]](#)).

In the context of selection screens, you use the additional statements PARAMETERS and SELECT-OPTIONS to create data objects with a special function (see [Working with Selection Screens \[Page 795\]](#)).

The DATA Statement

The DATA Statement

You use the DATA statement to define local and global variables. For further information about the difference between local and global variables, see [Passing Data Between Calling Programs and Subroutines \[Page 451\]](#).

The DATA statement has one basic form and three main variants:

- DATA statement for structures
- DATA statement for internal tables
- DATA statement for common memory blocks

The basic form of the DATA statement is described in

[Basic Form of the DATA Statement \[Page 120\]](#)

The DATA statement for structures is described in

[DATA Statement for Structures \[Page 126\]](#)

Due to the complex nature of internal tables, a separate section is dedicated to this topic. For further information about internal tables, refer to [Creating and Processing Internal Tables \[Page 260\]](#).

You declare common memory blocks for several programs in connection with external subroutines. Therefore, this topic is covered in the section [Declaring Data as a Common Area \[Page 452\]](#).

Basic Form of the DATA Statement

You use the basic form of the DATA statement to define internal variables in your program. The syntax is as follows:

Syntax

DATA <f>[(<length>)] <type> [<value>] [<decimals>].

In its basic form, the keyword DATA has the following parameters:

Parameters	Purpose
<f>	Declaring Variables [Page 121]
<length> <type>	Specifying the Data Type and the Length of the Variable [Page 122]
<value>	Specifying the Initial Value [Page 124]
<decimals>	Specifying the Decimal Places [Page 125]

Naming a Variable

Naming a Variable

The variable name <f> may be up to 30 characters long. You can use any alphanumeric characters except those listed below.

- Do not use the following characters:
 - plus sign +
 - period.
 - comma,
 - colon :
 - parentheses ()
- Do not create a name consisting entirely of numeric characters.

ABAP contains predefined data objects with reserved names which you are not allowed to use (see [System-defined Data Objects \[Page 117\]](#)). In addition, do not use variable names in a statement if they could be confused with the parameters of the keyword introducing the statement.

Follow these guidelines when writing DATA statements:

- Use meaningful field names which do not require additional comments.
- Do not use hyphens because they indicate structures (see [The DATA Statement for Structures \[Page 126\]](#)).
- Use underscores to split up long names.
- Avoid all other special characters.
- Always use a letter as the first character of a field name.

Examples of self-explanatory field names:

```
GROUP_TOTAL  
ACCOUNT_NO  
FOREIGN_CURRENCY  
BANK_CODE
```

Specifying the Data Type and the Length of the Variable

To specify the data type, use the <type> parameter. You can create the <type> parameter by using either of the following:

- TYPE <t>
- LIKE <g>

The TYPE parameter

With the TYPE parameter you can specify either a predefined data type or a user-defined data type (see [Data Types \[Page 105\]](#)). The syntax is as follows:

Syntax

DATA <f>[(<length>)] TYPE <t>.

You can define the length of some of the elementary data types. To do this, enter the required value in place of <length>. This applies to field types C, P, N, or X. For a list of initial values and valid values, see the table in [Predefined Elementary Data Types \[Page 106\]](#).

Examples of specifying data types with the TYPE parameter:

```
DATA: NUMBER      TYPE P,  
      DATE         TYPE D,  
      HEXADECFIELD TYPE X,  
      COUNT        TYPE I,  
      LINE(72)     TYPE C.
```

The LIKE parameter

With the LIKE parameter, you can assign the data type of already defined data objects to the variable. To do so, you use:

Syntax

DATA <f> LIKE <g>.

When you use the LIKE parameter, the field <f> is created with exactly the same type and structure as the data object <g>.

You can use any data object for <g>. With the LIKE parameter, you can reference the data type of a data object declared in the ABAP Dictionary (for example a table, a structure, a view, or their individual fields).

```
DATA NUMBER_1 TYPE P.  
DATA NUMBER_2 LIKE NUMBER_1.  
DATA MYNAME   LIKE SY-UNAME.
```

In this example, the data object NUMBER_2 is declared with the same data type as the data object NUMBER_1. The data object MYNAME has the same data type as the system-defined data object SY-UNAME.

The LIKE parameter is often used for auxiliary fields to store the contents of database fields temporarily. This facility enables you to avoid unintentional differences resulting from typing errors

Specifying the Data Type and the Length of the Variable

or changes in the definition of the database field. If, for example, you change the attributes of a database field, the system adapts the attributes of the auxiliary field automatically.

```
DATA PLANE LIKE SFLIGHT-PLANETYPE.
```

This statement creates a data object called PLANE with the same attributes as the ABAP Dictionary field SFLIGHT-PLANETYPE. PLANETYPE is a column of the database table SFLIGHT.

To create the data object <f> with the same type as a line of an existing internal table, use the LIKE parameter as follows:

Syntax

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

In this case, <itab> must be a data object created as an internal table (see [Creating Internal Table Data Objects \[Page 273\]](#)).

Defaults for Type and Length

If the parameters <length> and <type> are not specified in the DATA statement, a character field (type C) of length 1 is created. If a length is specified but no type, a character field of the given length is created.

```
DATA TEXTFIELD.
```

This example creates a character field TEXTFIELD which has length 1.

Specifying a Start Value

When you declare an internal variable with the DATA statement, you implicitly assign the data-type specific initial value as a start value to the field. For a list of the initial values of the various data types, see the table in [Predefined Elementary Data Types \[Page 106\]](#).

To change the start value of a field in the program, use the <value> parameter of the DATA statement. The syntax is as follows:

Syntax

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

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

- a [literal \[Page 113\]](#)
- a [constant \[Page 116\]](#)
- the explicit addition IS INITIAL

Examples of start value specifications:

```
DATA: COUNTER TYPE P VALUE 1,  
      DATE    TYPE D VALUE '19920601',  
      FLAG    TYPE C VALUE IS INITIAL.
```

After this data declaration, the character string FLAG contains its type specific initial value SPACE.

Specifying the Number of Digits after the Decimal Point

Specifying the Number of Digits after the Decimal Point

To define the number of digits after the decimal point for type P fields, you use the <decimals> parameter. The syntax is as follows:

Syntax

DATA <f> TYPE P DECIMALS <d>.....

The maximum number of digits <d> after the decimal point is 14 (see [Numeric Data Types \[Page 107\]](#)).

You can assign data objects to a packed number type variable, that have more decimal digits than the variable. If you have specified the program attribute *Fixed point arithmetic* (see [Maintaining Program Attributes \[Page 74\]](#)), the superfluous digits are rounded.

```
DATA WEIGHT TYPE P DECIMALS 2 VALUE '1.225'.
```

If the attribute *Fixed point arithmetic* is set, the value of WEIGHT is 1.23.

DATA Statement for Structures

A structure is a group of internal fields in a program. To declare a structure, you use the DATA statement and mark the beginning and the end of the group of fields with BEGIN OF and END OF. The syntax is as follows:

Syntax

```
DATA: BEGIN OF <fstring>,  
      <component declaration>,  
      .....  
      END OF <fstring>.
```

These statements define a structure <fstring>.

In <component declaration>, you declare the component fields by specifying the length, type, and, if necessary, the start value or number of decimal digits, as explained in [The Basic Form of the DATA Statement \[Page 120\]](#).

You address the individual components in structures by using the name of the structure as a prefix and joining it to the component with a hyphen: <fstring>-<component>.

The components of a structure can have different data types. Since fields of type I or F are aligned (see [Aligning Data Objects \[Page 233\]](#)), the system inserts empty filler fields between the components, if necessary. Structures are sometimes also called records.

```
DATA: BEGIN OF ADDRESS,  
      NAME(20)  TYPE C,  
      STREET(20) TYPE C,  
      NUMBER    TYPE P,  
      POSTCODE(5) TYPE N,  
      CITY(20)  TYPE C,  
      END OF ADDRESS.
```

This example defines a field string ADDRESS of length 73. The components can be addressed by ADDRESS-NAME, ADDRESS-STREET, and so on.

You can group the declaration of long structures in include programs (see [INCLUDE Programs \[Page 441\]](#)). However, if you use the data structures frequently, it is preferable to store them in the ABAP Dictionary. For further information about search helps, see the online documentation for the [ABAP Dictionary \[Ext.\]](#).

The CONSTANTS Statement

The CONSTANTS Statement

If you use a constant frequently in a program, you can declare it as a fixed value variable with the CONSTANTS statement as follows:

Syntax

```
CONSTANTS <c>[<length>] <type> <value> [<decimals>].
```

To define structures as constants, write:

```
CONSTANTS: BEGIN OF <fstring>,  
            <component declaration>,  
            .....  
            END OF <fstring>.
```

The parameters of these statements are identical to those for the DATA statement, described in [The Basic Form of the DATA Statement \[Page 120\]](#) and [The DATA Statement for Structures \[Page 126\]](#).

The use of the <value> parameter is obligatory for the CONSTANTS statement and not optional as in the DATA statement. The start value specified with the <value> parameter cannot be changed during the execution of the program.

```
CONSTANTS: MYNAME(10)      VALUE 'Fred',  
            BIRTHDAY  TYPE D VALUE '19600110',  
            ZERO      TYPE I VALUE IS INITIAL.
```

The last line of this example demonstrates the purpose of the argument IS INITIAL. Since the parameter VALUE is mandatory for the CONSTANTS statement, you can use IS INITIAL to assign the type specific initial value to a constant.

```
CONSTANTS: BEGIN OF MYADDRESS,  
            NAME(20)  TYPE C VALUE 'Fred Flintstone',  
            STREET(20) TYPE C VALUE 'Cave Avenue',  
            NUMBER    TYPE P VALUE 11,  
            POSTCODE(5) TYPE N VALUE 98765,  
            CITY(20)  TYPE C VALUE 'Bedrock',  
            END OF MYADDRESS.
```

This example defines a constant field string MYADDRESS. The components can be addressed by MYADDRESS-NAME, MYADDRESS-STREET, and so on, but they cannot be changed.

The STATICS Statement

If you want to retain the value of a variable beyond the runtime of a procedure (subroutine or function module - for more information, see [Modularizing ABAP Programs \[Page 437\]](#)), you define the variable with the STATICS statement in that procedure. The STATICS statement is a variation of the DATA statement. The syntax is as follows:

Syntax

STATICS <s>[<length>] <type> [<value>] [<decimals>].

To define field strings in a procedure as statically valid, you write:

```
STATICS: BEGIN OF <fstring>,  
          <component declaration>,  
          .....  
        END OF <fstring>.
```

The parameters of these statements are identical to those for the DATA statement, described in [The Basic Form of the DATA Statement \[Page 120\]](#) and [The DATA Statement for Structures \[Page 126\]](#).

If you call a procedure from your program several times, the procedure always uses the last value of a variable defined with STATICS in the procedure. A static variable cannot be addressed from outside of the procedure.

The TABLES Statement

The TABLES Statement

With the TABLES statement, you can create a data object called a table work area. A table work area is a field string which refers to ABAP Dictionary objects. The syntax is as follows:

Syntax:

TABLES <dbtab>.

<dbtab> is the name of the ABAP Dictionary object and also the name of the table work area created. The sequence and the names of the components of the table work area are the same as for the object declared in the ABAP Dictionary.

Valid ABAP Dictionary objects you can reference are

- database tables
- structures
- views

To create objects in the ABAP Dictionary, you can, for example, choose *Tools → ABAP Workbench → Development → ABAP Dictionary* on the R/3 initial screen or double-click on the object's name in the TABLES statement. For further information, see the online documentation for the [ABAP Dictionary \[Ext.\]](#).

The table work area provides an interface for loading data from database tables into your program or for modifying the contents of database tables with Open SQL statements (see [Reading and Processing Database Tables \[Page 538\]](#)).

To display a list of a table's components and their data types, you type SHOW <dbtab> in the command line of the ABAP Editor (choose *Edit → More functions → Command input*).

The data types of ABAP Dictionary objects are not the same as the data types of the ABAP type concept. Refer to the keyword documentation of the TABLES statement for a list of ABAP Dictionary data types and how they correspond to the data types in ABAP programs.

To address the components of a table work area, use the table name as prefix and append the component with a hyphen: <dbtab>-<component> (see [The DATA Statement for Structures \[Page 126\]](#)).

```
TABLES: SPFLI.
```

```
SELECT * FROM SPFLI.
```

```
  WRITE: SPFLI-MANDT, SPFLI-CARRID, SPFLI-CONNID,.....  
ENDSELECT.
```

In this example, the TABLES statement creates a table work area SPFLI. SPFLI has the same structure as the database table SPFLI declared in the ABAP Dictionary. In the SELECT loop, the table work area SPFLI is filled with rows from the database table SPFLI (see [Reading and Processing Database Tables \[Page 538\]](#)).

The TYPES Statement

You use the TYPES statement to create user-defined elementary data types and structured data types.

You can use data types defined by the TYPES statement in the same way you use predefined data types for declaring data objects. The syntax is as follows:

Syntax

```
TYPES <t>[<length>] <type> [<decimals>].
```

This statement defines a data type <t>.

To define a structured data type, you write:

```
TYPES: BEGIN OF <fstring>,
        <component declaration>,
        .....
      END OF <fstring>.
```

The parameters of these statements are identical to those for the DATA statement, described in [The Basic Form of the DATA Statement \[Page 120\]](#) and [The DATA Statement for Structures \[Page 126\]](#).

Note that you cannot use the <value> parameter in the TYPES statement because there is no memory associated with a data type. Therefore, you cannot assign a value to a data type.

```
TYPES: SURNAME(20) TYPE C,
        BEGIN OF ADDRESS,
          NAME TYPE SURNAME,
          ....
        END OF ADDRESS.

DATA: ADDRESS_1 TYPE ADDRESS,
      ADDRESS_2 TYPE ADDRESS,
      .....
```

This example shows how you can use the parameter TYPE to refer directly to a user-defined data type. Three main things happen here. First, a user-defined data type, SURNAME, of type C and length 20 is created. Second, a structured data type, ADDRESS, is defined. A component of ADDRESS, ADDRESS-NAME, is given the data type SURNAME. Thirdly, the data objects ADDRESS_1, ADDRESS_2... are created as structures. The type is specified as ADDRESS.

```
DATA COUNTS TYPE I.

TYPES: COMPANY LIKE SPFLI-CARRID,
      NO_FLIGHTS LIKE COUNTS.
```

This example shows the use of the LIKE parameter in a TYPES statement. It shows how you can create data types by referring to the types of already existing data objects. Most importantly, you can create types that refer to ABAP Dictionary objects (tables, structures, views, and their individual fields), as shown for the

The TYPES Statement

user-defined data type COMPANY. COMPANY refers to the ABAP Dictionary object CARRID, which is a column of the database table SPFLI.

* User-defined types referring to predefined types:

```
TYPES: SURNAME(20) TYPE C,  
       STREET(30)  TYPE C,  
       ZIP_CODE(10) TYPE N,  
       CITY(30)   TYPE C,  
       PHONE(20)  TYPE N,  
       DATE       LIKE SY-DATUM.
```

* User-defined structured type referring to above types:

```
TYPES: BEGIN OF ADDRESS,  
       NAME TYPE SURNAME,  
       CODE TYPE ZIP_CODE,  
       TOWN TYPE CITY,  
       STR TYPE STREET,  
       END OF ADDRESS.
```

* User-defined nested structure type referring to above types:

```
TYPES: BEGIN OF PHONE_LIST,  
       ADR TYPE ADDRESS,  
       TEL TYPE PHONE,  
       END OF PHONE_LIST.
```

```
DATA PL TYPE PHONE-LIST.
```

```
.....
```

```
WRITE PL-ADR-NAME.
```

This example shows how to create complex data types from simple type definitions. After a set of simple data types are created with ABAP predefined types, a structured type ADDRESS is defined using the data types defined earlier. Finally, a nested structure type, PHONE_LIST, is created, whose first component has the structured type ADDRESS. The last lines of the example show how a data object, PL, is created with type PHONE_LIST and how its components can be addressed.

For further information about how to include internal tables in data type structures, refer to [Creating and Processing Internal Tables \[Page 260\]](#).

Summarizing Examples

To summarize the information in the preceding topics, the following topics contain four examples which cover all the important features of declaring data types and data objects in ABAP:

[Example Using Predefined Elementary Data Types and Objects \[Page 133\]](#)

[Example using User-defined Elementary Data Types and Objects \[Page 134\]](#)

[Example Using Structures \[Page 135\]](#)

[Example using Internal Tables \[Page 136\]](#)

Example of Predefined Elementary Data Types and Objects

Example of Predefined Elementary Data Types and Objects

The following program is an example of how to declare data objects with predefined elementary data types.

```
PROGRAM SAPMZTST.
```

```
DATA TEXT1(14) TYPE C.
```

```
DATA TEXT2 LIKE TEXT1.
```

```
DATA NUMBER TYPE I.
```

```
TEXT1 = 'The number'.
```

```
NUMBER = 100.
```

```
TEXT2 = 'is an integer.'.
```

```
WRITE: TEXT1, NUMBER, TEXT2.
```

This program produces the following output on the screen:

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

In this example, the data objects TEXT1, TEXT2, and NUMBER are declared with the DATA statement. The data type of each is specified with the TYPE parameter or the LIKE parameter of the DATA statement. The data types used here (C, I) are predefined in the system. Then, values are assigned to the data objects and the contents of the data objects are displayed on the screen.

Example of User-Defined Elementary Data Types and Objects

The following program is an example of how to declare data objects with user-defined elementary data types.

```
PROGRAM SAPMZTST.  
TYPES MYTEXT(10) TYPE C.  
TYPES MYAMOUNT TYPE P DECIMALS 2.  
DATA TEXT TYPE MYTEXT.  
DATA AMOUNT TYPE MYAMOUNT.  
TEXT = '4 / 3 = '.  
AMOUNT = 4 / 3.  
WRITE: TEXT, AMOUNT.
```

This program produces the following output on the screen:

```
4 / 3 =      1.33
```

In this example, user-defined data types MYTEXT and MYAMOUNT are defined with the TYPES statement with reference to elementary data types that are predefined in the system. Then, the data objects TEXT and AMOUNT are declared with the DATA statement. Their data types are determined to be MYTEXT and MYAMOUNT. Values are assigned to the data objects and the contents of the data objects are displayed on the screen.

Example of Structures

The following program is an example of how to declare structured data objects as a structure.

```
PROGRAM SAPMZTST.

TYPES: BEGIN OF NAME,
        TITLE(5)    TYPE C,
        FIRST_NAME(10) TYPE C,
        LAST_NAME(10) TYPE C,
      END OF NAME.

TYPES: BEGIN OF MYLIST,
        CLIENT      TYPE NAME,
        NUMBER      TYPE I,
      END OF MYLIST.

DATA LIST TYPE MYLIST.

LIST-CLIENT-TITLE = 'Lord'.
LIST-CLIENT-FIRST_NAME = 'Howard'.
LIST-CLIENT-LAST_NAME = 'Mac Duff'.
LIST-NUMBER = 1.

WRITE LIST-CLIENT-TITLE.
WRITE LIST-CLIENT-FIRST_NAME.
WRITE LIST-CLIENT-LAST_NAME.
WRITE / 'Number'.
WRITE LIST-NUMBER.
```

This program produces the following output on the screen:

```
Lord Howard    Mac Duff
Number        1
```

In this example, the structured data types NAME and MYLIST are defined with the TYPES statement. The structure NAME contains three components: TITLE, FIRST_NAME and LAST_NAME, with the predefined elementary data type C. The structure MYLIST contains the two components CLIENT and NUMBER. CLIENT itself is already structured because it is given data type NAME. NUMBER is elementary, and has the predefined data type I. A structured data object is declared using data type MYLIST. Values are assigned to the components and their contents are then displayed on the screen.

Example of Internal Tables

The following program is an example of how to declare structured data objects as internal tables.

```
PROGRAM SAPMZTST.

TYPES: BEGIN OF MYSTRING,
        NUMBER TYPE I,
        NAME(10) TYPE C,
      END OF MYSTRING.

TYPES MYTAB TYPE MYSTRING OCCURS 5.

DATA STRING TYPE MYSTRING.
DATA ITAB TYPE MYTAB.

STRING-NUMBER = 1. STRING-NAME = 'John'.
APPEND STRING TO ITAB.
STRING-NUMBER = 2. STRING-NAME = 'Paul'.
APPEND STRING TO ITAB.
STRING-NUMBER = 3. STRING-NAME = 'Ringo'.
APPEND STRING TO ITAB.
STRING-NUMBER = 4. STRING-NAME = 'George'.
APPEND STRING TO ITAB.

LOOP AT ITAB INTO STRING.
  WRITE: / STRING-NUMBER,STRING-NAME.
ENDLOOP.
```

This program produces the following output on the screen:

```
1 John
2 Paul
3 Ringo
4 George
```

In this example, first a data type MYSTRING is defined as a structure. Then, based on the structure MYSTRING, a data type MYTAB is defined as an internal table with the OCCURS parameter of the TYPES statement. The data objects STRING and ITAB are declared with data types MYSTRING and MYTAB. The fields of the internal table ITAB are then filled line by line. By using the structure STRING, the contents of ITAB are then displayed on the screen. For further information about internal tables, refer to [Creating and Processing Internal Tables \[Page 260\]](#).

Type Groups

Type Groups

You use type groups to store user-defined data types or constants in the ABAP Dictionary for cross-program use.

In your ABAP program, you declare type groups with the TYPE-POOLS statement as follows:

Syntax

TYPE-POOLS <name>.

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

You maintain type groups either by choosing *Tools → ABAP Workbench → Development → ABAP Dictionary* or directly from your ABAP program.

In the first case, you specify a one- to five-character name for the type group in the field *Object name* on the *ABAP Dictionary: Initial Screen* and then choose *Type groups* followed by *Display*, *Change*, or *Create*:



In the second case, you double-click on the name <name> of the type group after the TYPE-POOLS statement in your ABAP program. If no type group of this name exists, you can create it. Otherwise, the system displays the definition of the existing type group and you can change it.

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

Syntax

TYPE-POOL <name>.

After this come the definitions of the data types and constants using the [TYPES \[Page 130\]](#) and [CONSTANTS \[Page 127\]](#) statements. All the names of these data types and constants must begin with the name of the type group and an underscore: <name>_.

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

```
TYPE-POOL HKTST.
```

```
TYPES: BEGIN OF HKTST_TYP1,
        COL1(10),
        COL2 TYPE I,
      END OF HKTST_TYP1.
```

```
TYPES HKTST_TYP2 TYPE P DECIMALS 2.
```

```
CONSTANTS HKTST_ELEVEN TYPE I VALUE 11.
```

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

Any ABAP program can use this definition with the TYPE-POOLS statement, as shown in the following program:

```
PROGRAM SAPMZTST.
TYPE-POOLS HKTST.
DATA: DAT1 TYPE HKTST_TYP1,
      DAT2 TYPE HKTST_TYP2 VALUE '1.23'.
WRITE: DAT2, / HKTST_ELEVEN.
```

The output is:

```
      1,23
      11
```

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

Determining the Attributes of Data Objects

Determining the Attributes of Data Objects

If you want to find out the data type of a data object or have to work with its attributes at runtime of your program, use the DESCRIBE statement. The syntax is as follows:

Syntax

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

The attributes of the data object <f> specified by the parameters of the statement are written to the variables following the parameters.

The DESCRIBE FIELDS statement has the following parameters:

Parameters	Purpose
LENGTH	Determining the Field Length [Page 140]
TYPE	Determining the Data Type [Page 141]
OUTPUT-LENGTH	Determining the Output Length [Page 143]
DECIMALS	Determining the Number of Decimal Places [Page 144]
EDIT MASK	Determining the Conversion Routine [Page 145]

Determining the Field Length

To determine the length of a data object, you use the LENGTH parameter with the DESCRIBE FIELD statement as follows:

Syntax

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

The system reads the length of the field <f> and writes the value to the field <l>.

```
DATA: TEXT(8), LEN TYPE I.  
DESCRIBE FIELD TEXT LENGTH LEN.
```

In this example, the field LEN contains the value 8.

Determining the Data Type

Determining the Data Type

To determine the data type of a field, you use the TYPE parameter with the DESCRIBE FIELD statement as follows:

Syntax

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

The system reads the data type of the field <f> and writes the value into the field <t>.

Besides returning the predefined data types C, D, F, I, N, P, T, and X (see the table in [Predefined Elementary Data Types \[Page 106\]](#)), this statement also returns

- s for two-byte integers with leading sign
- b for one-byte integers without leading sign
- h for internal tables
- C for structures
- C for structures containing at least one nested structure as a component.

While types s and b can come from references to ABAP Dictionary objects (see [The TABLES Statement \[Page 129\]](#)), the latter three types stem from user-defined types.

With the option COMPONENTS <n>, the statement returns

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

and writes the number of direct components of the structure to <n>.

```
TABLES SPFLI.
DATA: NUMTEXT(8) TYPE N, TYP.
DESCRIBE FIELD NUMTEXT TYPE TYP.
WRITE TYP.
```

```
DESCRIBE FIELD NUMTEXT TYPE TYP.
WRITE TYP.
```

This example produces the following output:

```
N T
```

In this example, the field TYP contains first the value 'N' and then the value 'T'.

```
TYPES: SURNAME(20) TYPE C,
       STREET(30) TYPE C,
       ZIP_CODE(10) TYPE N,
       CITY(30) TYPE C,
       PHONE(20) TYPE N,
       DATE LIKE SY-DATUM.
```

```
TYPES: BEGIN OF ADDRESS,
      NAME TYPE SURNAME,
      CODE TYPE ZIP_CODE,
      TOWN TYPE CITY,
      STR TYPE STREET,
      END OF ADDRESS.
```

```
TYPES: BEGIN OF PHONE_LIST,
      ADR TYPE ADDRESS,
      TEL TYPE PHONE,
      END OF PHONE_LIST.
```

```
DATA PL TYPE PHONE-LIST.
```

```
DATA: TYP, N TYPE I.
```

```
DESCRIBE FIELD PL TYPE TYP COMPONENTS N.
```

```
WRITE: TYP, N.
```

This example is similar to the last example in the section [The TYPES Statement \[Page 130\]](#). The output is as follows:

```
u      2
```

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

Determining the Output Length

Determining the Output Length

To determine the output length of a field, you use the OUTPUT-LENGTH parameter with the DESCRIBE FIELD statement as follows:

Syntax

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

The system reads the output length of the field <f> and writes the value to the field <o>.

```
DATA: FLOAT TYPE F, OUT TYPE I, LEN TYPE I.
```

```
DESCRIBE FIELD FLOAT LENGTH LEN OUTPUT-LENGTH OUT.
```

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

For further information, see [The WRITE Statement \[Page 891\]](#).

Determining the Decimal Places

To determine the number of decimals for a type P field, you use the DECIMALS parameter with the DESCRIBE FIELD statement as follows:

Syntax

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

The system reads the number of decimals of the field <f> and writes the value to the field <d>.

```
DATA: PACK TYPE P DECIMALS 2, DEC.
```

```
DESCRIBE FIELD PACK DECIMALS DEC.
```

This example results in field DEC containing the value 2.

Determining the Conversion Routine

Determining the Conversion Routine

To determine whether a conversion routine exists for a field in the ABAP Dictionary, and, if so, what it is called, use the EDIT MASK parameter with the DESCRIBE FIELD statement as follows:

Syntax

DESCRIBE FIELD <f> EDIT MASK <m>.

If a conversion routine exists for the field <f> in the ABAP Dictionary, the system writes it to the field <m> and sets the return code value in the system field SY-SUBRC to 0.

You can then use the field <m> directly as a format template in a WRITE statement, as shown in the following:

WRITE <f> USING EDIT MASK <m>.

For further information, see [The WRITE Statement \[Page 891\]](#).

If the field <f> has no conversion routine, the system sets the return code to 4.

Working with Text Elements

The ABAP programming environment allows you to create and maintain programs in several languages. You can store all texts the program outputs to the screen as text elements in text pools. For different languages, you can create your own text pools. When changing texts, you then do not have to change the program code, but just the appropriate text element(s).

In this section you learn about:

[Text Elements - Concept \[Page 147\]](#)

[Creating and Changing Text Elements \[Page 148\]](#)

[Comparing Text Elements \[Page 159\]](#)

[Copying Text Elements \[Page 167\]](#)

[Translating Text Elements \[Page 168\]](#)

Text Elements - Concept

Text elements are the texts that appear on the selection screen or output screen of an ABAP program. They can also be used to replace any literals in ABAP programs.

Text elements comprise

- the title of the program. Titles belong to the program attributes (see [Maintaining Program Attributes \[Page 74\]](#)).
- List titles and headings for the page headers of output lists (see [Generating Complex Lists \[Page 938\]](#))
- The selection texts that appear on selection screens (see [Working With Selection Screens \[Page 795\]](#))
- Text symbols, which can be used in ABAP statements instead of [literals \[Page 113\]](#).

You can store these text elements outside the program in language-dependent text pools. Your program automatically uses the text elements of the user's logon language.

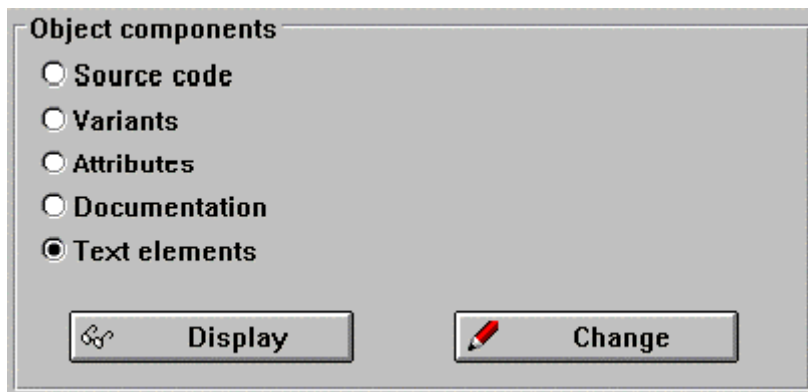
You can create and maintain text elements without changing the program coding. You can create standard text pools, which you can copy from program to program. If you consequently work with text symbols only and do not use string literals in ABAP statements, your program becomes language-independent. Only the text elements from the original language text pool must be translated to other languages.

The translation of text elements is fully supported by the ABAP workbench. From an existing text pool for the original language, a translator can create the text pools for different other languages.

Creating and Changing Text Elements

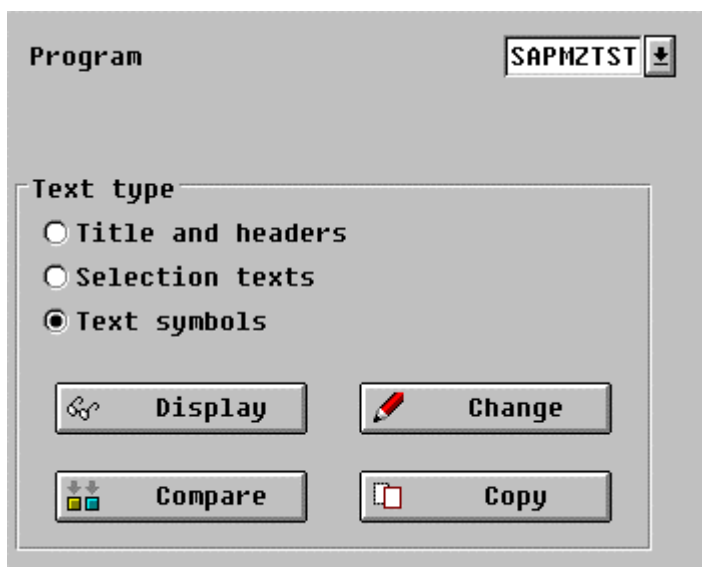
If you want to create or change program specific text elements, you can use several ways.

- Choose *Tools* → *ABAP Workbench* → *Development* → *Programming environ.* → *Text elements*
- In the *Object Browser* of the ABAP Development Workbench, choose *Program objects* for a specific program and then *Text elements*.
- Choose *Goto* → *Text elements* in the ABAP Editor
- On the *ABAP Editor: Initial screen* (SE38), enter in the *Program* field the name of the program for which you want to maintain the text elements.



Select *Text elements* and choose *Display* or *Change*.

All actions take you to the *ABAP Text Elements* screen:



You can now choose what types of text elements you want to maintain for your program.

Creating and Changing Text Elements

If you have changed the source code of your program, but have not yet generated it, the system generates automatically the program to update the relationship between the text elements and the program.

If the logon language differs from the original language of the program, i.e. the language under which the program was created, the following special features apply for all text elements:

- In display mode, you see a warning in the status bar, indicating that the original language is not the same as the logon language. The text elements are displayed in the logon language. If certain text elements exist in the original language, but not in the logon language, they are displayed in the original language and flagged accordingly in column *L*. during the maintenance using the language ID. This allows you to locate untranslated text elements (see example in [Translating Text Elements \[Page 168\]](#)).
- In change mode, the system asks whether you want to maintain the text elements in the original language or whether you want to change the original language. When you change the original language, texts which do not exist in the new original language are taken from the old original language, but are not flagged.

The following topics discuss the different text elements in more detail:

[Titles and Headers \[Page 150\]](#)

[Selection Texts \[Page 154\]](#)

[Text Symbols \[Page 157\]](#)

Titles and Headers

Every program must have a title. You enter the program title when specifying the program attributes (see [Maintaining Program Attributes \[Page 74\]](#)). You may change this title as you wish.

[Changing a Program Title \[Page 151\]](#)

You can create or change the header line for the output list of your program and the column headers for the different columns in the list.

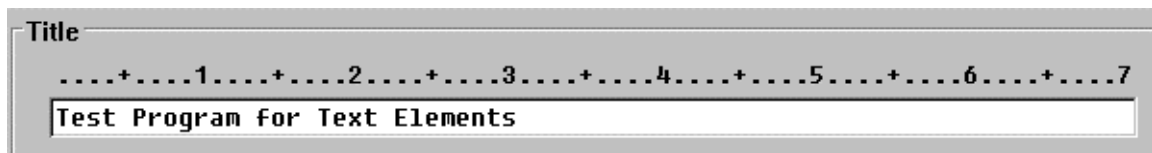
[Creating and Changing List and Column Headers \[Page 152\]](#)

Changing the Title of a Program

Changing the Title of a Program

To change the title of your program, select *Titles and headers* from the *ABAP Text Elements* screen and then choose *Change*.

You can enter a title of up to 70 characters in the *Title* field.



The screenshot shows a dialog box with a title bar. Inside, there is a label 'Title' followed by a text input field. Above the input field, there is a row of seven dots, each followed by a plus sign and a number from 1 to 7, indicating character positions. The input field contains the text 'Test Program for Text Elements'.

Save your changes by choosing *Save*.

Creating and Changing List and Column Headers

To create or change the titles in the output screen, select *Titles and headers* from the *ABAP Text Elements* screen and choose *Change*.

You can enter a list header of up to 70 characters in the *List header* field and column headers of up to 255 characters in the four lines of the *Column header* field.

The screenshot shows the 'List header' and 'Column header' fields in the ABAP Text Elements - Change screen. The 'List header' field contains the text 'List of Squares and Square Roots'. The 'Column header' field contains a table with four rows and two columns. The first row is 'Square', the second row is 'Squares', the third row is 'of', and the fourth row is 'Numbers'. The table is displayed in a grid format. Below the table, there are fields for 'From column' (001) and 'From' (255), and two arrow buttons for navigation.

You can use the options of the *Edit* menu to format the headings. If you do not specify any list header, the program title is displayed on the output screen instead.

Save your changes by choosing *Save*.

In the list and column headers, you can specify up to 10 placeholders &0 to &9 followed by up to 18 dots (.). During the TOP-OF-PAGE event (see [Page Header Layout \[Page 954\]](#)), the system replaces these placeholders with the contents of the system fields SY-TVAR0 to SYTVAR9. The output length of the system fields is given by the length of the placeholders including the dots. In case of a placeholder '&3.....', for example, the contents of SY-TVAR3 is output with a length of 8.

Suppose you have the following program:

```
PROGRAM SAPMZTST.
DATA: NUM1 TYPE I, NUM2 TYPE P DECIMALS 2.
DO 5 TIMES.
  NUM1 = SY-INDEX ** 2. NUM2 = SQRT( SY-INDEX).
  WRITE: / SY-INDEX, NUM1, NUM2.
ENDDO.
```

If the headers are created as shown above, the output screen looks as follows:

Creating and Changing List and Column Headers

Print

Find...

List of Squares and Square Roots

	Squares	Square
	of	Roots
Numbers	Numbers	of
		Numbers
1	1	1,00
2	4	1,41
3	9	1,73
4	16	2,00
5	25	2,24

For include programs (see [INCLUDE Programs \[Page 441\]](#)), you can only maintain the program title.

Selection Texts

You can replace the standard texts that appear for parameters and selection criteria on the selection screen (see [Working with Selection Screens \[Page 795\]](#)) with text elements.

To change the text on the selection screen, select *Selection texts* from the *ABAP Text Elements* screen and choose *Change*.

On the following screen, the column *Name* already contains the names of the parameters and selection criteria of your program (see example below). Now you can enter a selection text of up to 30 characters for each parameter and selection criterion.

If you have created selection criteria for ABAP Dictionary fields, you can choose *Utilities* → *Copy DD texts*. The system fills these selection texts automatically with the short texts assigned as attributes to the ABAP Dictionary fields. The entry *DDIC* occurs in the column *Type* of each line with Dictionary texts. You can mark single lines and choose *Utilities* → *Copy DD text*. Then, the system treats only the marked lines. The text elements with dictionary texts appear in display mode. You can change them after marking the respective lines and choosing *Utilities* → *No DD text*. The text element remains the same, but can now be modified.

If a short text is changed in the ABAP Dictionary after it was copied as a selection text to the text pool, this change is **not** transferred automatically to the text pool. To adjust the dictionary texts in the text pool with the texts in the ABAP Dictionary, you must choose *Utilities* → *Supplement DD texts*.

Save your changes by choosing *Save*.

If you change or delete any parameters or selection criteria in your program after you have maintained the selection texts once and then call selection text maintenance, you see a flagged checkbox in last column on the right (*not used*) for each text no longer required in the program. This facilitates the deletion of any unused selection texts. If you attempt to delete a selection text used in the program, the system outputs a warning message. When you store the selection texts, the system once again notes the existence of any unused texts and offers them to you for deletion.

Assume a program as follows:

```
PROGRAM SAPMZTST.
```

```
TABLES SBOOK.
```

```
PARAMETERS: TEXT(10).
```

```
SELECT-OPTIONS: SEL1 FOR SBOOK-CARRID,  
                SEL2 FOR SBOOK-CONNID.
```

The screen for changing the selection texts looks as follows:

Selection Texts

Program name **SAPMZTST** Lang. **E** English Sort sequence name

Name	L.	Text	Type	n.use
SEL1				
SEL2				
TEXT				

When you choose *Utilities* → *Copy DD texts*, it changes to:

Name	L.	Text	Type	n.use
SEL1		Airline carrier	DDIC	
SEL2		Connection number	DDIC	
TEXT				

Now, you can mark the line with (for example) SEL1 in column *name* and choose *Utilities* → *No DD text* to change its selection text:

Name	L.	Text	Type	n.use
SEL1		Enter airline here:		
SEL2		Connection number	DDIC	
TEXT		Enter comment here:		

The selection text for parameter TEXT is also changed.

After saving the selection texts and starting SAPMZTST, the selection screen looks as follows:

Enter comment here:	<input type="text"/>			
Enter airline here:	<input type="text"/>	to	<input type="text"/>	
Connection number	<input type="text"/>	to	<input type="text"/>	

Now, replace the parameter TEXT in the program by the parameter PARA:

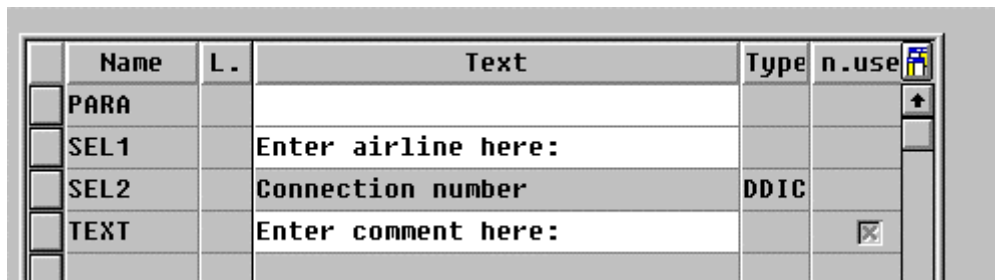
PROGRAM SAPMZTST.



TABLES SBOOK.

PARAMETERS: PARA(10).

SELECT-OPTIONS: SEL1 FOR SBOOK-CARRID,
SEL2 FOR SBOOK-CONNID.

Then, if you call selection text maintenance again, the screen looks as follows after you have generated the program:



	Name	L.	Text	Type	n.use	
<input type="checkbox"/>	PARA					
<input type="checkbox"/>	SEL1		Enter airline here:			
<input type="checkbox"/>	SEL2		Connection number	DDIC		
<input type="checkbox"/>	TEXT		Enter comment here:			

The new parameter PARA is displayed and the old parameter TEXT is flagged as *Not used*. If you save the selection text now, the system displays a dialog window that offers to delete all unused elements.

Text Symbols

Text Symbols

Text symbols are text constants which you enter and maintain outside a program. You should use text symbols instead of text literals in the final version of your program to keep it language-independent and easy to maintain.

Maintaining Text Symbols

To create or change text symbols, select *Text symbols* on the *ABAP Text Elements* screen and choose *Change*.

For each text symbol, you must specify a three-character identifier which contains no blanks and does not begin with the character '% '. You can assign a text of up to 132 characters to each text symbol.

Sym	Text	dLen	mLen	L.
010	This is symbol 010	18	132	
030	This is symbol 030	18	132	
AAA	This is symbol AAA with underscores _____	41	132	

You specify the three-character identifier in column *Sym* and the text in column *Text*.

Starting with Release 3.1g, blanks are not represented by underscores (_) any more and you can output underscores to the screen using text symbols.

Column *dLen* shows the actual length of the text. You can define the maximum length in column *mLen*. A language key in column *L.* shows that there is no text symbol defined for the current logon language but for the original language. The language key shows the original language of the program (see example in [Translating Text Elements \[Page 168\]](#)).

You can use the functions of the *Edit* menu to format text symbols.

To delete a text symbol, mark the line with the text symbol and choose *Delete*.

Save your changes by choosing *Save*.

Using Text Symbols in ABAP Programs

You use text symbols exactly as literals in your ABAP programs. With other words, at each position in a statement where you can write a literal, you can write also a text symbol. To do so, you use the following syntax for the text symbol:

Syntax

```
... TEXT-<idt>...
```

The system searches in the text pool for a text symbol with the identifier <idt> and treats TEXT-<idt> in the statement like literal that contains the text of the text symbol. If the text symbol <idt> does not exist in the text pool, the system treats TEXT-<idt> like the constant SPACE.

To avoid the use of SPACE, you can define a literal in your program that the system can use instead. To do this, use the following syntax:

Syntax

```
... '<text>'(<idt>)...
```

If a text symbol <idt> exists in the text pool, the system uses the text symbol instead of the literal '<text>'. Otherwise, it uses the literal <text>.

The ABAP Development Workbench supports you in creating text symbols as follows:

1. Type your statements with literals in the ABAP Editor.
2. Select the literals by double-clicking with the mouse.

The system creates automatically a text symbol with the contents of the literal and navigates to the maintenance of that text symbol. The entries in the columns *dLen* and *mLen* correspond initially to the length of the literal. You can change *mLen* when maintaining the text symbols.

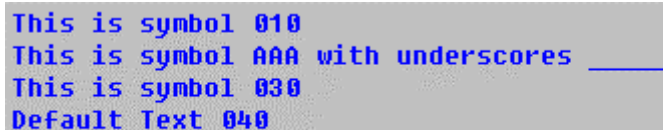
3. Change the text symbol if necessary and navigate back to the ABAP Editor.

The system has then automatically included the number <idt> in parentheses behind the literal. Your program will use the text symbol and not the literal when executing the program.

The following example shows the use of text symbols in WRITE statements. Remember, that you can use text symbols in other statements, for example for value assignments, as well.

```
PROGRAM SAPMZTST.  
WRITE: TEXT-010,  
      / TEXT-AAA,  
      / TEXT-020,  
      / 'Default Text 030'(030),  
      / 'Default Text 040'(040).
```

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



```
This is symbol 010  
This is symbol AAA with underscores _____  
This is symbol 030  
Default Text 040
```

The text symbols 020 and 040 do not exist. In the case of 020, the WRITE statement puts out SPACE to the list. The blank literal is not shown since the system by default suppresses blank lines (see [Inserting Blank Lines \[Page 1012\]](#)). In the case of 040, the text defined in the program is output to the screen.

Comparing Text Elements

By choosing the function *Compare* on the *ABAP Text Elements* screen, you can adapt the text elements of the program to the source code. This function is possible for *Selection texts* and *Text symbols* but not for *Titles and headers*.

[Comparing Selection Texts \[Page 160\]](#)

[Comparing Text Symbols \[Page 164\]](#)

Comparing Selection Texts

When you choose *Selection texts* and *Compare* on the *ABAP Text Elements* screen, the system supports you by finding missing or unused selection texts.

The function *Compare* enhances the functionality of the normal maintenance of selection texts but does not include the possibility to work with texts from the ABAP Dictionary (see [Selection Texts \[Page 154\]](#)).

The following example shows how to work with the function *Compare* with selection texts.

Assume the following program:

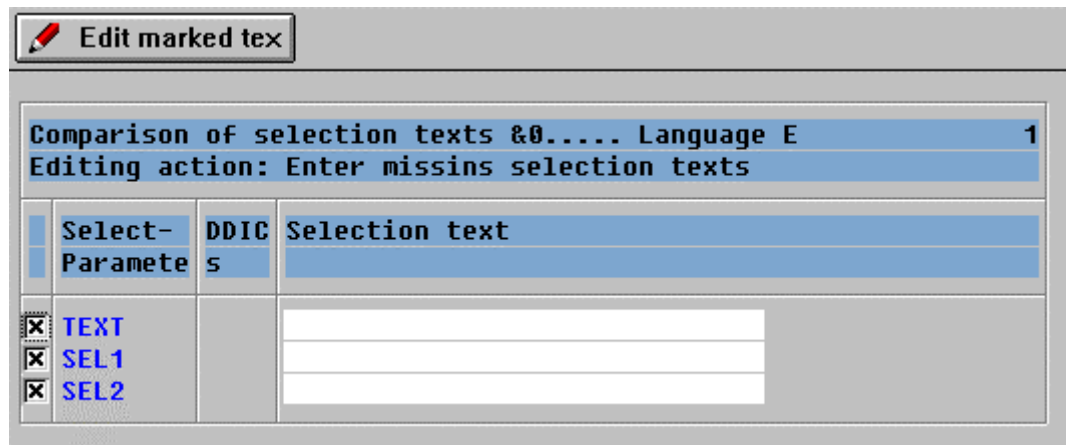
```
PROGRAM SAPMZTST.
```

```
TABLES SBOOK.
```

```
PARAMETERS: TEXT(10).
```

```
SELECT-OPTIONS: SEL1 FOR SBOOK-CARRID,  
                SEL2 FOR SBOOK-CONNID.
```

If you have not maintained the selection texts for this program yet and you choose the function *Compare* for *Selection texts* on the *ABAP Text Elements* screen, the following screen appears:



Edit marked text

Comparison of selection texts &0..... Language E 1
Editing action: Enter missing selection texts

	Select-Parameters	DDIC	Selection text
<input checked="" type="checkbox"/>	TEXT		
<input checked="" type="checkbox"/>	SEL1		
<input checked="" type="checkbox"/>	SEL2		

This screen shows for which parameters and select options the selection texts are missing. You can maintain these texts either in the normal selection text maintenance (for example to include texts from the ABAP Dictionary, see [Selection Texts \[Page 154\]](#)) or define them right here, for example, as follows:

Comparing Selection Texts

Edit marked text			
Comparison of selection texts &0..... Language E			1
Editing action: Enter missins selection texts			
<input type="checkbox"/>	Select-Parameters	DDIC	Selection text
<input checked="" type="checkbox"/>	TEXT		
<input checked="" type="checkbox"/>	SEL1		Airline
<input type="checkbox"/>	SEL2		Connection number

By choosing the function *Edit marked text*, you designate the changed and marked lines for adjustment.

Edit marked text			
Comparison of selection texts &0..... Language E			1
Editing action: Enter missins selection texts			
<input type="checkbox"/>	Select-Parameters	DDIC	Selection text
<input checked="" type="checkbox"/>	SEL2		
<input checked="" type="checkbox"/>	TEXT		
Comparison of selection texts &0..... Language E			2
Selection text with no reference to Dictionary			
<input type="checkbox"/>	Select-Parameters	DDIC	Selection text
	SEL1		Airline

In the above figure, SEL1 and SEL2 are changed but only SEL1 is marked. Therefore, only SEL1 is designated for adjustment. If you choose *Save now*, the selection text for SEL1 is saved in the text pool.

If you have maintained selection texts for all input fields of the selection screen and you choose *Compare*, you obtain for example the following screen:

Comparing Selection Texts

Comparison of selection texts &0..... Language E				1
Selection text with no reference to Dictionary				
	Select-Parameters	DDIC	Selection text	
	SEL1		Airline	
	SEL2		Connection number	
	TEXT		Comment	
>>> No errors in selection texts! <<<				

Now, replace the parameter TEXT in the program by the parameter PARA:

PROGRAM SAPMZTST.


TABLES SBOOK.

PARAMETERS: PARA(10).

SELECT-OPTIONS: SEL1 FOR SBOOK-CARRID,
SEL2 FOR SBOOK-CONNID.

Again, choose the function *Compare* for the selection texts. After generating the program, the system display the following screen:

Comparing Selection Texts

 Edit marked text

Comparison of selection texts &0..... Language E			1
Editing action: Enter missins selection texts			
<input type="checkbox"/>	Select- Paramete	DDIC s	Selection text
<input checked="" type="checkbox"/>	PARA		

Comparison of selection texts &0..... Language E			2
Editing action: Delete superfluous selection texts			
<input type="checkbox"/>	Select- Paramete	DDIC s	Selection text
<input checked="" type="checkbox"/>	TEXT		Comment

Comparison of selection texts &0..... Language E			3
Selection text with no reference to Dictionary			
<input type="checkbox"/>	Select- Paramete	DDIC s	Selection text
<input type="checkbox"/>	SEL1 SEL2		Airline Connection number

The system offers you to define a new selection text for PARA and to delete the selection text for TEXT.

Compare this with the example in the [Selection Texts \[Page 154\]](#) section.

Comparing Text Symbols

If you insert new text symbols or change existing ones in the program code, these text symbols are not automatically copied to the text pool. To update this list and to eliminate any discrepancies, choose *Text symbols* and the function *Compare* on the *ABAP Text Elements* screen.

After calling this function for a program, the system displays a screen showing all the text symbols which occur in the program code with their texts as defined in the program and in the text pool. On the right, you can see whether these texts are the same ("P=T") in the program and in the text pool. If the texts are not the same, the texts in the text pool are flagged with T and the texts defined in the program are flagged with P.

In the column +/- you can specify

- "-", to delete the text symbol from the text pool
- "+", to add the text symbol to the text pool
- " ", to keep the text symbol unchanged

If you choose *Adjust*, the system refreshes the text symbols in the text pool according to these settings. It does not change the program code.

The system offers you a further possibility for comparing text symbols. You can double-click on literals in WRITE statements where the numbers of text symbols are given in parentheses. If the contents of literal and text symbol are not the same, the system asks you, whether you want to replace the contents of the text symbol with the contents of the literal.

The following example shows how to work with the function *Compare* with text symbols.

Write the following program:

```
PROGRAM SAPMZTST.
```

```
WRITE: TEXT-010,  
      'Placeholder'(020),  
      TEXT-030.
```

Create no text symbols. The text symbol list is empty. After choosing *Compare* for *Text symbols* on the *ABAP Text Elements* screen, the system generates the program (if necessary) and displays the following picture:

Comparing Text Symbols

Adjust

Comparison of text symbols of source code and text pool for TEXTTEST Language E

Unpaired text no. - not defined in the text pool
 T/P +/- Symbol Text (P=Program/T=Text pool += Insert textp./-= Delete textp.)

P	+	010	
P	+	020	Placeholder
P	+	030	

The text symbols 010, 020, and 030 specified in the program are not defined in the text pool. To copy all the text symbols to the text pool, keep the symbols "+" in the column +/- and choose *Adjust*. When you have done this, the text symbol list looks as follows in display mode:

	Sym	Text	dLen	nLen	L.	
	010		0	1		+
	020	Placeholder	11	11		
	030		0	1		

The system has created the text symbols 010, 020, and 030 in the text pool. It has assigned no text to 010 and 030, and the text defined in the program to 020. Now you can change the text symbols as described in [Text Symbols \[Page 157\]](#).

Change the program code to

PROGRAM TEXTTEST.

WRITE: TEXT-010,
 'Placeholder'(020),
 'Text Symbol'(030).

Again choose *Compare* for *Text symbols* on the *ABAP Text Elements* screen. Now the comparison looks as follows:

Adjust			
Comparison of text symbols of source code and text pool for TEXTTEST Language E			
Correctly defined numbered texts			
T/P	+/-	Symbol	Text (P=Program/T=Text pool += Insert textp./-= Delete textp.)
P=T		010	
P=T		020	Placeholder
Comparison of text symbols of source code and text pool for TEXTTEST Language E			
Paired text no. contains different texts in program and text pool			
The text no. is referenced only once in the program			
T/P	+/-	Symbol	Text (P=Program/T=Text pool += Insert textp./-= Delete textp.)
P	+	030	Text_Symbol
T	-	030	

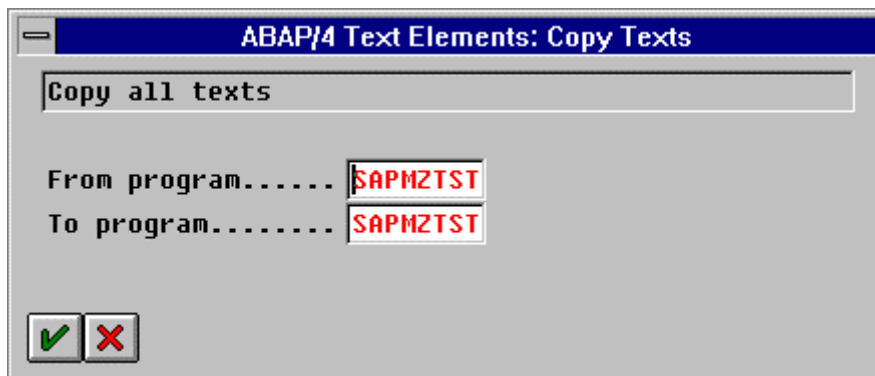
First, you see the two correctly defined text symbols 010 and 020 and then the message that the texts are different for text symbol 030 in the text pool and in the program. The "+" and "-" symbols allow you to define which text you want to keep.

Copying Text Elements

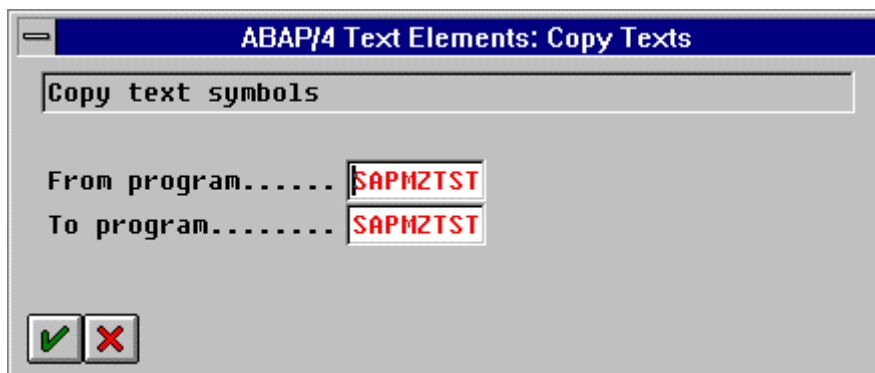
Copying Text Elements

You can copy text elements from one ABAP program to another. The copy function allows you to transport text pools with your own standard text elements which you want to use with different programs.

- To copy all text elements of a program, choose *Copy* in the push-button bar on the *ABAP Text Elements* screen.



- If you want to copy only certain text elements from the source program, mark these text elements with the radio button and choose *Copy text type* in the frame *Text type*. Example for copying text symbols:



Translating Text Elements

To translate text elements into other languages, start on the *ABAP Development Workbench* screen and choose *Utilities* → *Translation* → *Short/long texts*. On the subsequent *Translate Short/Long Texts* screen, choose *Translation* → *Short texts* → *Program* and then select the text element you want to translate. The following screen appears:

Enter the name of the program to which the text elements belong, the source and the target languages, and choose *Edit*.

The text elements are displayed in the language in which they were entered. You can enter remarks and use the line below the text elements to enter the translation (see example below).

After translating all text elements of the current type, use the functions in the *Goto* menu to translate the text elements of the other types.

After translating all text elements, save your translations by choosing *Save*.

In this way, you can create complete text pools in different languages.

If text pools of different languages exist, you can influence the output language of your program with

- **the logon language:** By default, the system automatically uses the logon language of the user.
- **the SET LANGUAGE statement:** With this ABAP statement, you can specify the output language within the program explicitly, regardless of the logon language.

Syntax

SET LANGUAGE <lg>.

The language <lg> can be a literal or a variable.

When you select a certain language (either through the logon language or explicit specification), the system searches in the text pool of that language for the text elements included in the program. If it does not find a text symbol in this text pool, it outputs the text in the program code to the screen or skips the relevant WRITE statement (see [Text Symbols \[Page 157\]](#)). The system does not use text elements from other languages.

Suppose you have the following program with the original language English.

```
PROGRAM TRANTEST.
```

```
WRITE TEXT-010.
```


Translating Text Elements

and the output looks as follows:

Output List
Welcome
Hello, how are you?

If you log on to the system in German and no English text symbols exist, you see the following line when the text symbols of the program are displayed:

Sym	Text	dLen	mLen	Spr
010	Hello, how are you?	19	132	E

The character "E" in the column *Spr* indicates that an English text symbol 010 exists.

The text elements are translated from English to German as follows in the ABAP Development Workbench.

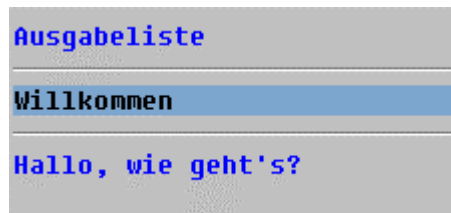
Titles and headings:

Title
Remark: translated 08/07/95
Test for Translation
Übersetzungstest
List heading
Remark:
Output List
Ausgabeliste
Column header 001
Remark:
Welcome
Willkommen

Text symbols:

010
Remark: translated 08/07/95 by HK
Hello, how are you?
Hallo, wie geht's?

If you logon to the R/3 system and specify the language "D", the output of the program TRANTEST looks as follows:



You can fix the output language of the program TRANTEST to German by changing the program code as follows.

```
PROGRAM TRANTEST.
```

```
SET LANGUAGE 'D'.
```

```
WRITE TEXT-010.
```

Now, the output is always German, regardless of the logon language.

Processing Data

This section describes how to work with (process) data objects. The following topics are covered:

[Assigning Values \[Page 172\]](#)

[Resetting Values to Initial Values \[Page 184\]](#)

[Numeric Operations \[Page 185\]](#)

[Processing Character Strings \[Page 199\]](#)

[Specifying Offset Values for Data Objects \[Page 216\]](#)

[Type Conversions \[Page 218\]](#)

Assigning Values

In ABAP, you can assign values to data objects in both declarative and operational statements.

In declarative statements, you assign start values to data objects on declaration. To do this, you can use the VALUE parameter in the DATA, CONSTANTS, or STATICS statements (see [Creating Data Objects and Data Types \[Page 118\]](#)).

To assign values to data objects in operational statements, you can use:

- the MOVE statement, which corresponds to the assignment operator (=)

[Assigning Values with MOVE \[Page 173\]](#)

- the WRITE TO statement

[Assigning Values with WRITE TO \[Page 179\]](#)

Most of the operations covered in this section apply not only to internal fields in programs, but also to program parameters, table work areas, system fields, field symbols, and formal parameters, and even constants and literals in cases where the data object is not meant to be changed. When discussing operations on fields, the reference is to fields in general, not just internal fields.

Assigning Values with MOVE

The topics in this section describe how to use the MOVE statement or the assignment operator (=). They include:

[Basic Assignments \[Page 174\]](#)

[Assigning Values with Offset Specifications \[Page 176\]](#)

[Copying Values between Components of Structures \[Page 178\]](#)

Basic Assignment Operations

To assign either a value (literal) or the contents of a source field to a target field, you can use the MOVE statement or the assignment operator (=).

The syntax for the MOVE statement is as follows:

Syntax

```
MOVE <f1> TO <f2>.
```

The MOVE statement transports the contents of a source field <f1>, which can be any data object, to a target field <f2>, which must be a variable. <f2> cannot be a literal or a constant. The contents of <f1> remain unchanged.

The syntax for the assignment operator (=) is as follows:

Syntax

```
<f2> = <f1>.
```

The MOVE statement and the assignment operator have the same function.

Multiple assignments such as

```
<f4> = <f3> = <f2> = <f1>.
```

are possible. ABAP processes them from right to left as follows:

```
MOVE <f1> TO <f2>.
```

```
MOVE <f2> TO <f3>.
```

```
MOVE <f3> TO <f4>.
```

In the above statements, decimal points must always be specified with a period (.), regardless of the user's master record.

The source and target fields can be of different data types. In contrast to other programming languages, where the assignment between different data types is often restricted to a small number of possible combinations, ABAP provides a wide range of automatic type conversions.

For example, you can assign the contents of a source field with an elementary data type to a target field with any other elementary data type (except data type D which cannot be assigned to data type T and vice versa). ABAP also supports assignment between structured data and elementary data objects or between data objects which have different structures.

For each assignment statement (with MOVE or the assignment operator), the system checks the data types of the source and target fields. If a type conversion is defined for that combination, it converts the contents of the source field to the data type of the target field and assigns it to the target field. For an overview of the possible type conversions and how they are defined in ABAP, see [Type Conversions \[Page 218\]](#).

```
DATA: T(10),  
      NUMBER TYPE P DECIMALS 1,  
      COUNT TYPE P DECIMALS 1.  
  
T = 1111.  
MOVE '5.3' TO NUMBER.  
COUNT = NUMBER.
```

Basic Assignment Operations

The result of this assignment is that the fields T, NUMBER, and COUNT contain the values '1111', 5.3, and 5.3. Note that, when assigning the number literal 1111, the system converts it to a character string with a length of 10.

You cannot use the MOVE statement (or the assignment operator) for this purpose. Instead, you must use field symbols (see the description in the [Working With Field Symbols \[Page 336\]](#) section).

Assigning Values with Offset Specifications

You can specify offset and length for elementary data objects in any ABAP statement (see [Specifying Offset Values for Data Objects \[Page 216\]](#)). In this case, the syntax of the MOVE statement is as follows:

Syntax

MOVE <f1>[+<o1>][(<l1>)] TO <f2>[+<o2>][(<l2>)].

The syntax of the assignment operator is as follows:

Syntax

<f2>[+<o2>][(<l2>)] = <f1>[+<o1>][(<l1>)].

The contents of the section of field <f1>, which begins at the position <o1>+1 and has a length of <l1>, are assigned to field <f2> where they overwrite the section which begins at the position <o2>+1 and has a length of <l2>.

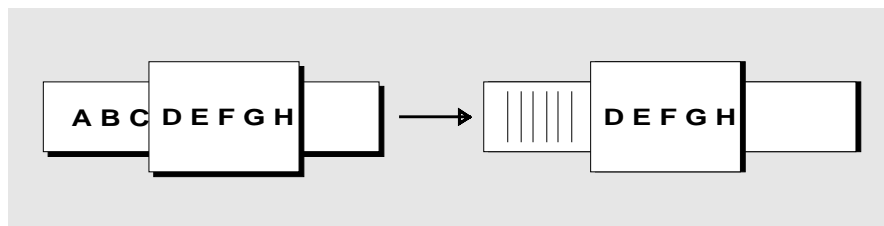
In the MOVE statement, all offset and length specifications can be variables. This also applies to statements with the assignment operator, as long as these can also be written as MOVE statements. In statements where no field name is specified after the assignment operator, (for example, in mathematical expressions), all offset and length specifications must be unsigned number literals. For further information, see [Numeric Operations \[Page 185\]](#).

SAP recommends that you assign values with offset and length specifications only between non-numeric fields. With numeric fields, the results can be meaningless.

```
DATA: F1(8) VALUE 'ABCDEFGH',
      F2(20).
```

```
F2+6(5) = F1+3(5).
```

In this example, the assignment operator functions as follows:



```
DATA: F1(8) VALUE 'ABCDEFGH',
      F2(8).
```

```
DATA: O TYPE I VALUE 2,
      L TYPE I VALUE 4.
```

```
MOVE F1 TO F2. WRITE F2.
MOVE F1+O(L) TO F2. WRITE / F2.
MOVE F1 TO F2+O(L). WRITE / F2.
```

Assigning Values with Offset Specifications

```
CLEAR F2.  
MOVE F1    TO F2+O(L). WRITE / F2.  
MOVE F1+O(L) TO F2+O(L). WRITE / F2.
```

This produces the following output:

```
ABCDEFGH
```

```
CDEF
```

```
CDABCD
```

```
  ABCD
```

```
    CDEF
```

First, the contents of F1 are assigned to F2 without offset specifications. Then, the same happens for F1 with offset and length specification. The next three MOVE statements overwrite the contents of F2 with offset 2. Note that F2 is filled with spaces on the right, in accordance with the rule listed in [Source Type Character \[Page 220\]](#).

Copying Values between Components of Structures

The value assignments described for the MOVE statement apply to both elementary and structured data objects. In addition, there is a variant of the MOVE statement which allows you to copy the contents of components of a source structure to the components of a target structure. The syntax is as follows:

Syntax

MOVE-CORRESPONDING <string1> TO <string2>.

This statement assigns the contents of components of structure <string1> to those components of structure <string2> which have the same names.

The system executes a MOVE statement for each name pair as follows:

MOVE STRING1-<component> TO STRING2-<component>.

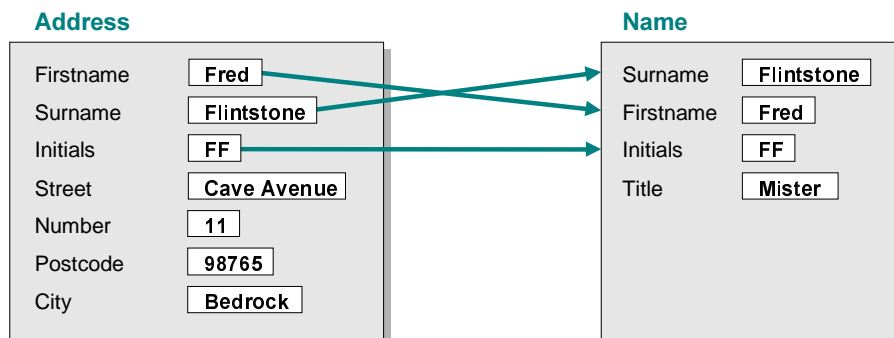
The system performs all the necessary type conversions separately. This process differs from value assignment involving complete structures. In this case, the conversion rules described in [Incompatible Structures and Elementary Fields \[Page 229\]](#) apply.

```
DATA: BEGIN OF ADDRESS,
      FIRSTNAME(20) VALUE 'Fred',
      SURNAME(20) VALUE 'Flintstone',
      INITIALS(4) VALUE 'FF',
      STREET(20) VALUE 'Cave Avenue',
      NUMBER TYPE I VALUE '11',
      POSTCODE TYPE N VALUE '98765',
      CITY(20) VALUE 'Bedrock',
      END OF ADDRESS.
```

```
DATA: BEGIN OF NAME,
      SURNAME(20),
      FIRSTNAME(20),
      INITIALS(4),
      TITLE(10) VALUE 'Mister',
      END OF NAME.
```

MOVE-CORRESPONDING ADDRESS TO NAME.

In this example, the values of NAME-SURNAME, NAME-FIRSTNAME and NAME-INITIALS are set to 'Flintstone', 'Fred', and 'FF'. NAME-TITLE retains the value 'Mister'.



Assigning Values with WRITE TO

Assigning Values with WRITE TO

When assigning values to data objects with the WRITE TO statement, you can use the formatting options of the WRITE statement (see [The WRITE Statement \[Page 891\]](#)).

The topics in this section describe

[Basic Form of the WRITE TO Statement \[Page 180\]](#)

[Specifying the Source Field at Runtime \[Page 182\]](#)

[Writing Values with Offset Specifications \[Page 183\]](#)

Basic Form of the WRITE TO Statement

To write a value (literal) or the contents of a source field to a target field, you use the WRITE TO statement:

Syntax

WRITE <f1> TO <f2> [<option>].

The WRITE TO statement writes the contents of a source field <f1>, which can be any data object, to a target field <f2>, which must be a variable. <f2> cannot be a literal or a constant. The contents of <f1> remain unchanged.

For <option>, you can use all formatting options of the WRITE statement except UNDER and NO-GAP (see [Formatting Options \[Page 896\]](#)).

The WRITE TO statement always checks the settings in the user's master record. These specify, for example, whether the decimal point appears as a period (.) or a comma (,).

The WRITE TO statement does not follow the conversion rules described in [Type Conversions \[Page 218\]](#). The target field is interpreted as a type C field. The system always converts the contents of the source field to type C. It then writes the resulting character string to the target field without converting it to the data type of the target field. Therefore, you should not use a target field with a numeric data type.

```
DATA: NUMBER TYPE F VALUE '4.3',
      TEXT(10),
      FLOAT TYPE F,
      PACK TYPE P DECIMALS 1.
```

```
WRITE NUMBER.
```

```
WRITE NUMBER TO TEXT EXPONENT 2.
WRITE / TEXT.
```

```
WRITE NUMBER TO FLOAT.
WRITE / FLOAT.
```

```
WRITE NUMBER TO PACK.
WRITE / PACK.
```

```
MOVE NUMBER TO PACK.
WRITE / PACK.
```

This produces the following output:

```
4.300000000000000E+00
0.043E+02
1.50454551753894E-153
20342<33452;30,3
4.3
```

The first output line displays the contents of the field NUMBER in the standard output format for type F fields. The second output line displays the character string which results from writing the field NUMBER with the formatting option EXPONENT 2 to a type C field with a length of 10. The third and fourth output

Basic Form of the WRITE TO Statement

lines show that it is not feasible to use target fields which are of numeric data type. The fifth output line shows that the MOVE statement works differently than the WRITE TO statement in that the type F field is correctly converted to type P (for further information about this conversion, see [Source Type Floating Point Number \[Page 222\]](#)).

Specifying the Source Field at Runtime

You can use the WRITE TO statement to specify the source field at runtime. To do this, enclose the name of the data object which contains the name of the source field in parentheses and place it in the source field position:

Syntax

WRITE (<f>) TO <g>.

The system places the value of the data object assigned to <f> in <g>.

However, if you want to specify target fields at runtime, you must use field symbols as described in [Working with Field Symbols \[Page 336\]](#).

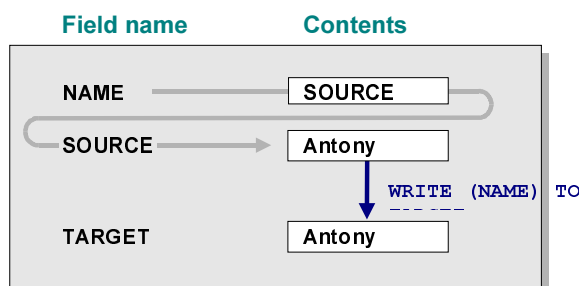
```
DATA: NAME(10) VALUE 'SOURCE',  
      SOURCE(10) VALUE 'Antony',  
      TARGET(10).
```

```
...  
WRITE (NAME) TO TARGET.  
WRITE: TARGET.
```

This produces the output

Antony

The connection between field names and field contents is shown in the following diagram.



Writing Values with Offset Specifications

Writing Values with Offset Specifications

You can specify offset and length for elementary data objects in any ABAP statement (see [Specifying Offset Values for Data Objects \[Page 216\]](#)). The syntax of the WRITE TO statement is as follows:

Syntax

```
WRITE <f1>[+<o1>][(<l1>)] TO <f2>[+<o2>][(<l2>)].
```

The contents of the part of the field <f1> which begins at position <o1>+1 and has a length of <l1> are assigned to field <f2> where they overwrite the section which begins at position <o2>+1 and has a length of <l2>.

In the WRITE TO statement, the offset and length specifications of the target field can be variables.

SAP recommends that you assign values with offset and length specifications only between non-numeric fields. For numeric fields, the results can be meaningless.

```
DATA: STRING(20),
      NUMBER(8) TYPE C VALUE '123456',
      OFFSET TYPE I VALUE 8,
      LENGTH TYPE I VALUE 12.
```

```
WRITE NUMBER+(6) TO STRING+OFFSET(LENGTH) LEFT-JUSTIFIED.
WRITE: / STRING.
CLEAR STRING.
```

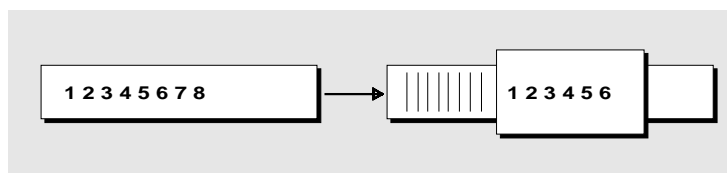
```
WRITE NUMBER+(6) TO STRING+OFFSET(LENGTH) CENTERED.
WRITE: / STRING.
CLEAR STRING.
```

```
WRITE NUMBER TO STRING+OFFSET(LENGTH) RIGHT-JUSTIFIED.
WRITE: / STRING.
CLEAR STRING.
```

This produces the following output:

```
123456
      123456
            123456
```

The first WRITE statement writes the first 6 positions of the field NUMBER left-justified to the last 12 positions of the field STRING.



The second WRITE statement writes the first 6 positions of NUMBER centered to the last 12 positions of STRING.

Resetting Values to Initial Values

You can reset the values of any data objects with the CLEAR statement as follows:

Syntax

CLEAR <f>.

This statement resets the contents of the data object <f> to its initial value. You can distinguish between

- elementary data types

The system resets the values of variables to their initial values, not to the start value which you assigned with the VALUE parameter of the DATA statement. For a list of the initial values of data objects with elementary data types, see the table in [Predefined Elementary Data Types \[Page 106\]](#). You cannot reset constants.

- Structures

If you apply the CLEAR statement to a structure, the contents of the individual components are reset to their initial values.

- internal tables

For an explanation of how the CLEAR statement works with internal tables, see [Initializing Internal Tables \[Page 334\]](#).

```
DATA NUMBER TYPE I VALUE '10'.
```

```
WRITE NUMBER.
```

```
CLEAR NUMBER.
```

```
WRITE / NUMBER.
```

The output appears as follows:

```
10
```

```
0
```

The CLEAR statement resets the contents of the field NUMBER from 10 to its initial value 0.

Numerical Operations

Numerical Operations

To process numeric data objects and assign the resulting value to a data object, you can use either the COMPUTE statement or the assignment operator (=).

The syntax for the COMPUTE statement is as follows:

Syntax

COMPUTE <n> = <expression>.

The use of the keyword COMPUTE is optional. In other words, you could also write the statement as follows:

Syntax

<n> = <expression>.

The effect of both statements is identical.

The result of the mathematical operation specified in <expression> is assigned to the field <n>. The assignment operator (=) works as described in the [Basic Assignments \[Page 174\]](#) section.

ABAP executes a numerical operation with a numerical precision that corresponds to one of the numerical data types I, P, or F. The numerical precision is defined by the operand of the numerical operation that has the highest hierarchy level. The system treats the target field <n> and floating-point functions **also** as operands. ABAP converts all numbers of a numerical operation **before** its execution into the hierarchically highest data type that occurs in the operation. Then the system executes the operation with the converted numbers (see [Type Conversions \[Page 218\]](#)). If necessary, the system converts the result back to the data type of the target field after the execution.

In ABAP, the sequence of type conversions during numerical operations is different from other programming languages. Especially, you can influence the numerical precision of the whole operation with the data type of the target field.

In mathematical expressions, you can combine operations in any permutation and specify them in parentheses.

The order of evaluation is:

1. expressions in parentheses
2. functions
3. ** (exponentiation)
4. *, /, MOD, DIV (multiplications, divisions)
5. +, - (additions, subtractions)

The mathematical operations specified in <expression> are described in the following topics:

[Performing Arithmetic Operations \[Page 187\]](#)

[Using Mathematical Functions \[Page 192\]](#)

The processing of the following data types is of special interest:

[Processing Packed Numbers \[Page 196\]](#)

[Processing Date and Time Fields \[Page 197\]](#)

Performing Arithmetic Operations

To define an arithmetic operation, you use the appropriate arithmetic operators. You use arithmetic operators for

[Basic Arithmetic \[Page 188\]](#)

[Performing Arithmetic Operations on Structures \[Page 190\]](#)

[Adding Sequences of Fields \[Page 191\]](#)

Basic Arithmetic Operations

ABAP supports the four basic arithmetic operations, as well as power calculation. You can specify the following arithmetic operators in a mathematical expression:

+	Addition
–	Subtraction
*	Multiplication
/	Division
DIV	Integer division
MOD	Remainder of integer division
**	Exponentiation

Instead of using operators in mathematical expressions, you can perform basic arithmetic operations with the keywords ADD, SUBTRACT, MULTIPLY, and DIVIDE.

The following table shows how you can express the basic arithmetic operations in ABAP:

Operation	Statement using mathematical expression	Statement using Keyword
Addition	$\langle p \rangle = \langle n \rangle + \langle m \rangle.$	ADD $\langle n \rangle$ TO $\langle m \rangle.$
Subtraction	$\langle p \rangle = \langle m \rangle - \langle n \rangle.$	SUBTRACT $\langle n \rangle$ FROM $\langle m \rangle.$
Multiplication	$\langle p \rangle = \langle m \rangle * \langle n \rangle.$	MULTIPLY $\langle m \rangle$ BY $\langle n \rangle.$
Division	$\langle p \rangle = \langle m \rangle / \langle n \rangle.$	DIVIDE $\langle m \rangle$ BY $\langle n \rangle.$
Integer division	$\langle p \rangle = \langle m \rangle \text{ DIV } \langle n \rangle.$	---
Remainder of division	$\langle p \rangle = \langle m \rangle \text{ MOD } \langle n \rangle.$	---
Exponentiation	$\langle p \rangle = \langle m \rangle ** \langle n \rangle.$	---

In the statements using keywords, the results of the operations are assigned to field $\langle m \rangle$.

The operands $\langle m \rangle$, $\langle n \rangle$, $\langle p \rangle$ can be any numeric fields. If the fields are not of the same data type, the system converts all fields into the hierarchically highest data type that occurs in the statement.

When using mathematical expressions, please note that the operators +, -, *, **, and /, as well as opening and closing parentheses, are ABAP words and must therefore be preceded and followed by blanks.

In division operations, the divisor cannot be zero if the dividend is not zero. With integer division, you use the operators DIV or MOD instead of /. Use DIV to obtain the integer quotient and MOD to obtain the remainder.

If you combine several mathematical expressions together, calculations are performed from left to right for operators of equal priority, except in the case of exponentiation which is performed from right to left. Therefore, $\langle n \rangle ** \langle m \rangle ** \langle p \rangle$ is the same as $\langle n \rangle ** (\langle m \rangle ** \langle p \rangle)$ and not the same as $(\langle n \rangle ** \langle m \rangle) ** \langle p \rangle$.

Basic Arithmetic Operations

```
DATA: COUNTER TYPE I.  
COMPUTE COUNTER = COUNTER + 1.  
COUNTER = COUNTER + 1.  
ADD 1 TO COUNTER.
```

Here, the three operational statements perform the same arithmetic operation, i.e. adding 1 to the contents of the field COUNTER and assigning the result to COUNTER.

```
DATA: PACK TYPE P DECIMALS 4,  
      N TYPE F VALUE '+5.2',  
      M TYPE F VALUE '+1.1'.
```

```
PACK = N / M.  
WRITE PACK.
```

```
PACK = N DIV M.  
WRITE / PACK.
```

```
PACK = N MOD M.  
WRITE /PACK.
```

The output appears as follows:

```
4.7273  
4.0000  
0.8000
```

This example shows the different types of division.

Performing Arithmetic Operations on Structures

Similar to copying values between structures with the MOVE-CORRESPONDING statement (see [Copying Values Between Components of Structures \[Page 178\]](#)), you can perform arithmetic operations with structures using the following keywords:

- ADD-CORRESPONDING
- SUBTRACT-CORRESPONDING
- MULTIPLY-CORRESPONDING
- DIVIDE-CORRESPONDING

ABAP performs the corresponding arithmetic operations on all structure components of the same name. However, these operations only make sense if all the components concerned have a numeric data type.

For further information about these keywords, see the ABAP keyword documentation.

```
DATA: BEGIN OF RATE,  
      USA TYPE F VALUE '0.6667',  
      FRG TYPE F VALUE '1.0',  
      AUT TYPE F VALUE '7.0',  
      END OF RATE.  
  
DATA: BEGIN OF MONEY,  
      USA TYPE I VALUE 100,  
      FRG TYPE I VALUE 200,  
      AUT TYPE I VALUE 300,  
      END OF MONEY.  
  
MULTIPLY-CORRESPONDING MONEY BY RATE.  
  
WRITE / MONEY-USA.  
WRITE / MONEY-FRG.  
WRITE / MONEY-AUT.
```

The output appears as follows:

```
67  
200  
2,100
```

Here, MONEY-USA is multiplied by RATE-USA and so on.

Adding Sequences of Fields

Adding Sequences of Fields

Besides basic addition described in [Basic Arithmetic Operations \[Page 188\]](#), the ADD statement has some variants for adding sequences of fields. For example:

- Adding sequences of fields and assigning the result to another field

Syntax

ADD <n₁> THEN <n₂> UNTIL <n_z> GIVING <m>.

If <n₁>, <n₂>, ..., <n_z> is a sequence of equidistant fields of the same type and length in memory, they are summed and the result is assigned to <m>.

- Adding sequences of fields and adding the result to the contents of another field

Syntax

ADD <n₁> THEN <n₂> UNTIL <n_z> TO <m>.

This statement works exactly like the previous one, but with the exception that the sum of the field values is added to the old contents of <m>.

For information about other similar variants, see the keyword documentation of the ADD statement.

```
DATA: BEGIN OF SERIES,  
      N1 TYPE I VALUE 10,  
      N2 TYPE I VALUE 20,  
      N3 TYPE I VALUE 30,  
      N4 TYPE I VALUE 40,  
      N5 TYPE I VALUE 50,  
      N6 TYPE I VALUE 60,  
      END OF SERIES.
```

```
DATA SUM TYPE I.
```

```
ADD SERIES-N1 THEN SERIES-N2 UNTIL SERIES-N5 GIVING SUM.  
WRITE SUM.
```

```
ADD SERIES-N2 THEN SERIES-N3 UNTIL SERIES-N6 TO SUM.  
WRITE / SUM.
```

Output

150

350

Here, the contents of components N1 to N5 are summed and assigned to the field SUM. Then, the contents of components N2 to N6 are summed and added to the value of SUM.

Using Mathematical Functions

To specify a mathematical expression, you can choose an operation from a set of built-in ABAP functions.

Syntax

[COMPUTE] <n> = <func>(<m>).

The blanks between the parentheses and the argument <m> are obligatory.

The result of calling the function <func> with the argument <m> is assigned to <n>.

The available functions are described in the following topics:

[Functions for all numeric data types \[Page 193\]](#)

[Floating-Point Functions \[Page 195\]](#)

Functions for all numeric data types

Functions for all numeric data types

The following built-in functions work with all three numeric data types (F, I, and P) as an argument.

Functions for all numeric data types

Function	Result
ABS	Absolute value of argument.
SIGN	Sign of argument: $\begin{array}{ll} 1 & X > 0 \\ \text{SIGN}(X) = 0 & \text{if } X = 0 \\ -1 & X < 0 \end{array}$
CEIL	Smallest integer value not smaller than the argument.
FLOOR	Largest integer value not larger than the argument.
TRUNC	Integer part of argument.
FRAC	Fraction part of argument.

```
DATA N TYPE P DECIMALS 2.
DATA M TYPE P DECIMALS 2 VALUE '-5.55'.
```

```
N = ABS( M). WRITE: 'ABS: ', N.
N = SIGN( M). WRITE: / 'SIGN: ', N.
N = CEIL( M). WRITE: / 'CEIL: ', N.
N = FLOOR( M). WRITE: / 'FLOOR:', N.
N = TRUNC( M). WRITE: / 'TRUNC:', N.
N = FRAC( M). WRITE: / 'FRAC: ', N.
```

The output appears as follows:

```
ABS:      5.55
SIGN:     1.00-
CEIL:     5.00-
FLOOR:    6.00-
TRUNC:    5.00-
FRAC:     0.55-
```

The argument of these functions does not have to be a numeric data type. If you choose another type, it is converted to a numeric type. For performance reasons, however, you should use the correct type whenever possible. The functions itself do not have a data type of their own. They do not change the numerical precision of a numerical operation.

```
DATA: T1(10),
      T2(10) VALUE '-100'.
```

```
T1 = ABS( T2).
```

```
WRITE T1.
```

This produces the following output:

```
100
```

Two conversions are performed. First, the contents of field T2 (type C) are converted to type P. Then the system processes function ABS using the results of the conversion. Then, during the assignment to the type C field T1, the result of the function is converted back to type C.

Floating-Point Functions

Floating-Point Functions

The following built-in functions work with the floating point data type (F) as an argument.

Functions for floating point data types

Function	Explanation
ACOS, ASIN, ATAN; COS, SIN, TAN	Trigonometric functions.
COSH, SINH, TANH	Hyperbolic functions.
EXP	Exponential function with base e (e=2.7182818285).
LOG	Natural logarithm with base e.
LOG10	Logarithm with base 10.
SQRT	Square root.

For all functions, the normal mathematical constraints (e.g. square root is possible only for positive numbers) apply. Otherwise, a runtime error occurs.

The argument of these functions does not have to be a floating point data type. If you choose another type, it is converted to type F, as described in [Type Conversions \[Page 218\]](#). The functions itself have the data type F. They can change the numerical precision of a numerical operation.

```
DATA: RESULT TYPE F,  
      PI(10) VALUE '3.141592654'.
```

```
RESULT = COS( PI).
```

```
WRITE RESULT.
```

The output is -1.000000000000000E+00. The character field PI is automatically converted to a type F field before the calculation is performed.

Processing Packed Numbers

If the program attribute *Fixed point arithmetic* is not set, type P fields are interpreted as integers without decimal places. The DECIMALS parameter of the DATA statement only affects the format of the WRITE output.

For this reason, SAP recommends that you always set the program attribute *Fixed point arithmetic* when working with type P fields (see [Maintaining Program Attributes \[Page 74\]](#)).

Then, the keyword DECIMALS refers not only to the position of the decimal point when you output a type P field to the output list, but also takes into account the decimal places in arithmetic operations.

In the case of intermediate results, ABAP calculates up to 31 places (before and after the decimal point). If *Fixed point arithmetic* is set, calculations with packed numbers in ABAP work in the same way as in pocket calculators.

```
DATA: P TYPE P.
```

```
P = 1 / 3 * 3.
```

```
WRITE P.
```

If the program attribute *Fixed point arithmetic* is set, the result is 0 because the result of the division is rounded internally to 0.

If the program attribute *Fixed point arithmetic* is set, the result is 1 because the result of the division is stored internally as 0.33333333333333333333333333333333 with an accuracy of up to 31 digits.

Processing Date and Time Fields

Processing Date and Time Fields

The data type of date and time fields is not numeric. Nevertheless, you can process date and time fields similar to numeric fields because of the automatic type conversions which are performed (see [Convertibility of Elementary Data Types \[Page 219\]](#)).

When you process date and time fields, it can be useful to use offset (see [Specifying Offset Values for Data Objects \[Page 216\]](#)).

DATA: ULTIMO TYPE D.

ULTIMO = SY-DATUM.

ULTIMO+6(2) = '01'. " = first day of this month

ULTIMO = ULTIMO - 1. " = last day of last month

Here, the last day of the previous month is assigned to the date field ULTIMO.

1. ULTIMO is filled with the present date.
2. Using an offset specification, the day is changed to the first day of the current month.
3. 1 is subtracted from ULTIMO. Its contents are changed to the last day of the previous month. Before performing the subtraction, the system converts ULTIMO to the number of days since 01.01.0001 and converts the result back to a date.

DATA: DIFF TYPE I,
SECONDS TYPE I,
HOURS TYPE I.

DATA: T1 TYPE T VALUE '200000',
T2 TYPE T VALUE '020000'.

DIFF = T2 - T1.

SECONDS = DIFF MOD 86400.

HOURS = SECONDS / 3600.

The last three lines can be replaced by the following line

HOURS = ((T2 - T1) MOD 86400) / 3600.

The number of hours between 02:00:00 and 20:00:00 is calculated.

First, the difference between the time fields is calculated, which is -64800, since T1 is converted internally to 72000, and T2 to 7200.

Second, with the operation MOD, this negative difference is converted to the total number of seconds. Note that if the difference were positive, it would remain unchanged by the MOD operation.

Third, the number of hours is calculated by dividing the number of seconds by 3600.

In some cases (e.g. sorting dates in descending order), it is useful to convert a date from format D to an inverted date by using the keyword CONVERT.

Syntax

CONVERT DATE <d1> INTO INVERTED-DATE <d2>.

CONVERT INVERTED-DATE <d1> INTO DATE <d2>.

These statements convert the field <d1> from the formats DATE or INVERTED-DATE to the formats INVERTED-DATE or DATE and assign it to the field <d2>.

For the conversion, ABAP forms the nine's complement.

```
DATA: ODATE TYPE D VALUE '19955011',
      IDATE LIKE ODATE.
```

```
DATA FIELD(8).
```

```
FIELD = ODATE. WRITE / FIELD.
```

```
CONVERT DATE ODATE INTO INVERTED-DATE IDATE.
```

```
FIELD = IDATE. WRITE / FIELD.
```

```
CONVERT INVERTED-DATE IDATE INTO DATE ODATE.
```

```
FIELD = ODATE. WRITE / FIELD.
```

The output appears as follows:

```
19955011
```

```
80049488
```

```
19955011
```

Processing Character Strings

ABAP provides several keywords for processing data objects of type C, also known as character strings.

The system does not perform type conversions during operations on character strings. With each of the string processing keywords described in the following topics, the system treats all operands as type C fields, regardless of their actual type.

You can process character strings by

[Shifting Field Contents \[Page 200\]](#)

[Replacing Field Contents \[Page 204\]](#)

[Converting to Upper/Lower Case and Substituting Characters \[Page 206\]](#)

[Converting into a Sortable Format \[Page 207\]](#)

[Overlaying Strings \[Page 208\]](#)

[Finding Strings \[Page 209\]](#)

[Condensing Field Contents \[Page 212\]](#)

[Finding out the Length of a Character String \[Page 211\]](#)

[Concatenating Strings \[Page 213\]](#)

[Splitting Strings \[Page 214\]](#)

An explanation of a MOVE statement variant which works only with character fields, you find under

[Assigning Parts of Character Strings \[Page 215\]](#).

Shifting Field Contents

To shift the contents of a field, use the different variants of the SHIFT statement described in the following topics. Using SHIFT allows you to shift field contents byte by byte (or character by character in the case of text fields).

With the SHIFT statement, you can execute the following:

[Shifting a Structure by a Given Number of Positions \[Page 201\]](#)

[Shifting a Structure up to a Given String \[Page 202\]](#)

[Shifting a Structure According to the First or Last Character \[Page 203\]](#)

Shifting a Structure by a Given Number of Positions

Shifting a Structure by a Given Number of Positions

To shift field contents by a given number of positions, use the SHIFT statement as follows:

Syntax

SHIFT <c> [BY <n> PLACES] [<mode>].

This statement shifts the field <c> by <n> positions. If you omit BY <n> PLACES, <n> is interpreted as one. If <n> is 0 or negative, <c> remains unchanged. If <n> exceeds the length of <c>, <c> is padded with blanks. <n> can be variable.

With the different (<mode>) options, you can shift the field <c> in the following ways:

- **LEFT:**
Shift <n> positions to the left and pad with <n> blanks on the right (default setting).
- **RIGHT:**
Shift <n> positions to the right and pad with <n> spaces on the left.
- **CIRCULAR:**
Shift <n> positions to the left so that <n> characters on the left appear on the right.

```
DATA: T(10) VALUE 'abcdefghij',  
      STRING LIKE T.  
  
STRING = T.  
WRITE STRING.  
  
SHIFT STRING.  
WRITE / STRING.  
  
STRING = T.  
SHIFT STRING BY 3 PLACES LEFT.  
WRITE / STRING.  
  
STRING = T.  
SHIFT STRING BY 3 PLACES RIGHT.  
WRITE / STRING.  
  
STRING = T.  
SHIFT STRING BY 3 PLACES CIRCULAR.  
WRITE / STRING.
```

The output appears as follows:

```
abcdefghij  
bcdefghij  
defghij  
  abcdefg  
defghijabc
```

Shifting a Structure up to a Given String

To shift field contents up to a given string, use the SHIFT statement as follows

Syntax

SHIFT <c> UP TO <str> <mode>.

ABAP searches the field contents of <c> until it finds the string <str> and shifts the field <c> up to the field margin. The (<mode>) options are the same as described in the section [Shifting a Structure by a Given Number of Positions \[Page 201\]](#). <str> can be a variable.

If <str> is not found in <c>, SY-SUBRC is set to 4 and <c> is not shifted. Otherwise, SY-SUBRC is set to 0.

```
DATA: T(10) VALUE 'abcdefghij',
      STRING LIKE T,
      STR(2) VALUE 'ef'.

STRING = T.
WRITE STRING.

SHIFT STRING UP TO STR.
WRITE / STRING.

STRING = T.
SHIFT STRING UP TO STR LEFT.
WRITE / STRING.

STRING = T.
SHIFT STRING UP TO STR RIGHT.
WRITE / STRING.

STRING = T.
SHIFT STRING UP TO STR CIRCULAR.
WRITE / STRING.
```

The output appears as follows:

```
abcdefghij
efghij
efghij
  abcdef
efghijabcd
```

Shifting a Structure According to the First or Last Character

Shifting a Structure According to the First or Last Character

You can use the SHIFT statement to shift a field to the left or to the right, provided the first or last character satisfies a certain condition. To do this, use the following syntax:

Syntax

SHIFT <c> LEFT DELETING LEADING <str>.

SHIFT <c> RIGHT DELETING TRAILING <str>.

This statement shifts the field <c> to the left or to the right, provided the first character on the left or the last character on the right occur in <str>. The right or left of the field is then padded with blanks. <str> can be a variable.

```
DATA: T(14) VALUE '  abcdefghij',  
      STRING LIKE T,  
      STR(6) VALUE 'ghijkl'.
```

```
STRING = T.  
WRITE STRING.
```

```
SHIFT STRING LEFT DELETING LEADING SPACE.  
WRITE / STRING.
```

```
STRING = T.  
SHIFT STRING RIGHT DELETING TRAILING STR.  
WRITE / STRING.
```

The output appears as follows:

```
  abcdefghij  
abcdefghij  
      abcdef
```

Replacing Field Contents

To replace certain parts of structures with other strings, you use the REPLACE statement.

Syntax

REPLACE <str1> WITH <str2> INTO <c> [LENGTH <l>].

ABAP searches the field <c> for the first occurrence of the first <l> positions of the pattern <str1>. If no length is specified, it searches for the pattern <str1> in its full length.

Then, the statement replaces the first occurrence of the pattern <str1> in field <c> with the string <str2>. If a length <l> was specified, only the relevant part of the pattern is replaced.

If the return code value of the system field SY-SUBRC is set to 0, this indicates that <str1> was found in <c> and replaced by <str2>. A return code value other than 0 means that nothing was replaced.

<str1>, <str2>, and <len> can be variables.

```
DATA: T(10) VALUE 'abcdefghij',
      STRING LIKE T,
      STR1(4) VALUE 'cdef',
      STR2(4) VALUE 'klmn',
      STR3(2) VALUE 'kl',
      STR4(6) VALUE 'klmnop',
      LEN TYPE I VALUE 2.

STRING = T.
WRITE STRING.

REPLACE STR1 WITH STR2 INTO STRING.
WRITE / STRING.

STRING = T.
REPLACE STR1 WITH STR2 INTO STRING LENGTH LEN.
WRITE / STRING.

STRING = T.
REPLACE STR1 WITH STR3 INTO STRING.
WRITE / STRING.

STRING = T.
REPLACE STR1 WITH STR4 INTO STRING.
WRITE / STRING.
```

The output appears as follows:

```
abcdefghij
abklmng hij
abklmne fgh
abklghij
abklmnopgh
```

Replacing Field Contents

Note in the last line how the field STRING is truncated on the right. The search pattern 'cdef' of length 4 is replaced by 'klmnop' of length 6. Then, the rest of the field STRING is filled up to the end of the field.

Converting to Upper/Lower Case and Substituting Characters

You can convert the letters to upper/lower case or use substitution rules.

To converting to upper/lower case, use the TRANSLATE statement as follows:

Syntax

TRANSLATE <c> TO UPPER CASE.

TRANSLATE <c> TO LOWER CASE.

These statements convert all lower case letters in the field <c> to upper case or vice versa.

When using substitution rules, use the following syntax:

Syntax

TRANSLATE <c> USING <r>.

This statement replaces all characters in field <c> according to the substitution rule stored in field <r>. <r> contains pairs of letters, where the first letter of each pair is replaced by the second letter. <r> can be a variable.

For more variants of the TRANSLATE statement with more complex substitution rules, see the keyword documentation.

```
DATA: T(10) VALUE 'AbCdEfGhIj',  
      STRING LIKE T,  
      RULE(20) VALUE 'AxbXCydYEzfZ'.  
  
STRING = T.  
WRITE STRING.  
  
TRANSLATE STRING TO UPPER CASE.  
WRITE / STRING.  
  
STRING = T.  
TRANSLATE STRING TO LOWER CASE.  
WRITE / STRING.  
  
STRING = T.  
TRANSLATE STRING USING RULE.  
WRITE / STRING.
```

The output appears as follows:

```
AbCdEfGhIj  
ABCDEFGHIJ  
abcdefghij  
xXyYzZGhIj
```

Converting into a Sortable Format

Converting into a Sortable Format

You can convert character fields into an alphabetically sortable format as follows:

Syntax

CONVERT TEXT <c> INTO SORTABLE CODE <sc>.

This statement fills a sortable target field <sc> for a character field <c>. The field <c> must be of type C and the field <sc> must be of type X with a minimum size of 16 times the size of <c>.

The purpose of this statement is to create an accompanying field <sc> for a character field <c>, which can serve as an alphabetical sort key for <c>. You use sort functions with internal tables and extracts (see [Sorting Internal Tables \[Page 319\]](#) and [Sorting Extract Datasets \[Page 933\]](#)).

If you sort unconverted character fields, the system creates an order that corresponds to the platform-specific internal coding of the individual letters. The conversion CONVERT TEXT creates target fields in such a way that, after sorting the target fields, the order of the corresponding character fields is alphabetical. For example, in Germany the order is 'Miller, Moller, Möller, Muller' instead of 'Miller, Moller, Muller, Möller'.

The method of conversion depends on the text environment of the running ABAP program. The text environment is defined in the user's master record. As an exception, you can set the text environment in the program by using:

Syntax

SET LOCALE LANGUAGE <lg> [COUNTRY <cy>] [MODIFIER <m>].

This statement sets the text environment according to the language <lg>. With the option COUNTRY, you can specify the country additionally to the language, provided there are country-specific differences for languages. With the option MODIFIER, you can specify another identifier, provided there are differences in the language within one country, as for example, the sorting sequence differing between phone books and dictionaries.

The fields <lg>, <cy>, and <m> must be of type C and must have the same lengths as the key fields of table TCP0C. Table TCP0C is a table, in which the text environment is maintained platform-dependent. During the statement SET LOCALE, the system sets the text environment according to the entries in TCP0C. With the exception of the platform identity, which is transferred internally, the table key is specified with the SET statement. The platform identifier is passed implicitly. If <lg> equals SPACE, the system sets the text environment according to the user's master record. If there is no entry in the table for the key specified, the system reacts with a runtime error.

The text environment influences **all** operations in ABAP that depend on the character set.

For more information about this topic, see the keyword documentation of CONVERT TEXT and of SET LOCALE LANGUAGE.

For an example of alphabetical sorting, see [Sorting Internal Tables \[Page 319\]](#).

Overlaying Character Fields

To overlay a character field with another character field, use the OVERLAY statement as follows:

Syntax

OVERLAY <c1> WITH <c2> [ONLY <str>].

This statement overlays all positions in field <c1> containing letters which occur in <str> with the contents of <c2>. <c2> remains unchanged. If you omit ONLY <str>, all positions of <c1> containing spaces are overwritten.

If at least one character in <c1> was replaced, SY-SUBRC is set to 0. In all other cases, SY-SUBRC is set to 4. If <c1> is longer than <c2>, it is overlaid only in the length of <c2>.

```
DATA: T(10) VALUE 'a c e g i',  
      STRING LIKE T,  
      OVER(10) VALUE 'ABCDEFGHIJ',  
      STR(2) VALUE 'ai'.  
  
STRING = T.  
WRITE STRING.  
WRITE / OVER.  
  
OVERLAY STRING WITH OVER.  
WRITE / STRING.  
  
STRING = T.  
OVERLAY STRING WITH OVER ONLY STR.  
WRITE / STRING.
```

The output appears as follows:

```
a c e g i  
ABCDEFGHIJ  
aBcDeFgHiJ  
A c e g l
```


Searching for Character Strings

Searching for Character Strings

To search a character field for a particular pattern, use the SEARCH statement as follows:

Syntax

SEARCH <c> FOR <str> <options>.

This statement searches the field <c> for the character string in <str>. If successful, the return code value of SY-SUBRC is set to 0 and SY-FDPOS is set to the offset of the string in the field <c>. Otherwise, SY-SUBRC is set to 4.

The search string <str> can have one of the following forms.

<str>	Function
<pattern>	<pattern> (any sequence of characters) is sought. Trailing blanks are ignored.
.<pattern>.	Seatches for <pattern>. Trailing blanks are not ignored.
*<pattern>	A word ending with <pattern> is sought.
<pattern>*	A word starting with <pattern> is sought.

Words are separated by blanks, commas, periods, semicolons, colons, question marks, exclamation marks, parentheses, slashes, plus signs, and equal signs.

```
DATA STRING(30) VALUE 'This is a little sentence.'.
```

```
WRITE: / 'Searched', 'SY-SUBRC', 'SY-FDPOS'.
ULINE /1(26).
```

```
SEARCH STRING FOR 'X'.
WRITE: / 'X', SY-SUBRC UNDER 'SY-SUBRC',
        SY-FDPOS UNDER 'SY-FDPOS'
```

```
SEARCH STRING FOR 'itt '.
WRITE: / 'itt ', SY-SUBRC UNDER 'SY-SUBRC',
        SY-FDPOS UNDER 'SY-FDPOS'
```

```
SEARCH STRING FOR '.e.'.
WRITE: / '.e.', SY-SUBRC UNDER 'SY-SUBRC',
        SY-FDPOS UNDER 'SY-FDPOS'.
```

```
SEARCH STRING FOR '*e'.
WRITE: / '*e ', SY-SUBRC UNDER 'SY-SUBRC',
        SY-FDPOS UNDER 'SY-FDPOS'.
```

```
SEARCH STRING FOR 's*'.
WRITE: / 's* ', SY-SUBRC UNDER 'SY-SUBRC',
        SY-FDPOS UNDER 'SY-FDPOS'.
```

The output appears as follows:

SEARCHED SY-SUBRC SY-FDPOS

```
X      4      0
```

```
itt    0     11
```

.e.	0	15
*e	0	10
s*	0	17

The different options (<options>) for searching the character field <c> are

- ABBREVIATED

Field <c> is searched for a word containing the string in <str>. The characters can be separated by other characters. The first letter of the word and the string <str> must be the same.

- STARTING AT <n1>

Searches the field <c> for <str> starting at position <n1>. The result SY-FDPOS refers to the offset relative to <n1> and not to the start of the field.

- ENDING AT <n2>

Searches the field <c> for <str> up to position <n2>.

- AND MARK

If the search string is found, all the characters in the search string (and all the characters inbetween when using ABBREVIATED) are converted to upper case.

```
DATA: STRING(30) VALUE 'This is a fast first example.',
      POS TYPE I,
      OFF TYPE I.
```

```
WRITE / STRING.
```

```
SEARCH STRING FOR 'ft' ABBREVIATED.
```

```
WRITE: / 'SY-FDPOS:', SY-FDPOS.
```

```
POS = SY-FDPOS + 2.
```

```
SEARCH STRING FOR 'ft' ABBREVIATED STARTING AT POS AND MARK.
```

```
WRITE / STRING.
```

```
WRITE: / 'SY-FDPOS:', SY-FDPOS.
```

```
OFF = POS + SY-FDPOS - 1.
```

```
WRITE: / 'Off:', OFF.
```

The output appears as follows:

```
This is a fast first example.
```

```
SY-FDPOS:  10
```

```
This is a fast FIRST example.
```

```
SY-FDPOS:   4
```

```
Off:      15
```

Note that in order to find the second word containing 'ft' after finding the word 'fast', you have to add 2 to the offset SY-FDPOS and start the search at the position POS. Otherwise, the word 'fast' would be found again. To obtain the offset of 'first' in relation to the start of the field STRING, it is calculated from POS and SY-FDPOS.

Obtaining the Length of a Character String

Obtaining the Length of a Character String

To determine the length of a character string up to the last character other than SPACE, use the built-in function STRLEN as follows:

Syntax

[COMPUTE] <n> = STRLEN(<c>).

STRLEN processes any operand <c> as a character data type, regardless of its real type. **No** conversions are performed.

The keyword COMPUTE is optional. For more information about built-in functions, see [Using Mathematical Functions \[Page 192\]](#).

```
DATA: INT TYPE I,  
      WORD1(20) VALUE '12345'.  
      WORD2(20).  
      WORD3(20) VALUE ' 4  ' .  
  
INT = STRLEN( WORD1). WRITE INT.  
INT = STRLEN( WORD2). WRITE / INT.  
INT = STRLEN( WORD3). WRITE / INT.  
The results are 5, 0, and 4 respectively.
```

Condensing Field Contents

To delete superfluous blanks in character fields, use the CONDENSE statement as follows:

Syntax

CONDENSE <c> [NO-GAPS].

This statement removes any leading blanks in the field <c> and replaces other sequences of blanks by exactly one blank. The result is a left-justified sequence of words, each separated by one blank. If the addition NO-GAPS is specified, all blanks are removed.

```
DATA: STRING(25) VALUE 'one two three four',  
      LEN TYPE I.
```

```
LEN = STRLEN( STRING).  
WRITE: STRING, '!'.  
WRITE: / 'Length: ', LEN.
```

```
CONDENSE STRING.  
LEN = STRLEN( STRING).  
WRITE: STRING, '!'.  
WRITE: / 'Length: ', LEN.
```

```
CONDENSE STRING NO-GAPS.  
LEN = STRLEN( STRING).  
WRITE: STRING, '!'.  
WRITE: / 'Length: ', LEN.
```

The output appears as follows:

```
one two three four !  
Length:      25  
  
one two three four   !  
Length:      18  
  
onetwothreefour     !  
Length:      15
```

Note that the total length of the field STRING remains unchanged (exclamation mark!), but that the deleted blanks appear again on the right.

Concatenating Character Strings

Concatenating Character Strings

To concatenate separate character strings into one, use the CONCATENATE statement as follows:

Syntax

```
CONCATENATE <c1>... <cn> INTO <c> [SEPARATED BY <s>].
```

This statement concatenates the character fields <c1> to <cn> and assigns the result to <c>.

Trailing blanks are ignored during this operation.

The addition SEPARATED BY <s> allows you to specify a character field <s> which is placed in its defined length between the individual fields.

If the result fits into <c>, SY-SUBRC is set to 0. However, if the result has to be truncated, SY-SUBRC is set to 4.

```
DATA: C1(10) VALUE 'Sum',  
      C2(3)  VALUE 'mer',  
      C3(5)  VALUE 'holi',  
      C4(10) VALUE 'day',  
      C5(30),  
      SEP(3) VALUE ' - '.
```

```
CONCATENATE C1 C2 C3 C4 INTO C5.  
WRITE C5.
```

```
CONCATENATE C1 C2 C3 C4 INTO C5 SEPARATED BY SEP.  
WRITE / C5.
```

The output appears as follows:

Summerholiday

Sum - mer - holi - day

In C1 to C5, the trailing blanks are ignored. The separator SEP retains them.

Splitting Character Strings

To split a character string into two or more smaller strings, use the SPLIT statement as follows:

Syntax

SPLIT <c> AT INTO <c1>... <cn>.

The system searches the field <c> after the separator . The parts before and after the separator are placed in the target fields <c1>... <cn>.

To place all fragments in different target fields, you must specify enough target fields. Otherwise, the last target field is filled with the rest of the field <c> and still contains delimiters.

If all target fields are long enough and no fragment has to be truncated, SY-SUBRC is set to 0. Otherwise it is set to 4.

```
DATA: STRING(60),
      P1(20) VALUE '+++++',
      P2(20) VALUE '+++++',
      P3(20) VALUE '+++++',
      P4(20) VALUE '+++++',
      DEL(3) VALUE '***'.

STRING = ' Part 1 *** Part 2 *** Part 3 *** Part 4 *** Part 5'.
WRITE STRING.

SPLIT STRING AT DEL INTO P1 P2 P3 P4.

WRITE / P1.
WRITE / P2.
WRITE / P3.
WRITE / P4.
```

The output appears as follows:

```
Part 1 *** Part 2 *** Part 3 *** Part 4 *** Part 5
Part 1
Part 2
Part 3
Part 4 *** Part 5
```

Note that the contents of the fields P1...P4 are totally overwritten and that they are padded with trailing blanks.

You can also place the parts which make up the original string in an internal table.

Syntax

SPLIT <c> AT INTO TABLE <itab>.

For each part of the character string, the system adds a new table line (for further information about internal tables, see [Creating and Processing Internal Tables \[Page 260\]](#)).

Assigning Parts of Character Strings

Assigning Parts of Character Strings

The following variant of the MOVE statement works only with type C fields:

Syntax

MOVE <c1> TO <c2> PERCENTAGE <p> [RIGHT].

Copies the percentage <p> percent of the character field <c1> left-justified (or right-justified if specified with the RIGHT option) to <c2>.

The value of <p> can be a number between 0 and 100. The length to be copied from <c1> is rounded up or down to the next whole number.

If one of the arguments in the statement is not type C, the parameter PERCENTAGE is ignored.

```
DATA C1(10) VALUE 'ABCDEFGHIJ',  
      C2(10).  
MOVE C1 TO C2 PERCENTAGE 40.  
WRITE C2.  
MOVE C1 TO C2 PERCENTAGE 40 RIGHT.  
WRITE / C2.
```

The output appears as follows:

```
'ABCD  '  
      ABCD
```

Specifying Offset Values for Data Objects

In ABAP, you can specify offset values for elementary data objects in all statements which process these data objects. To do so, specify the name of a data object in a statement as follows:

Syntax

`<f>[+<o>][(<l>)]`

The operation of the statement is performed for the part of the field `<f>` that begins at position `<o>+1` and has a length of `<l>`.

If the length `<l>` is not specified, the field is processed for all positions between `<o>` and the end of the field.

In general, offset `<o>` and length `<l>` must be specified as unsigned number literals. Dynamic specification using variables is possible in the following exceptional cases:

- assigning values using MOVE or the assignment operator (see [Assigning Values with Offset Specification \[Page 176\]](#)).
- assigning values using WRITE TO (see [Writing Values with Offset Specification \[Page 183\]](#)).
- Assigning fields to fields symbols with ASSIGN (see [Working with Field Symbols \[Page 336\]](#)).
- Passing data to subroutines in a PERFORM statement (see [Passing Data Using Parameters \[Page 454\]](#)).

Offset specifications make sense with character fields, numeric text fields, hexadecimal fields, and date and time fields.

Do not use offset specifications with the numeric fields of types F, I, and P, or with structures. They cannot be used with literals or text symbols (for further information about text symbols, see [Text Symbols \[Page 157\]](#)).

```
DATA TIME TYPE T VALUE '172545'.
```

```
WRITE TIME.
```

```
WRITE / TIME+2(2).
```

```
CLEAR TIME+2(4).
```

```
WRITE / TIME.
```

The output appears as follows:

```
172545
```

```
25
```

```
170000
```

Specifying Offset Values for Data Objects

First, the minutes are selected by specifying an offset in the WRITE statement. Then, the minutes and seconds are set to their initial values by specifying an offset in the clear statement.

Type Conversions

You can assign the contents of a data object of one data type to the data object of another data type. In this case, the data types involved must be convertible.

As a rule, all compatible data types are convertible.
(For further information about compatible data types, see [Compatibility of Data Types \[Page 111\]](#)).

With compatible data types, you do not even need to convert the value of the source field to the data type of the target field because all the technical properties are the same.

In ABAP, two non-compatible data types can be converted to each other if a conversion rule is defined for the purpose.

You can use the MOVE statement or the assignment operator (=) to assign a value of a data object of one data type to the data object of another data type if, and only if, the two data types are convertible. The value of the source object is converted to the data type of the target object. The conversion follows the rules that are defined in the system and described in the following topics.

With all ABAP operations that perform value assignments between data objects (e.g. arithmetic operations or filling internal tables), the system handles all the necessary type conversions as for the MOVE statement. If the data types of two fields are not convertible and you try to assign the contents from one field to the other, the system reports an error during the syntax check or even reacts with a runtime error.

The following topics describe the conversion rules defined in ABAP between non-compatible data types.

[Convertibility of Elementary Data Types \[Page 219\]](#)

[Convertibility of Structures \[Page 227\]](#)

[Convertibility of Internal Tables \[Page 232\]](#)

With some ABAP statements which pass data between different objects, the alignment of data objects also plays a role. For information about the alignment of data objects, see

[Alignment of Data Objects \[Page 233\]](#)

Convertibility of Elementary Data Types

Convertibility of Elementary Data Types

The eight elementary data types predefined in ABAP are listed in the table in the [Predefined Elementary Data Types \[Page 106\]](#) section.

There are 64 possible type combinations between these elementary data types. ABAP supports automatic type conversion and length adjustment for all of them except type D (date) and type T (time) fields which cannot be converted into each other.

The following conversion tables define the rules for converting elementary data types for all possible combinations of source and target fields.

[Source Type Character \[Page 220\]](#)

[Source Type Date \[Page 221\]](#)

[Source Type Floating Point Number \[Page 222\]](#)

[Source Type Numeric Text \[Page 223\]](#)

[Source Type Packed Number \[Page 224\]](#)

[Source Type Time \[Page 225\]](#)

[Source Type Hexadecimal \[Page 226\]](#)

Type I is always treated in the same way as type P without decimal places. Wherever type P is mentioned, the same applies to type I fields. If the attribute *Fixed point arithmetic* (see [Maintaining Program Attributes \[Page 74\]](#)) is set for a program, the system rounds type P fields according to the number of decimal places or pads them with zeros.

Source Type Character

Conversion table for source type C

Target	Conversion
C	The data in the target field is left-justified. If the field is too long, it is filled with blanks from the right. If it is too short, the contents are truncated on the right.
D	The character field should contain an 8-character date in YYYYMMDD format.
F	The contents of the source field must be a valid representation of a type F field as described in Literals [Page 113] .
N	Only the digits in the source field are copied. They are compressed on the right and padded with zeros on the left.
P	The source field must contain the representation of a decimal number, i.e. a sequence of digits with an optional sign and no more than one decimal point. The source field can contain blanks. If the target field is too short, an overflow may occur. This may cause the system to terminate the program.
T	The character field should contain a 6-character time in HHMMSS format.
X	Since the character field should contain a hexadecimal-character string, the only valid characters are 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. This character string is packed as a hexadecimal number, transported left-justified, and padded with zeros or truncated on the right.

Source Type Date

Source Type Date

Conversion table for source type D

Target	Conversion
C	The date is transported left-justified without conversion.
D	Transport without conversion.
F	The date is converted into a packed number. The packed number is then converted into a floating point number (see corresponding table).
N	The date is converted into a character field. The character field is then converted into a numeric text field (see corresponding table).
P	The date is converted to the number of days since 01.01.0001.
T	Not supported. Results in an error message during the syntax check or in a runtime error.
X	The date is converted to the number of days since 01.01.0001 in hexadecimal format.

Source Type Floating Point Number

Conversion table for source type F

Target	Conversion
C	The floating point number is converted to the <mantissa>E<exponent> format and transported to the character field. The value of the mantissa lies between 1 and 10 unless the number is zero. The exponent is always signed. If the target field is too short, the mantissa is rounded. The length of the character field should be at least 6 bytes.
D	The source field is converted into a packed number. The packed number is then converted into a date field (see corresponding table).
F	Transport without conversion.
N	The source field is converted into a packed number. The packed number is then converted into a numeric text field (see corresponding table).
P	The floating point number is converted to an integer or fixed point value and, if necessary, rounded.
T	The source field is converted into a packed number. The packed number is then converted into a time field (see corresponding table).
X	The source field is converted into a packed number. The packed number is then converted into a hexadecimal number (see corresponding table).

Source Type Numeric Text

Source Type Numeric Text

Conversion table for source type N

Target	Conversion
C	The numeric field is treated like a character field. Leading zeros are retained.
D	The numeric field is converted into a character field. The character field is then converted into a date field (see corresponding table).
F	The numeric field is converted into a packed number. The packed number is then converted into a floating point number (see corresponding table).
N	The numeric field is transported right-justified and padded with zeros or truncated on the left.
P	The numeric field is packed and transported right-justified with a positive sign. If the target field is too short, the program may be terminated.
T	The numeric field is converted into a character field. The character field is then converted into a time field (see corresponding table).
X	The numeric field is converted into a packed number. The packed number is then converted into a hexadecimal number (see corresponding table).

Source Type Packed Number

Conversion table for source type P

Target	Conversion
C	The packed field is transported right-justified to the character field, if required with a decimal point. The first position is reserved for the sign. Leading zeros appear as blanks. If the target field is too short, the sign is omitted for positive numbers. If this is still not sufficient, the field is truncated on the left. ABAP indicates the truncation with an asterisk (*). If you want the leading zeros to appear in the character field, use UNPACK instead of MOVE.
D	The packed field value represents the number of days since 01.01.0001 and is converted to a date in YYYYMMDD format.
F	The packed field is accepted and transported as a floating point number.
N	The packed field is rounded if necessary, unpacked, and then transported right-justified. The sign is omitted. If required, the target field is padded with zeros on the left.
P	The packed field is transported right-justified. If the target field is too short, an overflow occurs.
T	The packed field value represents the number of seconds since midnight and is converted to a time in HHMMSS format.
X	The packed field is rounded if necessary and then converted to a hexadecimal number. Negative numbers are represented by the two's complement. If the target field is too short, the number is truncated on the left.

Type I is always treated in the same way as type P without decimal places.

Source Type Time

Source Type Time

Conversion table for source type T

Target	Conversion
C	The source field is transported left-justified without conversion.
D	Not supported. Results in an error message during the syntax check or in a runtime error.
F	The source field is converted into a packed number. The packed number is then converted into a floating point number (see corresponding table).
N	The date is converted into a character field. The character field is then converted into a numeric text field (see corresponding table).
P	The date is converted to the number of seconds since midnight.
T	Transport without conversion.
X	The date is converted to the number of seconds since midnight in hexadecimal format.

Source Type Hexadecimal

Conversion table for source type X

Target	Conversion
C	The value in the hexadecimal field is converted to a hexadecimal character string, transported left-justified to the target field, and padded with zeros.
D	The source field value represents the number of days since 01.01.0001 and is converted to a date in YYYYMMDD format.
F	The source field is converted into a packed number. The packed number is then converted into a floating point number (see corresponding table).
N	The source field is converted into a packed number. The packed number is then converted into a numeric text field (see corresponding table).
P	The value of the source field is interpreted as a hexadecimal number. It is converted to a packed decimal number and transported right-justified to the target field. If the hexadecimal field is longer than 4 bytes, only the last four bytes are converted. If it is too short, a runtime error may occur.
T	The source field value represents the number of seconds since midnight and is converted to a time in HHMMSS format.
X	The value is transported left-justified and filled with X'00' on the right, if necessary.

Convertibility of Structures

With structures, a distinction is made between the following:

[Compatible Structures \[Page 228\]](#)

[Incompatible Structures and Elementary Fields \[Page 229\]](#)

[Structures with Internal Tables as Components \[Page 231\]](#)

Compatible Structures

According to the general rule, compatible structures are always convertible.

A MOVE statement transports compatible structures component by component.

Incompatible Structures and Elementary Fields

Incompatible Structures and Elementary Fields

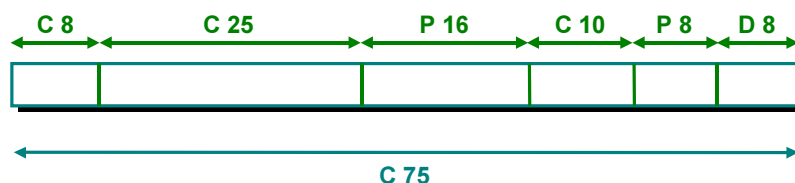
ABAP contains a rule for converting

- structures to non-compatible structures
- elementary fields to structures
- structures to elementary fields

In each of these cases, the system first converts all the structures concerned to type C fields and then performs the conversion between the two remaining elementary fields. The length of the type C fields is the sum of the lengths of the structure components.

If a structure is aligned (see [Aligning Data Objects \[Page 233\]](#)), the filler fields are also added to the length of the type C field.

The following structure is not aligned:



You should keep this rule in mind for all operations with structures. However, it applies only to structures which contain **no internal tables** as components.

If you convert a longer structure into a shorter one, the superfluous parts are omitted. If you convert a shorter structure into a longer one, the parts at the end are not initialized according to their type, but padded with blanks.

Conversions between non-compatible structures can make sense if, for example, the structure of a shorter structure corresponds to the structure of the beginning of a longer one.

```
DATA: BEGIN OF FS1,
      INT    TYPE I      VALUE 5,
      PACK   TYPE P DECIMALS 2 VALUE '2.26',
      TEXT(10) TYPE C     VALUE 'Fine text',
      FLOAT  TYPE F      VALUE '1.234e+05',
      DATA  TYPE D      VALUE '19950916',
END OF FS1.

DATA: BEGIN OF FS2,
      INT    TYPE I      VALUE 3,
      PACK   TYPE P DECIMALS 2 VALUE '72.34',
      TEXT(5) TYPE C     VALUE 'Hello',
END OF FS2.

WRITE: / FS1-INT, FS1-PACK; FS1-TEXT, FS1-FLOAT, FS1-DATE.
WRITE: / FS2-INT, FS2-PACK, FS2-TEXT.
```

Incompatible Structures and Elementary Fields

```
MOVE FS1 TO FS2.  
WRITE: / FS2-INT, FS2-PACK, FS2-TEXT.
```

The output appears as follows:

```
      5      2.26 Fine text  1.23400000000000E+05 09161995  
      3      72.34 Hello  
      5      2.26 Fine
```

This example defines two different structures, FS1 and FS2. The structure of the structures is the same for the first two components. After assigning FS1 to FS2, only the result for the first two components is as if they had been moved component-by-component. FS2-TEXT is filled with the first five characters of FS1-TEXT. All other positions of FS1 are omitted.

Numeric components of structures filled by assigning values of non-compatible structures can contain meaningless or even illegal values which result in a runtime error.

Structures with Internal Tables as Components

Structures with Internal Tables as Components

Structures which contain internal tables as components are convertible only if they are compatible.

Convertibility of Internal Tables

Internal tables are only convertible to other internal tables. Internal tables are not convertible to structures or elementary fields. Internal tables are convertible if their line types are convertible.

This can have the following consequences:

- Internal tables which have internal tables as line types are convertible if the internal tables which define the line types are convertible.
- Internal tables which have line types that are structures with internal tables as components are convertible only if the structures are **compatible** (see [Structures with Internal Tables as Components \[Page 231\]](#)).

For further information about internal tables, refer to [Creating and Processing Internal Tables \[Page 260\]](#).

Alignment of Data Objects

If fields are of type I or F, they are **aligned**. If structures contain type I or type F components, they are also aligned and filler fields may be inserted in front of them. The alignment of such fields or structures occurs because type I and type F fields occupy special platform-specific addresses in memory. For example, the address of a type I field must be divisible by 4 and the address of a type F field must be divisible by 8.

The system normally aligns fields and structures automatically during their declaration.

You only have to pay attention to the alignment in the following cases:

- if you pass fields or structures as parameters to subroutines which require another type for that parameter (see [Passing Data Using Parameters \[Page 454\]](#))
- if you change the type of a field symbol (see [Determining the Data Type of a Field Symbol \[Page 356\]](#)) or define a structured field symbol (see [Defining Structured Field Symbols \[Page 343\]](#)).
- if, when using Open SQL statements, you have work areas which do not have the same type as the table work area (see [Database Tables and SQL Concepts \[Page 539\]](#)).

Controlling the Flow of an ABAP Program

To execute different program components depending on certain conditions, or to combine repetitious statement sequences together in loops, you can use the standard keywords which ABAP provides to control the flow of a program.

Note that these keywords can only be used within processing blocks, and do not allow you to branch beyond the limits of a processing block.

For this reason, we distinguish in this section between the **internal control** of an ABAP program and the **external control** by events (see [Controlling the Flow of ABAP Programs Using Events \[Page 1208\]](#)).

To control the internal flow of your ABAP programs, follow the principles of structured programming and sub-divide your processing blocks into individual blocks of logically associated statements (control structure). Each of these blocks of statements is responsible for one of the tasks of your program.

To make your program easier to read, you should indent the statement blocks. Use the *Edit → Pattern* and *Program → Pretty printer* functions in the ABAP Editor (for further information about layout utilities, see [ABAP Program Layout \[Page 95\]](#)).

The following sections describe how to control program flow between blocks of statements using keywords such as IF, CASE, DO and WHILE. These keywords allow you to branch to other blocks and program loops, either conditionally or unconditionally. Conditions use logical expressions which may be either true or false.

[Programming Logical Expressions \[Page 235\]](#)

[Programming Branches and Loops \[Page 247\]](#)

Programming Logical Expressions

You control the internal flow of your program with conditions. To formulate conditions, use logical expressions which compare data fields as follows:

Syntax

.... <f1> <operator> <f2>...

This expression compares two fields. It can be either true or false. You use logical expressions in condition statements with the keywords IF, CHECK, and WHILE.

Depending on the data types of the operands <f1> and <f2>, you can use different logical operators. These allow you to perform

[Comparisons with All Field Types \[Page 236\]](#)

[Comparisons with Character Strings and Numeric Strings \[Page 238\]](#)

[Comparisons of Bit Structures \[Page 241\]](#)

Besides the above comparisons, you can perform tests to check whether data fields fulfill certain criteria. You can use these tests for

[Checking Whether a Field Belongs to a Range \[Page 243\]](#)

[Checking for the Initial Value \[Page 244\]](#)

[Checking Selection Criteria \[Page 245\]](#)

In addition, you can combine several logical expressions into one single logical expression.

[Combining Several Logical Expressions \[Page 246\]](#)

Comparisons with All Field Types

For comparisons with all field types, you can use the following operators in logical expressions:

<operator>	Meaning
EQ	equal to
=	equal to
NE	not equal to
<>	not equal to
><	not equal to
LT	less than
<	less than
LE	less than or equal to
<=	less than or equal to
GT	greater than
>	greater than
GE	greater than or equal to
>=	greater than or equal to

The operands can be database fields, internal fields, literals, or constants.

Besides elementary fields, you can use structured data types as operands together with operators from the above table. Field strings are compared component by component and nested structures are broken down into elementary fields. For further information about comparing internal tables, see [Comparing Internal Tables \[Page 332\]](#).

If it makes sense, you can compare fields of different data types. Fields can be compared if they are convertible. Before performing the comparison, the system executes an automatic [type conversion \[Page 218\]](#) according to the following hierarchical rules:

1. If one of the operands is a floating point number (type F), the system also converts the other operands to type F.
2. If one of the operands is a packed field (type P), the system also converts the other operands to type P.
3. If one of the operands is a date field (type D) or a time field (type T), the system converts the other operands to type D or T. Comparisons between date and time fields are not supported and result in termination of the program.
4. If one of the operands is a character field (type C) and the other operand is a hexadecimal field (type X), the system converts the operand of type X to type C.
5. If one of the operands is a character field (type C) and the other a numeric field (type N), the system converts both operands to type P.

Comparisons with All Field Types

```
DATA: F TYPE F VALUE '100.00',  
      P TYPE P VALUE '50.00' DECIMALS 2,  
      I TYPE I VALUE '30.00'.
```

WRITE 'The following logical expressions are true:'.

```
IF F >= P.  
  WRITE: / F, '>=', P.  
ELSE.  
  WRITE: / F, '<', P.  
ENDIF.
```

```
IF I EQ P.  
  WRITE: / I, 'EQ', P.  
ELSE.  
  WRITE: / I, 'NE', P.  
ENDIF.
```

The output appears as follows:

The following logical expressions are true:

```
1.0000000000000000E+02 >=      50.00  
30 NE      50.00
```

Here, two logical expressions are used in IF statements. If a logical expression is true, it is displayed on the screen. If a logical expression is false, the inverse expression appears on the screen.

Comparisons with Character Strings and Numeric Strings

For comparisons with character strings (type C) and numeric text (type N), you can use the following operators in logical expressions.

<operator>	Meaning
CO	Contains Only
CN	Contains Not only
CA	Contains Any
NA	contains Not Any
CS	Contains String
NS	contains No String
CP	Contains Pattern
NP	contains No Pattern

With comparisons containing one of these operations, the operands should be of type N or C because the system does not perform any other type conversion apart from type N to type C.

The function of the operators is as follows:

CO (Contains Only)

The logical expression

`<f1> CO <f2>`

is true if `<f1>` contains only characters from `<f2>`. The comparison is case-sensitive. Trailing blanks are included. If the comparison is true, the system field SY-FDPOS contains the length of `<f1>`. If it is false, SY-FDPOS contains the offset of the first character of `<f1>` that does not occur in `<f2>`.

CN (Contains Not only)

The logical expression

`<f1> CN <f2>`

is true if `<f1>` does also contains characters other than those in `<f2>`. The comparison is case-sensitive. Trailing blanks are included. If the comparison is true, the system field SY-FDPOS contains the offset of the first character of `<f1>` that does not also occur in `<f2>`. If it is false, SY-FDPOS contains the length of `<f1>`.

CA (Contains Any)

The logical expression

`<f1> CA <f2>`

Comparisons with Character Strings and Numeric Strings

is true if <f1> contains at least one character from <f2>. The comparison is case-sensitive. If the comparison is true, the system field SY-FDPOS contains the offset of the first character of <f1> that also occurs in <f2>. If it is false, SY-FDPOS contains the length of <f1>.

NA (contains Not Any)

The logical expression

<f1> NA <f2>

is true if <f1> does not contain any character from <f2>. The comparison is case-sensitive. If the comparison is true, the system field SY-FDPOS contains the length of <f1>. If it is false, SY-FDPOS contains the offset of the first character of <f1> that occurs in <f2>.

CS (Contains String)

The logical expression

<f1> CS <f2>

is true if <f1> contains the character string <f2>. Trailing spaces are ignored and the comparison is **not** case-sensitive. If the comparison is true, the system field SY-FDPOS contains the offset of <f2> in <f1>. If it is false, SY-FDPOS contains the length of <f1>.

NS (contains No String)

The logical expression

<f1> NS <f2>

is true if <f1> does not contain the character string <f2>. Trailing spaces are ignored and the comparison is **not** case-sensitive. If the comparison is true, the system field SY-FDPOS contains the length of <f1>. If it is false, SY-FDPOS contains the offset of <f2> in <f1>.

CP (Contains Pattern)

The logical expression

<f1> CP <f2>

is true if <f1> contains the pattern <f2>. If <f2> is of type C, you can use the following wildcards in <f2>:

- * for any character string
- + for any single character

Trailing spaces are ignored and the comparison is **not** case-sensitive. If the comparison is true, the system field SY-FDPOS contains the offset of <f2> in <f1>. If it is false, SY-FDPOS contains the length of <f1>.

If you want to perform a comparison on a particular character in <f2>, place the escape character # in front of it. You can use the escape character # to specify

- characters in upper and lower case
- the wildcard character "*" (enter #*) #*)
- the wildcard character "+" (enter #+) #+)
- the escape symbol itself (enter ##) ##)

Comparisons with Character Strings and Numeric Strings

- blanks at the end of a string (enter #____) #____)

NP (contains No Pattern)

The logical expression

<f1> NP <f2>

is true if <f1> does not contain the pattern <f2>. In <f2>, you can use the same wildcards and escape character as for the operator CP.

Trailing spaces are ignored and the comparison is **not** case-sensitive. If the comparison is true, the system field SY-FDPOS contains the length of <f1>. If it is false, SY-FDPOS contains the offset of <f2> in <f1>.

```
DATA: F1(5) TYPE C VALUE <f1>,
```

```
      F2(5) TYPE C VALUE <f2>.
```

```
IF F1 <operator> F2.
```

```
  WRITE: / 'Comparison true, SY-FDPOS=', SY-FDPOS.
```

```
ELSE.
```

```
  WRITE: / 'Comparison false, SY-FDPOS=', SY-FDPOS.
```

```
ENDIF.
```

The following table shows the results of executing this program, depending on which operators and F1 / F2 fields you use.

<f1>	<operator>	<f2>	Result	SY-FDPOS
'BD '	CO	'ABCD '	true	5
'BD '	CO	'ABCDE'	false	2
'ABC12'	CN	'ABCD '	true	3
'ABABC'	CN	'ABCD '	false	5
'ABcde'	CA	'Bd '	true	1
'ABcde'	CA	'bD '	false	5
'ABAB '	NA	'AB '	false	0
'ababa'	NA	'AB '	true	5
'ABcde'	CS	'bC '	true	1
'ABcde'	CS	'ce '	false	5
'ABcde'	NS	'bC '	false	1
'ABcde'	NS	'ce '	true	5
'ABcde'	CP	'*b*'	true	1
'ABcde'	CP	'*#b*'	false	5
'ABcde'	NP	'*b*'	false	1
'ABcde'	NP	'*#b*'	true	5

Comparisons of Bit Structures

Comparisons of Bit Structures

To compare the bit structure of the first byte of the first operand of a logical expression with the bit structure of the second operand, use the following operators.

<operator>	Meaning
O	bits are one
Z	bits are zero
M	bits are mixed

The second operand should have a length of one byte.

It is not necessary, but advantageous, to use a hexadecimal field (type X) of length one for the second operand because it has a length of one byte and its numeric value is directly related to its bit structure.

The function of the operators is as follows:

O (bits are one)

The logical expression

<f> O <hex>

is true if the bit positions that are 1 in <hex>, are 1 in <f>.

Z (bits are zero)

The logical expression

<f> Z <hex>

is true if the bit positions that are 1 in <hex>, are 0 in <f>.

M (bits are mixed)

The logical expression

<f> M <hex>

is true if from the bit positions that are 1 in <hex>, at least one is 1 and one is 0 in <f>.

```
DATA: C VALUE 'C',
      HEXDEC TYPE X,
      I TYPE I.
HEXDEC = 0.
DO 256 TIMES.
  I = HEXDEC.
  IF C O HEXDEC.
    WRITE: / HEXDEC, I.
  ENDIF.
  HEXDEC = HEXDEC + 1.
ENDDO.
```

The output appears as follows:

```
00      0
01      1
02      2
03      3
40     64
41     65
42     66
43     67
```

Here, the bit structure of the character 'C' is compared to all hexadecimal numbers HEXDEC between '0' and 'FF' (255 in the decimal system), using the operator O. The decimal value of HEXDEC, I is determined by using the automatic type conversion during the assignment of HEXDEC to I. If the comparison is true, the hexadecimal number and its decimal value are displayed on the screen. The following table shows the bit structure of these numbers.

hexadecimal	decimal	bit structure
00	0	00000000
01	1	00000001
02	2	00000010
03	3	00000011
40	64	01000000
41	65	01000001
42	66	01000010
43	67	01000011

The bit structure of the character 'C' is defined for the current hardware platform by its ASCII code number 67. As you see, the numbers which occur in the list output are those where the bit positions filled with 1 are the same as in the bit structure of 67.

Checking Whether a Field Belongs to a Range

Checking Whether a Field Belongs to a Range

To check whether a value belongs to a particular range, you use a logical expression with the BETWEEN parameter as follows:

Syntax

.... <f1> BETWEEN <f2> AND <f3>.....

This expression is true if <f1> occurs in the range between <f2> and <f3>. It is a short form of the following logical expression:

IF <f1> GE <f2> AND <f1> LE <f3>.

```
DATA: NUMBER TYPE I,  
      FLAG.  
  
....  
NUMBER =...  
  
....  
IF NUMBER BETWEEN 3 AND 7.  
  FLAG = 'X'.  
ELSE.  
  FLAG = ''.  
ENDIF.
```

Here, if the contents of NUMBER occur between 3 and 7, the field FLAG is set to "X".

Checking for the Initial Value

To check whether a field is set to its initial value, you use a logical expression with the IS INITIAL parameter as follows.

Syntax

.... <f> IS INITIAL.....

This expression is true if the field <f> contains the initial value for its type. In general, any field, elementary or structured (field strings and internal tables), contains its initial value after a CLEAR <f> statement is executed (see [Resetting Values to Initial Values \[Page 184\]](#)).

```
DATA FLAG VALUE 'X'.
IF FLAG IS INITIAL.
  WRITE / 'Flag is initial'.
ELSE.
  WRITE / 'Flag is not initial'.
ENDIF.

CLEAR FLAG.

IF FLAG IS INITIAL.
  WRITE / 'Flag is initial'.
ELSE.
  WRITE / 'Flag is not initial'.
ENDIF.
```

The output appears as follows:

Flag is not initial

Flag is initial.

Here, the character string FLAG does not contain its initial value after the DATA statement because it is set to the start value 'X' with the VALUE parameter. When the CLEAR statement is executed, it is reset to its initial value.

Checking Selection Criteria

To check whether the contents of a field match the selection criteria in a selection table, you use a logical expression with the IN parameter as follows:

Syntax

... <f> IN <seltab>....

This expression is true if the contents of the field <f> match the conditions in the selection table <seltab>.

For more information about creating selection criteria, see [SELECT-OPTIONS - Defining Selection Criteria \[Page 815\]](#).

For further information about checking selection conditions in logical expressions, including an example, see [Using Selection Tables in Logical Expressions \[Page 859\]](#).

Combining Several Logical Expressions

You can combine several logical expressions together in one single expression by using the logical link operators AND, OR, and NOT:

- To combine several logical expressions together in one single expression which is true only if all of the component expressions are true, link the expressions with AND.
- To combine several logical expressions together in one single expression which is true if at least one of the component expressions is true, link the expressions with OR.
- To invert the result of a logical expression, specify NOT in front of it.

NOT takes priority over AND, and AND takes priority over OR. However, you can use any combination of parentheses to specify the processing sequence. ABAP treats each parenthesis as though it were a word. You must therefore leave at least one space before and after each one.

ABAP processes logical expressions from left to right. If it recognizes one of the component expressions as true or false, it does not perform the remaining comparisons or checks in this component. This means that you can improve performance by organizing logical expressions in such a way that you place comparisons which are often false at the beginning of an AND chain and expensive comparisons, such as searches for character strings, at the end.

```
DATA: F TYPE F VALUE '100.00',
      N(3) TYPE N VALUE '123',
      C(3) TYPE C VALUE '456'.
```

```
WRITE 'The following logical expression is true:'.
```

```
IF ( C LT N ) AND ( N GT F ).
  WRITE: / ('C','lt',N,) AND ('N','gt',F,').
ELSE.
  WRITE: / ('C','ge',N,) OR ('N','le',F,').
ENDIF.
```

The output appears as follows:

```
The following logical expression is true:
```

```
( 456 ge 123 ) OR ( 123 le 1.0000000000000000E+02 )
```

In this example, a logical expressions is used in an IF statement. If the logical expression is true, it is displayed on the screen. If it is false, the inverted expression appears on the screen.

Programming Branches and Loops

You can define conditional and unconditional branches and loops in your program. ABAP has a series of statements with which you can set up branches and loops. These are described in the following sections:

Branchings

[Conditional Branching using IF \[Page 248\]](#)

[Conditional Branching using CASE \[Page 249\]](#)

Loops

[Unconditional Looping using DO \[Page 251\]](#)

[Conditional Loops using WHILE \[Page 254\]](#)

[Terminating Loops \[Page 256\]](#)

Conditional Branching using IF

The IF statement allows you to divert the program flow to a particular statement block, depending on a condition. This statement block consists of all the commands which occur between an IF statement and the next ELSEIF, ELSE, or ENDIF statement.

Syntax

```
IF <condition1>.  
    <statement block>  
ELSEIF <condition2>.  
    <statement block>  
ELSEIF <condition3>.  
    <statement block>  
.....  
ELSE.  
    <statement block>  
ENDIF.
```

If the first condition is true, the system executes all the statements up to the end of the first statement block and then continues processing after the ENDIF statement. If the first condition is false, the system processes the following ELSEIF statement in the same way as the IF statement. ELSE begins a statement block which is processed if none of the IF or ELSEIF conditions is true. The end of the last statement block must always be concluded by ENDIF.

To formulate conditions in IF or ELSEIF statements, you can use any logical expression described in [Programming Logical Expressions \[Page 235\]](#).

ABAP allows you to nest IF... ENDIF blocks to any depth. However, they must start and finish within the same processing block. In other words, an IF - ENDIF block cannot contain an event keyword.

```
DATA: TEXT1(30) VALUE 'This is the first text',  
      TEXT2(30) VALUE 'This is the second text',  
      TEXT3(30) VALUE 'This is the third text',  
      STRING(5) VALUE 'eco'.
```

```
IF TEXT1 CS STRING.  
    WRITE / 'Condition 1 is fulfilled'.  
ELSEIF TEXT2 CS STRING.  
    WRITE / 'Condition 2 is fulfilled'.  
ELSEIF TEXT3 CS STRING.  
    WRITE / 'Condition 3 is fulfilled'.  
ELSE.  
    WRITE / 'No condition is fulfilled'.  
ENDIF.
```

The output appears as follows:

Condition 2 is fulfilled.

Here, the second logical expression TEXT2 CS STRING is true because the string "eco" occurs in TEXT2.

Conditional Branching with CASE

Conditional Branching with CASE

To execute different statement blocks depending on the contents of particular data fields, you use the CASE statement as follows:

Syntax

```
CASE <f>.
  WHEN <f1>.
    <statement block>
  WHEN <f2>.
    <statement block>
  WHEN <f3>.
    <statement block>
  WHEN...
  .....
  WHEN OTHERS.
    <statement block>
ENDCASE.
```

The system executes the statement block after the WHEN statement if the contents of <f> equals the contents of <f_i>, and continues processing after the ENDCASE statement. The statement block after the optional WHEN OTHERS statement is executed if the contents of <f> does not equal any of the <f_i> contents. The last statement block must be concluded with ENDCASE.

The conditional branching using CASE is a shorter form of similar processing with IF:

```
IF <f> = <f1>.
  <statement block>
ELSEIF <f> = <f2>.
  <statement block>
ELSEIF <f> = <f3>.
  <statement block>
ELSEIF <f> =...
  ...
ELSE.
  <statement block>
ENDIF.
```

ABAP allows you to nest CASE... ENDCASE blocks and combine them with IF... ENDIF blocks. However, they must start and finish within the same processing block.

```
DATA: TEXT1 VALUE 'X',
      TEXT2 VALUE 'Y',
      TEXT3 VALUE 'Z',
      STRING VALUE 'A'.

CASE STRING.
  WHEN TEXT1.
    WRITE: / 'String is', TEXT1.
  WHEN TEXT2.
```

```

        WRITE: / 'String is', TEXT2.
    WHEN TEXT3.
        WRITE: / 'String is', TEXT3.
    WHEN OTHERS.
        WRITE: / 'String is not', TEXT1, TEXT2, TEXT3.
    ENDCASE.

```

The output appears as follows:

String is not X Y Z

Here, the last statement block after WHEN OTHERS is processed because the contents of STRING, "A", does not equal "X", "Y", or "Z".

Unconditional Looping using DO

Unconditional Looping using DO

If you want to process a statement block more than once, you can program a loop with the DO statement as follows:

Syntax

```
DO [<n> TIMES] [VARYING <f> FROM <f1> NEXT <f2>].
```

```
    <statement block>
```

```
ENDDO.
```

The system continues processing the statement block introduced by DO and concluded by ENDDO until it finds an EXIT, STOP, or REJECT statement (see [Terminating Loops \[Page 256\]](#)).

You can limit the number of times the loop is processed with the TIMES option. <n> can be literal or a variable. If <n> is 0 or negative, the system does not process the loop.

The system field SY-INDEX contains the number of times the loop has been processed.

Avoid endless loops when working with the DO statement. If you do not use the TIMES option, include at least one EXIT, STOP, or REJECT statement in the statement block so that the system can leave the loop.

This example shows the basic form of a DO loop.

```
DO.
```

```
WRITE SY-INDEX.
```

```
    IF SY-INDEX = 3.
```

```
        EXIT.
```

```
    ENDIF.
```

```
ENDDO.
```

The output appears as follows:

```
      1      2      3
```

The outer loop is processed three times. Here, the processing passes through the loop three times and then leaves it after the EXIT statement.

You can nest DO loops as many times as you like and also combine them with other loops.

This example shows two nested loops, both using the TIMES option.

```
DO 2 TIMES.
```

```
    WRITE SY-INDEX.
```

```
    SKIP.
```

```
    DO 3 TIMES.
```

```
        WRITE SY-INDEX.
```

```
    ENDDO.
```

```
SKIP.
```

Unconditional Looping using DO

ENDDO.

The output appears as follows:

```

1
1      2      3
2
1      2      3

```

The outer loop is processed twice. Each time the outer loop is processed, the inner loop is processed three times. Note that the system field SY-INDEX contains the number of loop passes for each loop individually.

You can assign new values to a variable <f> in each loop pass by using the VARYING option. <f₁>, <f₂>, <f₃>, ... must be a series of equidistant fields of the same type and length in memory. In the first loop pass, <f₁> is assigned to <f>; in the second loop pass <f₂> is assigned to <f>; and so on. You can use several VARYING options in one DO statement.

If you change the control variable <f> inside the DO loop, the system changes the corresponding field <f₁> automatically.

Ensure that you do not process the loop more times than the number of variables <f₁>, <f₂>, <f₃>, ... involved, since this may result in a runtime error.

This example shows how VARYING options can be used in a DO loop.

```

DATA: BEGIN OF TEXT,
      WORD1(4) VALUE 'This',
      WORD2(4) VALUE 'is',
      WORD3(4) VALUE 'a',
      WORD4(4) VALUE 'loop',
      END OF TEXT.
DATA: STRING1(4), STRING2(4).
DO 4 TIMES VARYING STRING1 FROM TEXT-WORD1 NEXT TEXT-WORD2.
  WRITE STRING1.
  IF STRING1 = 'is'.
    STRING1 = 'was'.
  ENDIF.
ENDDO.
SKIP.
DO 2 TIMES VARYING STRING1 FROM TEXT-WORD1 NEXT TEXT-WORD3
  VARYING STRING2 FROM TEXT-WORD2 NEXT TEXT-WORD4.
  WRITE: STRING1, STRING2.
ENDDO.

```

The output appears as follows:

```

This is  a  loop
This was a  loop

```

The field string TEXT represents a series of four equidistant fields in memory. Each time the first DO loop is processed, its components are assigned one by one

Unconditional Looping using DO

to STRING1. If STRING1 contains "is", it is changed to "was" and TEXT-WORD2 is automatically changed to "was". Each time the second DO loop is processed, the components of TEXT are passed to STRING1 and to STRING2.

Conditional Loops using WHILE

If you want to process a statement block more than once as long as a condition is true, you can program a loop with the WHILE statement as follows:

Syntax

```
WHILE <condition> [VARY <f> FROM <f1> NEXT <f2>].
  <statement block>
ENDWHILE.
```

The system continues processing the statement block introduced by WHILE and concluded by ENDWHILE statements as long as <condition> is true or until the system finds an EXIT, STOP, or REJECT statement (see [Terminating Loops \[Page 256\]](#)).

For <condition>, use any logical expression described in [Programming Logical Expressions \[Page 235\]](#).

The system field SY-INDEX contains the number of times the loop has been processed.

You can nest WHILE loops any number of times and also combine them with other loops.

The VARY option of the WHILE statement works in the same way as the VARYING option of the DO statement (see [Unconditional Loop Processing Using DO \[Page 251\]](#)). It allows you to assign new values to a variable <f> each time the loop is processed. <f₁>, <f₂>, <f₃>, ... must be a series of equidistant fields of the same type and length in memory. In the first loop pass, <f₁> is assigned to <f>; in the second loop pass <f₂> is assigned to <f>; and so on. You can use several VARY options in one WHILE statement.

Avoid endless loops when working with the WHILE statement. Remember that the condition in the WHILE statement should become false at some time or that the system should be able to leave the loop by finding an EXIT, STOP, or REJECT statement.

```
DATA: LENGTH  TYPE I VALUE 0,
      STRL    TYPE I VALUE 0,
      STRING(30) TYPE C VALUE 'Test String'.
```

```
STRL = STRLEN( STRING).
```

```
WHILE STRING NE SPACE.
```

```
  WRITE STRING(1).
```

```
  LENGTH = SY-INDEX.
```

```
  SHIFT STRING.
```

```
ENDWHILE.
```

```
WRITE: / 'STRLEN:      ', STRL.
```

```
WRITE: / 'Length of string:', LENGTH.
```

The output appears as follows:

```
T e s t   S t r i n g
```

```
STRLEN:                11
```

```
Length of string:      11
```

Conditional Loops using WHILE

Here, a WHILE loop is used to determine the length of a character string. This is done by shifting the string one position to the left each time the loop is processed until it contains only blanks. This example has been chosen to demonstrate the WHILE statement. However, the string length itself can be determined more easily and more efficiently by using the built-in function STRLEN, which is also shown in the example.

Terminating Loops

To terminate the processing of a loop, use one of the following keywords.

Keyword	Meaning
CONTINUE	Terminating a Loop Pass Unconditionally [Page 257]
CHECK	Terminating a Loop Pass Conditionally [Page 258]
EXIT	Terminating a Loop Entirely [Page 259]

You can only use the CONTINUE keyword in loops. The CHECK and EXIT keywords can also be used outside loops. There, they fulfill different functions. For example, they can terminate subroutines or an entire processing block. For more information about the CHECK and EXIT statements and how they work outside loops, see [Exiting Subroutines \[Page 474\]](#) and [Leaving Processing Blocks \[Page 1233\]](#).

The following topics describe how CONTINUE, CHECK, and EXIT work in DO and WHILE loops, as well as in

- LOOP - ENDLOOP loops, which are used to process internal tables (see [Processing Loops \[Page 324\]](#)).
- SELECT... ENDSELECT loops, for reading data from database tables (see [Selecting All Data from Several Lines \[Page 544\]](#)).

Terminating a Loop Pass Unconditionally

Terminating a Loop Pass Unconditionally

To terminate a loop pass immediately without any condition, use the CONTINUE statement as follows:

Syntax

CONTINUE.

After a CONTINUE statement, the system skips all the remaining statements in the current statement block and continues with the next loop pass.

```
DO 4 TIMES.  
  IF SY-INDEX = 2.  
    CONTINUE.  
  ENDIF.  
  WRITE SY-INDEX.  
ENDDO.
```

The output appears as follows:

```
1      3      4
```

Here, the system terminates the second loop pass without processing the WRITE statement.

Terminating a Loop Pass Conditionally

To terminate a loop pass conditionally, use the CHECK statement as follows:

Syntax

CHECK <condition>.

If the condition is **false**, the system skips all the remaining statements in the current statement block and continues with the next loop pass. For <condition>, use any logical expression described in [Programming Logical Expressions \[Page 235\]](#).

```
DO 4 TIMES.  
    CHECK SY-INDEX BETWEEN 2 and 3.  
    WRITE SY-INDEX.  
ENDDO.
```

The output appears as follows:

```
      2      3
```

Here, the system terminates the first and the fourth loop pass without processing the WRITE statement because SY-INDEX does not fall between 2 and 3.

Terminating a Loop Entirely

Terminating a Loop Entirely

To terminate a loop entirely without any condition, use the EXIT statement as follows:

Syntax

EXIT.

After an EXIT statement, the system immediately leaves the loop and continues the processing after the closing statement (ENDDO, ENDWHILE, ENDSELECT). Within nested loops, the system only leaves the current loop.

```
DO 4 TIMES.  
  IF SY-INDEX = 3.  
    EXIT.  
  ENDIF.  
  WRITE SY-INDEX.  
ENDDO.
```

The output appears as follows:

```
1      2
```

Here, the system terminates the entire loop processing in the third loop pass without processing the WRITE statement or the fourth loop pass.

Creating and Processing Internal Tables

This section deals with internal tables. Besides field strings, internal tables constitute another of the structured data types provided by ABAP.

The topics in this section describe

[What are Internal Tables? \[Page 261\]](#)

[Creating Internal Tables \[Page 270\]](#)

[Working with Internal Tables \[Page 277\]](#)

When you are working with large volumes of data in internal tables, the amount of computer time expended can be crucial to performance. For examples of how to achieve optimum performance, choose *Test → Runtime analysis* on the *ABAP Development Workbench* initial screen (or Transaction SE30), and choose *Tips & Tricks*. Under *Internal Tables*, you will find examples of different tasks which show you how to improve performance.

What are Internal Tables?

What are Internal Tables?

Internal tables provide a means of taking data from a fixed structure and storing it in working memory in ABAP. The data is stored line by line in memory, and each line has the same structure. Each component of a line is called a column in the internal table.

Internal tables can exist in ABAP as data types or as data objects. A data type is an abstract description of an internal type. Concrete data objects are instances of the data type.

The following topics give you an introduction to internal tables:

[Internal Tables as Data Types \[Page 262\]](#)

[Internal Tables as Dynamic Data Objects \[Page 263\]](#)

[Using Internal Tables \[Page 264\]](#)

[Accessing Internal Tables \[Page 265\]](#)

[Operations on Internal Tables \[Page 266\]](#)

[Choosing a Table Type \[Page 267\]](#)

[Declaring Internal Tables \[Page 268\]](#)

[Internal Tables as Parameters for Routines \[Page 269\]](#)

Internal Tables as Data Types

Internal tables and structures are the two structured data types in ABAP (see [Data Types \[Page 105\]](#)). The data type of an internal table is fully defined by its

- Line type
The line type of an internal table can be any ABAP data type (even another internal table)
- Key
The key identifies table rows. There are two kinds of key for internal tables - the standard key and a user-defined key. You can specify whether the key should be UNIQUE or NON-UNIQUE. Internal tables with a unique key cannot contain duplicate entries. The uniqueness depends on the table access method.
 - The standard key is made up of all non-numeric columns in the internal table which themselves are not, and do not contain, internal tables.
 - The user-defined key can be made up of any columns of the internal table which themselves are not, and do not contain, internal tables. Internal tables with a user-defined key are called key tables. When you define the key, the sequence of its fields is significant. You should remember this, for example, if you intend to sort the table according to the key.
- Access method
The access method determines how ABAP will access individual table entries. Internal tables can be divided into three groups according to access method:
 - Standard tables have an internal linear index. The system can access records either by using the table index or the key. The response time for key access is proportional to the number of entries in the table. The key of a standard table is not always unique. You may not specify a unique key when you define the table.
 - Sorted tables are always saved sorted by the key. They also have an internal linear index. The system can access records either by using the table index or the key. The response time for key access is logarithmically proportional to the number of table entries, since the system uses a binary search. The key of a sorted table can be either unique or non-unique. In the table definition, you must specify whether the key is to be UNIQUE or NON-UNIQUE. Standard tables and sorted tables belong to the generic category of index tables.
 - Hashed tables have no linear index. You can only access a hashed table using its key. The response time is independent of the number of table entries, and is constant, since the system access the table entries using a hash algorithm. The key of a hashed table must be unique. When you define the table, you must specify the key as UNIQUE.

Unlike other user-defined ABAP data types, the data type of an internal table does not have to be fully specified, but can be constructed generically. This means that you do not necessarily have to give the key (or the line type and the key) when you define the table.

Internal Tables as Dynamic Data Objects

Data objects that are defined either with the data type of an internal table, or directly as an internal table, are always fully defined in respect of their line type, key and access method. However, the number of lines is not fixed. Thus internal tables are dynamic data objects, since they can contain any number of lines of a particular type. The only restriction on the number of lines an internal table may contain are the limits of your system installation. The line types of internal tables can be any ABAP data types - elementary, structured, or internal tables. The individual lines of an internal table are called table lines or table entries.

Using Internal Tables

In ABAP, internal tables fulfill the function of arrays. Since they are dynamic data objects, they save the programmer the task of dynamic memory management in his or her programs. You should use internal tables whenever you want to process a dataset with a fixed structure within a program. A particularly important use for internal tables is for storing and formatting data from a database table within a program. They are also the best way of including very complicated data structures in an ABAP program.

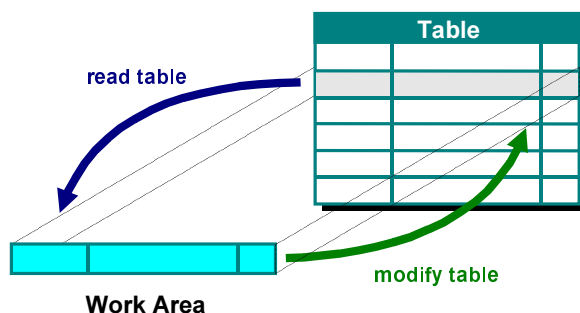
Accessing Internal Tables

Accessing Internal Tables

There are two ways of working with an internal table. You can either address the whole table, or an individual entry.

If you access the whole table body, you can use the table name in ABAP statements such as MOVE, as though it were an elementary field. This then affects the whole data object.]

When you access individual table entries, you are not working directly with the data in the table. Instead, you work with another data object as a work area. The work area is an interface to the entries in the internal table, and must at least be convertible into the line type of the internal table. Your work areas will usually have the same data type as the lines of the internal table. When you read data from a table record, the data you are reading overwrites the current contents of the work area. You can then use this data in the program. When you write data to the internal table, this must first be placed in the work area. The system then transfers it from the work area to the appropriate table entry.



You can create internal tables with a header line. This has the same data type as a table entry. You can then use the header line as a table work area. The work area and the internal table have the same name. The ABAP statements that you use to access individual table entries can use the header line implicitly as a work area. When you use ABAP statements that access the whole table, you must write the internal table name <name>[]. This distinguishes the internal table from the header line. (The system normally interprets the name in these statements as the name of the work area, and not as that of the internal table). In order to avoid this potential confusion, you should create internal tables without a header line where possible. In particular, internal tables nested in structures or other internal tables must not have a header line, since this can lead to ambiguous expressions.

table with header line

```
LOOP AT <itab>.
  ....
  WRITE <itab>-<field>.
  ....
ENDLOOP.
```

```
APPEND <itab>.
MODIFY <itab>.
READ TABLE <itab>.
```

table without header

```
LOOP AT <itab> INTO <wa>.
  ....
  WRITE <wa>-<field>.
  ....
ENDLOOP.
```

```
APPEND <wa> TO <itab>.
MODIFY <itab> FROM <wa>.
READ TABLE <itab> INTO <wa>.
```

Operations on Internal Tables

The following are typical operations on the individual entries in an internal table:

- Filling the table using the ABAP commands
 - INSERT
 - APPEND
 - COLLECT
- Reading the table using the ABAP commands
 - READ
 - LOOP
- Modifying the table using the ABAP command
 - MODIFY
- Deleting entries using the ABAP command
 - DELETE

Which ABAP statements you use depends on the access method of the internal table. For example, you would use APPEND to fill a standard table, but INSERT to fill a hashed table. There is also a set of generic ABAP commands for cases where you do not know the table type when you write the program (for example, in routines):

- INSERT... INTO TABLE
- READ TABLE
- MODIFY TABLE
- DELETE TABLE

You can use these commands for all table types. The system executes them appropriately to the type of internal table in question.

Choosing a Table Type

Choosing a Table Type

The table type (and particularly the access method) that you will use depends on how the typical internal table operations will be most frequently executed.

- Standard table

This is the most appropriate type if you are going to address the individual table entries using the index. Index access is the quickest possible access. You should fill a standard table by appending lines (ABAP APPEND statement), and read, modify and delete entries by specifying the index (INDEX option with the relevant ABAP command). The access time for a standard table increases in a linear relationship with the number of table entries. If you need key access, standard tables are particularly useful if you can fill and process the table in separate steps. For example, you could fill the table by appending entries, and then sort it. If you use the binary search option with key access, the response time is logarithmically proportional to the number of table entries.

- Sorted table

This is the most appropriate type if you need a table which is sorted as you fill it. You fill sorted tables using the INSERT statement. Entries are inserted according to the sort sequence defined through the table key. Any illegal entries are recognized as soon as you try to add them to the table. The response time for key access is logarithmically proportional to the number of table entries, since the system always uses a binary search. Sorted tables are particularly useful for partially sequential processing in a LOOP if you specify the beginning of the table key in the WHERE condition.

- Hashed table

This is the most appropriate type for any table where the main operation is key access. You cannot access a hashed table using its index. The response time for key access remains constant, regardless of the number of table entries. Like database tables, hashed tables always have a unique key. Hashed tables are useful if you want to construct and use an internal table which resembles a database table.

Declaring Internal Tables

Like other ABAP data objects, you can either create an internal table as a data types (TYPES statement) and then create a data object based on it, or you can create a fully-defined data object directly (DATA statement). When you create an internal table as a data object, you should ensure that only the administration entry which belongs to an internal table is declared statically. Unlike all other ABAP data objects, you do not have to specify the memory required for an internal table. Table rows are added to and deleted from the table dynamically at runtime by the various statements for adding and deleting records.

When you declare an internal table, you can use the OCCURS parameter to reserve an initial amount of memory space. This can, in certain circumstances, affect the response time when you fill the table (see performance note for internal tables in the keyword documentation).

Internal Tables as Parameters for Routines

You can pass internal tables without a header line to subroutines and function modules, either by value or by reference. Tables with a header line may only be passed with reference to subroutines, using the TABLES addition.

Whenever you pass a table without a header line as an actual parameter to a formal parameter with a header line (TABLES), the system automatically creates the corresponding header line in the routine. Conversely, if you want to pass the body of a table with header line to a formal parameter without a header line, you can do this by placing brackets after the table name (<name>[]).

Creating Internal Tables

Release 3.1 documentation

When creating an internal table, you can decide whether you want to create an internal table data type with the TYPES statement first and then a data object with that type, or whether you want to create an internal table data object directly with the DATA statement. You can create internal table data objects with or without header lines.

The following topics explain about

[Creating Internal Tables \[Page 271\]](#)

[Creating Internal Table Data Objects \[Page 273\]](#)

Creating Internal Table Data Types

Creating Internal Table Data Types

To create an internal table data type, you use the [TYPES statement \[Page 130\]](#) as follows:

Syntax

TYPES <t> <type> OCCURS <n>.

This creates an internal table data type <t> by using the OCCURS option of the TYPES statement. The lines of the internal table have the data type specified in <type>. You can specify the data type of the lines either using the TYPE or LIKE parameter (see [The Basic Form of the DATA Statement \[Page 120\]](#)).

By using the LIKE parameter to refer to an object defined in the ABAP Dictionary, you can create internal tables which have the same line structure as objects stored in the Dictionary, and which reflect the structure of database tables. This is very important when reading and processing database tables (see [Reading and Processing Database Tables \[Page 538\]](#)).

<n> specifies an initial number of lines. Memory is reserved for the number of lines specified as soon as the first line is written to an internal table data object created with type <t>. If more lines are added to an internal table than specified by <n>, the reserved memory expands automatically. If there is not enough space in memory for an internal table, it is written to a buffer or to the disk (paging area).

```
TYPES VECTOR TYPE I OCCURS 10.
```

This example creates an internal table data type VECTOR which has lines consisting of the elementary type I field.

```
TYPES: BEGIN OF LINE,  
      COLUMN1 TYPE I,  
      COLUMN2 TYPE I,  
      COLUMN3 TYPE I,  
      END OF LINE.
```

```
TYPES ITAB TYPE LINE OCCURS 10.
```

This example creates an internal table data type ITAB which has lines with the same structure as the field string LINE.

```
TYPES VECTOR TYPE I OCCURS 10.
```

```
TYPES: BEGIN OF LINE,  
      COLUMN1 TYPE I,  
      COLUMN2 TYPE I,  
      COLUMN3 TYPE I,  
      END OF LINE.
```

```
TYPES ITAB TYPE LINE OCCURS 10.
```

```
TYPES: BEGIN OF DEEPLINE,  
      TABLE1 TYPE VECTOR,  
      TABLE2 TYPE ITAB,  
      END OF DEEPLINE.
```

TYPES DEEPTABLE TYPE DEEPLINE OCCURS 10.

This example creates the same internal table data types (VECTOR and ITAB) as in the above examples. It then creates a data type DEEPLINE as a field string which contains these internal tables as components. Via this field string, a data type DEEPTABLE is created as internal table. Therefore, the elements of this internal table are themselves internal tables.

Creating Internal Table Data Objects

To create an internal table data object, you can use the [DATA statement \[Page 119\]](#) in several ways. You can

[Create Internal Tables by Referring to Another Table \[Page 274\]](#)

[Create Internal Tables by Referring to a Structure \[Page 275\]](#)

[Create Internal Tables with a New Structure \[Page 276\]](#)

With the first two possibilities, the creation of a header line is optional, while tables that are created with a new line structure always have a header line.

Creating Internal Tables by Referring to Another Table

To create an internal table data object by referring to an existing internal table data type or data object, you use the DATA statement as follows:

Syntax

DATA <f> <type> [WITH HEADER LINE].

You can use the <type> option to refer to a table data type or table data object by using TYPE or LIKE (for more information about these options, see [The Basic Form of the DATA Statement \[Page 120\]](#)). The data object <f> is declared as an internal table with the same structure.

If you use the WITH HEADER LINE option, the internal table is created with a table work area <f> (see [Choosing a Table Type \[Page 267\]](#)).

If you want to create an internal table with a header line, the line type cannot directly be an internal table. However, it can be a structure which has internal tables as components.

```
TYPES: BEGIN OF LINE,  
       COLUMN1 TYPE I,  
       COLUMN2 TYPE I,  
       COLUMN3 TYPE I,  
       END OF LINE.
```

```
TYPES ITAB TYPE LINE OCCURS 10.
```

```
DATA TAB1 TYPE ITAB.
```

```
DATA TAB2 LIKE TAB1 WITH HEADER LINE.
```

As shown in [Creating Internal Table Data Types \[Page 271\]](#), this example creates a data type ITAB as an internal table. The data object TAB1 has the same structure as ITAB by referring to ITAB using the TYPE parameter of the DATA statement. The data object TAB2 has the same structure by referring to TAB1 using the LIKE parameter of the DATA statement. TAB2 is created with header line. Therefore, the table work area TAB2 can be addressed in the program by using TAB2-COLUMN1, TAB2-COLUMN2, and TAB2-COLUMN3.

Creating Internal Tables by Referring to a Structure

Creating Internal Tables by Referring to a Structure

To create an internal table data object by referring to an existing line structure, you use the DATA statement as follows:

Syntax

DATA <f> <type> OCCURS <n> [WITH HEADER LINE].

This creates an internal table <f> by using the OCCURS option of the DATA statement. The lines of the internal table have the data type specified in <type>. To specify the data type, you can use either the TYPE or the LIKE parameter (for more information about these parameters, see [The Basic Form of the DATA Statement \[Page 120\]](#)).

By using the LIKE parameter to refer to an object defined in the ABAP Dictionary, you can create internal tables which have the same line structure as objects stored in the Dictionary, and which reflect the structure of database tables. This is very important when reading and processing database tables (see [Reading and Processing Database Tables \[Page 538\]](#)).

<n> specifies an initial number of lines. Memory is reserved for the number of lines specified as soon as the first line is written to an internal table data object created with type <f>. If more lines are added to an internal table than specified by <n>, the reserved memory expands automatically. If there is not enough space in memory for an internal table, it is written to a buffer or to the disk (paging area).

The features described above are the same as those for creating internal table data types with the TYPES statement (see [Creating Internal Table Data Types \[Page 271\]](#)).

As an additional feature, you can use the WITH HEADER LINE option with the DATA statement. This creates a table work area <f> with the same structure as the lines of the internal table <f> (for more information about header lines, see [Choosing a Table Type \[Page 267\]](#)).

```
DATA FLIGHT_TAB LIKE SFLIGHT OCCURS 10.
```

This example creates a data object FLIGHT_TAB which has the same structure as the database table SFLIGHT.

This example shows you how to create the same internal table using two different procedures.

```
TYPES VECTOR_TYPE TYPE I OCCURS 10.  
DATA VECTOR TYPE VECTOR_TYPE WITH HEADER LINE.
```

Here, an internal table data type VECTOR_TYPE is created with lines consisting of an elementary type I field is created first. Then, a data object VECTOR is created as an internal table by referring to VECTOR_TYPE. A table work area VECTOR is also created by using the WITH HEADER LINE option. In this case, the table work area consists of one type I field which can be addressed by the name VECTOR.

```
DATA VECTOR TYPE I OCCURS 10 WITH HEADER LINE.
```

In this case, exactly the same data object VECTOR is created by using the OCCURS option directly in the DATA statement.

Creating Internal Tables with a New Structure

To create an internal table data object without referring either to an existing object or to an existing line structure, you use the DATA statement as follows:

Syntax

```
DATA: BEGIN OF <f> OCCURS <n>,  
      <component declaration>,  
      .....  
      END OF <f>.
```

This defines an internal table <f> and declares the components of its lines in <component declaration>. Apart from the OCCURS parameter, the syntax is the same as for defining structures (see [The DATA Statement for Structures \[Page 126\]](#)).

This statement **always** creates a table work area <f> (see [Choosing a Table Type \[Page 267\]](#)). Its effect is therefore the same as when you create a field string <f> first and then an internal table <f> with the same line structure as the field string.

<n> specifies an initial number of lines. Memory is reserved for the number of lines specified as soon as the first line is written to an internal table data object created with type <f>. If more lines are added to an internal table than specified by <n>, the reserved memory expands automatically. If there is not enough space in memory for an internal table, it is written to a buffer or to the disk (paging area).

```
DATA: BEGIN OF ITAB OCCURS 10,  
      COLUMN1 TYPE I,  
      COLUMN2 TYPE I,  
      COLUMN3 TYPE I,  
      END OF ITAB.
```

This example creates an internal table and the corresponding table work area ITAB.

Working with Internal Tables

The following topics describe how to work with internal tables. This section explains how to:

[Fill Internal Tables \[Page 278\]](#)

[Read Internal Tables \[Page 292\]](#)

[Change and Delete Lines of Internal Tables \[Page 309\]](#)

[Sort Internal Tables \[Page 319\]](#)

[Create Ranked Lists \[Page 323\]](#)

[Process Loops \[Page 324\]](#)

[Compare Internal Tables \[Page 332\]](#)

[Initialize Internal Tables \[Page 334\]](#)

Filling Internal Tables

To fill an internal table, you can either append data line by line, or you can copy the contents of another table.

To fill an internal table line by line, you can use either the APPEND, COLLECT, or INSERT statements.

- To use an internal table just for storing data, you are recommended to use APPEND for performance reasons. With APPEND, you can also create ranked lists.

[Appending Lines \[Page 279\]](#)

- To calculate totals of numeric fields or to ensure that no duplicate entries occur in an internal table, use the COLLECT statement which processes lines depending on the standard key.

[Appending Lines Depending on the Standard Key \[Page 281\]](#)

- To insert a new line before an existing line in an internal table, use the INSERT statement.

[Inserting Lines \[Page 283\]](#)

To copy the contents of one internal table into another one, use the variants of the APPEND, INSERT, or MOVE statements.

- To append lines of an internal table to another internal table, use a variant of the APPEND statement.

[Appending Lines of an Internal Table \[Page 286\]](#)

- To insert lines of an internal table in another internal table, use a variant of the INSERT statement.

[Inserting Lines of an Internal Table \[Page 287\]](#)

- To copy the entire contents of an internal table into another internal table, and overwrite that target table, use the MOVE statement.

[Copying Internal Tables \[Page 289\]](#)

For information about how to fill internal tables with data from database tables using the SELECT statement, see [Reading Data into an Internal Table \[Page 555\]](#).

Appending Lines

Appending Lines

To append a line to an internal table, use the APPEND statement as follows:

Syntax

APPEND [<wa> TO|INITIAL LINE TO] <itab>.

This statement appends a new line to the internal table <itab>.

By using the <wa> TO option, you specify the source area <wa> which you want to append. In the case of tables with a header line, you can omit the TO option. Then the table work area becomes the source area.

Intead of <wa> TO, you can use the addition INITIAL LINE TO. This addition adds a line full of initial values to the table.

APPEND works regardless of whether a line with the same standard key already exists or not (see [Using Internal Tables \[Page 264\]](#)). For this reason, you may get duplicate entries.

After each APPEND statement, the system field SY-TABIX contains the index of the appended line.

```
DATA: BEGIN OF ITAB OCCURS 10.
      COL1 TYPE C,
      COL2 TYPE I,
    END OF ITAB.

DO 3 TIMES.
  APPEND INITIAL LINE TO ITAB.
  ITAB-COL1 = SY-INDEX. ITAB-COL2 = SY-INDEX ** 2.
  APPEND ITAB.
ENDDO.

LOOP AT ITAB.
  WRITE: / ITAB-COL1, ITAB-COL2.
ENDLOOP.
```

This example creates an internal table ITAB with header line and two columns. The table is filled in the DO loop. Each time the processing passes through the loop, an initialized line is appended and then the table work area is filled with the loop index and the square root of the loop index and appended. The produces the following output:

```
      0
1      1
      0
2      4
      0
3      9
```

```
DATA: BEGIN OF LINE1,
      COL1(3) TYPE C,
      COL2(2) TYPE N,
```

```
      COL3  TYPE I,  
      END OF LINE1.  
  
DATA TAB1 LIKE LINE1 OCCURS 10.  
  
DATA: BEGIN OF LINE2,  
      FIELD1(1) TYPE C,  
      FIELD2  LIKE TAB1,  
      END OF LINE2.  
  
DATA  TAB2 LIKE LINE2 OCCURS 1.  
  
LINE1-COL1 = 'abc'. LINE1-COL2 = '12'. LINE1-COL3 = 3.  
APPEND LINE1 TO TAB1.  
  
LINE1-COL1 = 'def'. LINE1-COL2 = '34'. LINE1-COL3 = 5.  
APPEND LINE1 TO TAB1.  
  
LINE2-FIELD1 = 'A'. LINE2-FIELD2 = TAB1.  
APPEND LINE2 TO TAB2.  
  
REFRESH TAB1.  
  
LINE1-COL1 = 'ghi'. LINE1-COL2 = '56'. LINE1-COL3 = 7.  
APPEND LINE1 TO TAB1.  
  
LINE1-COL1 = 'jkl'. LINE1-COL2 = '78'. LINE1-COL3 = 9.  
APPEND LINE1 TO TAB1.  
  
LINE2-FIELD1 = 'B'. LINE2-FIELD2 = TAB1.  
APPEND LINE2 TO TAB2.  
  
LOOP AT TAB2 INTO LINE2.  
  WRITE: / LINE2-FIELD1.  
  LOOP AT LINE2-FIELD2 INTO LINE1.  
    WRITE: / LINE1-COL1, LINE1-COL2, LINE1-COL3.  
  ENDLOOP.  
ENDLOOP.
```

The produces the following output:

```
A  
abc 12      3  
def 34      5  
  
B  
ghi 56      7  
jkl 78      9
```

This example creates two internal tables (TAB1 and TAB2) without a table work area. TAB2 has a deep structure because the second component of LINE2 has the structure of internal table TAB1. LINE1 is filled and appended to TAB1. Then, LINE2 is filled and appended to TAB2. After clearing TAB1 with the REFRESH statement (see [Initializing Internal Tables \[Page 334\]](#)), the same procedure is repeated. Note that the number of lines in TAB2 was specified only with 1 in the OCCURS parameter. The contents of TAB2 is output.

Appending Lines Depending on the Standard Key

Appending Lines Depending on the Standard Key

To fill an internal table with lines which have unique standard keys, you use the COLLECT statement as follows:

Syntax

COLLECT [<wa> INTO] <itab>.

This statement specifies the source area <wa> which you want to append by using the INTO option. In the case of tables with a header line, you can omit the INTO option. Then the table work area becomes the source area.

The system checks whether a table entry with the same standard key exists (all non-numeric fields, see [Operations on Internal Tables \[Page 266\]](#)). If not, the COLLECT statement has the same effect as the APPEND statement and a new line is appended to the table.

If an entry with the same key already exists, the COLLECT statement does not append a new line, but adds the contents of the numeric fields in the work area to the contents of the numeric fields in the existing entry. The system field SY-TABIX contains the index of the processed line.

The work area you specify for COLLECT must be compatible with, and not only convertible to, the line type of the internal table. You cannot use the COLLECT statement with internal tables of deep structure, i.e. with lines which have internal tables as components.

If you use only the COLLECT statement to fill an internal table, no duplicate entries can occur. To fill an internal table without duplicate entries, you should therefore use COLLECT rather than APPEND or INSERT.

```
DATA: BEGIN OF ITAB OCCURS 3,
      COLUMN1(3) TYPE C,
      COLUMN2(2) TYPE N,
      COLUMN3   TYPE I,
    END OF ITAB.

ITAB-COLUMN1 = 'abc'. ITAB-COLUMN2 = '12'. ITAB-COLUMN3 = 3.
COLLECT ITAB.
WRITE / SY-TABIX.

ITAB-COLUMN1 = 'def'. ITAB-COLUMN2 = '34'. ITAB-COLUMN3 = 5.
COLLECT ITAB.
WRITE / SY-TABIX.

ITAB-COLUMN1 = 'abc'. ITAB-COLUMN2 = '12'. ITAB-COLUMN3 = 7.
COLLECT ITAB.
WRITE / SY-TABIX.

LOOP AT ITAB.
  WRITE: / ITAB-COLUMN1, ITAB-COLUMN2, ITAB-COLUMN3.
ENDLOOP.
```

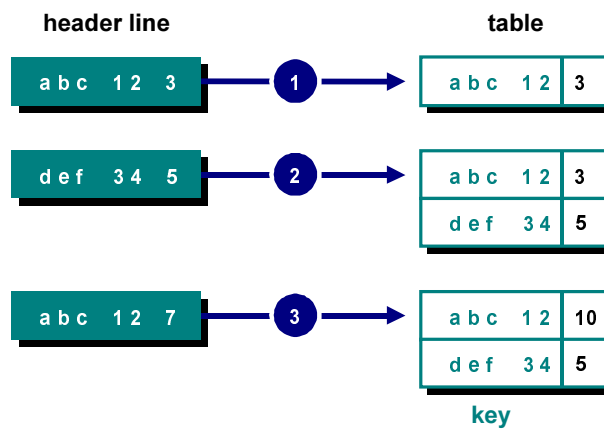
The produces the following output:

```
1
2
```

Appending Lines Depending on the Standard Key

```
1
abc 12      10
def 34      5
```

This example creates an internal table ITAB with a table work area. The first two COLLECT statements work like APPEND statements. In the third COLLECT statement, the first line of ITAB is modified. The following figure diagram shows the three steps:



Inserting Lines

Inserting Lines

To insert a new line before a line in an internal table, you use the INSERT statement as follows:

Syntax

```
INSERT [<wa> INTO|INITIAL LINE INTO] <itab> [INDEX <idx>].
```

This statement specifies the source area <wa> which you want to insert using the INTO option. In the case of tables with a header line, you can omit the TO option. Then the table work area becomes the source area.

Instead of <wa> TO, you can use the addition INITIAL LINE INTO. This addition adds a line full of initial values to the table.

If you use the INDEX option, the new line is inserted before the line which has the index <idx>. After the insertion, the new entry has the index <idx> and the index of the following lines is incremented by 1.

If the table consists of <idx> - 1 entries, the system appends the new entry after the last existing table line. If the table has less than <idx> - 1 entries, the system cannot insert the entry and SY-SUBRC is set to 4. If an operation is successful, SY-SUBRC is set to 0.

If you use the INSERT statement without the INDEX option, the system can process it only within a LOOP - ENDLOOP loop by inserting the new entry before the current line (i.e. the line which has the index returned by SY-TABIX).

```
DATA: BEGIN OF LINE,
      COL1 TYPE I,
      COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE LINE OCCURS 10.

DO 2 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

LINE-COL1 = 11. LINE-COL2 = 22.

INSERT LINE INTO ITAB INDEX 2.

INSERT INITIAL LINE INTO ITAB INDEX 1.

LOOP AT ITAB INTO LINE.
  WRITE: / SY-TABIX, LINE-COL1, LINE-COL2.
ENDLOOP.
```

The produces the following output:

1	0	0
2	1	1
3	11	22
4	2	4

This example creates an internal table ITAB and fills it with two lines. A new line containing values is inserted before the second line. Then, an initialized line is inserted before the first line.

```
DATA: BEGIN OF LINE,
      COL1 TYPE I,
      COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE LINE OCCURS 10.

DO 2 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

LOOP AT ITAB INTO LINE.
  LINE-COL1 = 3 * SY-TABIX. LINE-COL2 = 5 * SY-TABIX.
  INSERT LINE INTO ITAB.
ENDLOOP.

LOOP AT ITAB INTO LINE.
  WRITE: / SY-TABIX, LINE-COL1, LINE-COL2.
ENDLOOP.
```

The produces the following output:

1	3	5
2	1	1
3	6	10
4	2	4

This example creates an internal table ITAB and fills it with two lines. A new line is inserted before each existing line inside the first LOOP - ENDLOOP loop. The following diagram shows how the example works:

Inserting Lines

	col1	col2
	itab	
1	1	1
2	2	4

1			3	5	1st loop
2	1	1			
3	2	4			

1	3	5			2nd loop
2	1	1			
3			6	10	
4	2	4			

Appending Lines of an Internal Table

To append part or all of an internal table to another internal table, you use the APPEND statement as follows:

Syntax

APPEND LINES OF <itab1> [FROM <n1>] [TO <n2>] TO <itab2>.

Without the FROM and TO options, this statement appends the entire table <itab1> to <itab2>. If you use these options, you can specify the first or the last line of <itab1> to be appended using its index <n1> or <n2>.

This method of appending lines of one table to another is about 3 to 4 times faster than appending them line by line in a loop.

After the APPEND statement, the system field SY-TABIX contains the index of the last line appended.

```
DATA: BEGIN OF ITAB OCCURS 10.
      COL1 TYPE C,
      COL2 TYPE I,
    END OF ITAB.

DATA JTAB LIKE ITAB OCCURS 10 WITH HEADER LINE.

DO 3 TIMES.
  ITAB-COL1 = SY-INDEX. ITAB-COL2 = SY-INDEX ** 2.
  APPEND ITAB.
  JTAB-COL1 = SY-INDEX. JTAB-COL2 = SY-INDEX ** 3.
  APPEND JTAB.
ENDDO.

APPEND LINES OF JTAB FROM 2 TO 3 TO ITAB.

LOOP AT ITAB.
  WRITE: / ITAB-COL1, ITAB-COL2.
ENDLOOP.
```

This example creates two internal tables of the same type, ITAB and JTAB, both with a header line. In the DO loop, ITAB is filled with a list of square numbers, and JTAB with a list of cube numbers. Then, the last two lines of JTAB are appended to ITAB. The output of ITAB is as follows:

1	1
2	4
3	9
2	8
3	27

Inserting Lines of an Internal Table

Inserting Lines of an Internal Table

To insert part or all of an internal table into another internal table, you use the INSERT statement as follows:

Syntax

```
INSERT LINES OF <itab1> [FROM <n1>] [TO <n2>]
      INTO <itab2> [INDEX <idx>].
```

Without the FROM and TO options, this statement inserts the entire table <itab1> into <itab2>. If you use these options, you can specify the first or the last line of <itab1> to be inserted using its index <n1> or <n2>.

If you use the INDEX option, the lines of <itab1> are inserted before the line in <itab2> which has the index <idx>. If you do not use the INDEX option, the system can only process it in a LOOP - ENDLOOP block by inserting the new entries before the current line (i.e. the line with the index returned in SY-TABIX.)

Depending on the size of the tables and where they are inserted, this method of inserting lines of one table into another can be up to 20 times faster than inserting them line by line in a loop.

```
DATA: BEGIN OF LINE,
      COL1 TYPE I,
      COL2 TYPE I,
      END OF LINE.

DATA: ITAB LIKE LINE OCCURS 10,
      JTAB LIKE LINE OCCURS 10.

DO 3 TIMES.
  LINE-COL1 = SY-INDEX. LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.

  LINE-COL1 = SY-INDEX. LINE-COL2 = SY-INDEX ** 3.
  APPEND LINE TO JTAB.
ENDDO.

INSERT LINES OF ITAB INTO JTAB INDEX 1.

LOOP AT JTAB INTO LINE.
  WRITE: / SY-TABIX, LINE-COL1, LINE-COL2.
ENDLOOP.
```

This example creates two internal tables of the same type, ITAB and JTAB, and fills them with 3 lines. Then, the entire table ITAB is inserted before the first line of JTAB. The output of JTAB is as follows:

1	1	1
2	2	4
3	3	9
4	1	1
5	2	8

6 3 27

Copying Internal Tables

Copying Internal Tables

If you want to copy the entire contents of one internal table into another in one execution, use the MOVE statement or the assignment operator (=) as follows:

Syntax

MOVE <itab1> TO <itab2>.

This statement is equivalent to:

<itab2> = <itab1>.

Multiple assignments such as

<itab4> = <itab3> = <itab2> = <itab1>.

are also possible. ABAP processes them from right to left:

<itab2> = <itab1>.

<itab3> = <itab2>.

<itab4> = <itab3>.

These statements perform a complete operation. The contents of the entire table is copied, including the data of any other internal tables which are components of the table. The original contents of the target table are overwritten.

The syntax is the same as for copying elementary fields (for further information, see [Basic Assignments \[Page 174\]](#)).

For tables with a header line, the table work area and the table itself have the same name. To distinguish between them in the above statements, you must enter two square brackets ([]) after the name to address the internal table and not the table work area.

You can copy internal tables only to other internal tables. Internal tables are not convertible to structures or elementary fields. To copy internal tables into other internal tables, their line types must be convertible (for more information about convertibility, see [Convertibility of Internal Tables \[Page 232\]](#)).

```
DATA: BEGIN OF LINE,
      COL1,
      COL2,
      END OF LINE.

DATA ETAB LIKE LINE OCCURS 10 WITH HEADER LINE.
DATA FTAB LIKE LINE OCCURS 10.

LINE-COL1 = 'A'. LINE-COL2 = 'B'.
APPEND LINE TO ETAB.

MOVE ETAB[] TO FTAB.

LOOP AT FTAB INTO LINE.
  WRITE: / LINE-COL1, LINE-COL2.
ENDLOOP.
```

The produces the following output:

```
A B
```

This example creates two internal tables ETAB and FTAB with the same line structure LINE. ETAB is created with a table work area. After filling ETAB using the APPEND statement, its contents are copied to FTAB. Note the use of the square brackets ([]).

```
DATA: ITAB TYPE I OCCURS 10,  
      FTAB TYPE F OCCURS 10,  
      FL TYPE F.
```

```
DO 3 TIMES.  
  APPEND SY-INDEX TO ITAB.  
ENDDO.
```

```
FTAB = ITAB.
```

```
LOOP AT FTAB INTO FL.  
  WRITE: / FL.  
ENDLOOP.
```

The produces the following output:

```
1.0000000000000000E+00  
2.0000000000000000E+00  
3.0000000000000000E+00
```

Here, the internal table ITAB has the elementary line type I and FTAB has the elementary line type F. These line types are convertible (for further information about convertibility, see [Convertibility of Elementary Data Types \[Page 219\]](#)) and ITAB can be copied to FTAB.

```
DATA: BEGIN OF ILINE,  
      NUM TYPE I,  
    END OF ILINE,  
      BEGIN OF FLINE,  
      NUM TYPE F,  
    END OF FLINE,  
      ITAB LIKE ILINE OCCURS 10,  
      FTAB LIKE FLINE OCCURS 10.
```

```
DO 3 TIMES.  
  ILINE-NUM = SY-INDEX.  
  APPEND ILINE-NUM TO ITAB.  
ENDDO.
```

```
FTAB = ITAB.
```

```
LOOP AT FTAB INTO FLINE.  
  WRITE: / FLINE-NUM.  
ENDLOOP.
```

The produces the following output:

```
6.03823403895813E-154  
6.03969074613219E-154
```

Copying Internal Tables

6.04114745330626E-154

Here, the line types of the internal tables ITAB and FTAB are field strings, each with one component of type I or F. Therefore, when assigning ITAB to FTAB, the contents of Table ITAB are converted to type C fields and then written to FTAB (for more information about convertibility of field strings, see [Convertibility of Structures \[Page 227\]](#)). The system interprets the transferred data as type F fields, and obtains meaningless results.

Reading Internal Tables

To read the contents of internal tables for further processing, you can use either the LOOP or the READ statement.

You use the the LOOP statement to read internal tables line by line.

[Reading Internal Tables Line by Line \[Page 293\]](#)

You can select a single line by the READ statement:

[Reading Single Lines Using the Index \[Page 295\]](#)

[Reading Single Lines Using a Key \[Page 296\]](#)

Once you have read the line, you can

[Compare the Contents of Single Lines \[Page 303\]](#)

[Read Parts of Single Lines \[Page 305\]](#)

You can use the SEARCH statement to search internal tables for string patterns.

[Searching Internal Tables for Character Strings \[Page 306\]](#)

You can use the DESCRIBE statement to determine the attributes of internal tables.

[Determining the Attributes of Internal Tables \[Page 308\]](#)

Reading Internal Tables Line by Line

Reading Internal Tables Line by Line

To read an internal table into a work area line by line, you can program a loop using the LOOP statement. The syntax is as follows:

Syntax

```
LOOP AT <itab> [INTO <wa>] [FROM <n1>] [TO <n2>]
      [WHERE <condition>].
```

```
.....
```

```
ENDLOOP.
```

You specify the target area <wa> using the INTO option. In the case of tables with a header line, you can omit the INTO option. The table work area then becomes the target area.

The internal table <itab> is read line by line into <wa> or into the table work area <itab>. For each line read, the system processes the statement block introduced by LOOP and concluded by ENDLOOP. You can control the flow of the statement block within a LOOP - ENDLOOP block with the control keyword AT (see [Loop Processing \[Page 324\]](#)).

Within the statement block, the system field SY-TABIX contains the index of the current line. The loop ends as soon as all lines of the table have been processed. After the ENDLOOP statement, the system field SY-SUBRC is set to 0 if at least one line was read. Otherwise it is set to 4.

You can restrict the number of lines to be processed in the loop by using the FROM, TO, or WHERE options.

- By using the FROM option, you can specify with <n₁> the index of the first line to be read.
- By using the TO option, you can specify with <n₂> the index of the last line to be read.
- By using the WHERE option, you can specify any logical expression as <condition> (see [Programming Logical Expressions \[Page 235\]](#)). The first operand must be a component of the internal table's line structure. You cannot use the WHERE option if you use the control keyword AT inside the loop.

The FROM and TO options restrict the number of lines which the system has to read. The WHERE option only prevents unnecessary filling of the work area. With the WHERE option, the system must read all lines. To improve performance, you should use the FROM and TO options as much as possible. It can be also beneficial under certain conditions to leave the loop with the EXIT statement (see [Terminating a Loop Entirely \[Page 259\]](#)) instead of using the WHERE option.

```
DATA: BEGIN OF LINE,
      COL1 TYPE I,
      COL2 TYPE I,
      END OF LINE.
```

```
DATA ITAB LIKE LINE OCCURS 10.
```

```
DO 30 TIMES.
```

```
  LINE-COL1 = SY-INDEX. LINE-COL2 = SY-INDEX * SY-INDEX.
```

```
APPEND LINE TO ITAB.  
ENDDO.
```

```
LOOP AT ITAB INTO LINE FROM 10 TO 25 WHERE COL2 > 400.  
  WRITE: / SY-TABIX, LINE-COL2.  
ENDLOOP.
```

The produces the following output:

21	441
22	484
23	529
24	576
25	625

Here, an internal table ITAB based on the field string LINE is created. The table is filled in a DO loop with the numbers between 1 and 30 and the squares of these numbers. The DO loop reads the table line by line. The index of the lines to be read is restricted to the numbers between 10 and 25, and the contents of the second component of each line is restricted to numbers greater than 400.

Reading Single Lines Using the Index

Reading Single Lines Using the Index

To read a single line from an internal table using its index, you use the READ statement as follows:

Syntax

```
READ TABLE <itab> [INTO <wa>] INDEX <idx>.
```

You specify the target area <wa> using the INTO option. In the case of tables with a header line, you can omit the INTO option. The table work area then becomes the target area.

The system reads the line with the index <idx> from the table <itab>. This is quicker than using the key to access the table (see [Reading Single Lines using the Key \[Page 296\]](#)).

If an entry with the specified index was found, the system field SY-SUBRC is set to 0 and SY-TABIX contains the index of that line. Otherwise, SY-SUBRC is set to a value other than 0.

If <idx> is less than or equal to 0, a runtime error occurs. If <idx> exceeds the table size, the system sets the return code value in SY-SUBRC to 4.

```
DATA: BEGIN OF ITAB OCCURS 10,
      COL1 TYPE I,
      COL2 TYPE I,
      END OF ITAB.
```

```
DO 20 TIMES.
  ITAB-COL1 = SY-INDEX.
  ITAB-COL2 = 2 * SY-INDEX.
  APPEND ITAB.
ENDDO.
```

```
READ TABLE ITAB INDEX 7.
```

```
WRITE: SY-SUBRC, SY-TABIX.
WRITE: / ITAB-COL1, ITAB-COL2.
```

The produces the following output:

```
0      7
      7      14
```

Here, an internal table ITAB is created with a header line and filled with 20 lines. The line with index 7 is read and output..

Reading Single Lines Using a Key

You can read single lines from an internal table using a key as follows:

[Reading Single Lines With User-Defined Keys \[Page 297\]](#)

[Reading Single Lines Using the Standard Key \[Page 300\]](#)

When reading single lines with keys, you can use a binary search instead of a sequential search. For further information about binary searching, see

[Binary Search \[Page 302\]](#)

Reading Single Lines With User-Defined Keys

Reading Single Lines With User-Defined Keys

To read a single line from an internal table with a user-defined key, you use the WITH KEY option of the READ statement as follows:

Syntax

```
READ TABLE <itab> [INTO <wa>] WITH KEY <key> [BINARY SEARCH].
```

You specify the target area <wa> using the INTO option. In the case of tables with a header line, you can omit the INTO option. The table work area then becomes the target area.

The system reads the first entry of <itab> which matches the key defined in <key>. For more information about the BINARY SEARCH addition, see [Binary Search \[Page 302\]](#)

If an entry with the appropriate key was found, the system field SY-SUBRC is set to 0 and SY-TABIX contains the index of that line. Otherwise, SY-SUBRC is set to a value other than 0.

You can define several keys <key> as described in the following:

Defining a Sequence of Key Fields

To define your own sequence of key fields, use the WITH KEY option as follows:

Syntax

```
....WITH KEY <k1> = <f1>... <kn> = <fn>...
```

The user-defined key consists of the table components <k₁>...<k_n>. The fields <f₁>...<f_n> are the values which the contents of the key fields must match.

If the data type of <f_i> is not compatible to the data type of <k_i>, then <f_i> is converted to the type of <k_i>.

You can set key fields at runtime by replacing <k_i> with (<n_i>). The key field is the contents of the field <n_i>. If <n_i> is blank at runtime, the system ignores this key field. If <n_i> contains an invalid component name, a runtime error occurs.

You can specify offset and length for any component of the key (see [Specifying Offset Values for Data Objects \[Page 216\]](#)).

Defining the Entire Line as Key

You can define the entire line of an internal table as its key by using the WITH KEY option as follows:

Syntax

```
....WITH KEY = <value>...
```

If the data type of <value> is not compatible to the data type of the table lines, <value> is converted to the data type of the table lines.

With such a key, you can choose a particular line also of an internal table which is defined directly by an elementary data type or an internal table, and not by a field string.

Defining the Beginning of a Line as the Key

To define the beginning of an internal table line as the key, you use the WITH KEY option as follows:

Syntax

Reading Single Lines With User-Defined Keys

....WITH KEY <k>...

The system compares the (left-justified) beginning of a line with <k>. <k> cannot contain an internal table or a structure containing an internal table. In contrast to the above two options, the comparison is processed using the data type of <k>.

```
DATA: BEGIN OF LINE,
      COL1 TYPE C,
      COL2 TYPE P DECIMALS 5,
      COL3 TYPE I,
      COL4 TYPE I,
      END OF LINE.

DATA ITAB LIKE LINE OCCURS 10.

DO 10 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SQRT( SY-INDEX).
  LINE-COL3 = SY-INDEX ** 2.
  LINE-COL4 = SY-INDEX ** 3.
  APPEND LINE TO ITAB.
ENDDO.

READ TABLE ITAB INTO LINE WITH KEY COL3 = 9 COL4 = 36.
WRITE: / SY-SUBRC, SY-TABIX.

READ TABLE ITAB INTO LINE WITH KEY COL3 = 9 COL4 = 27.
WRITE: / SY-SUBRC, SY-TABIX.

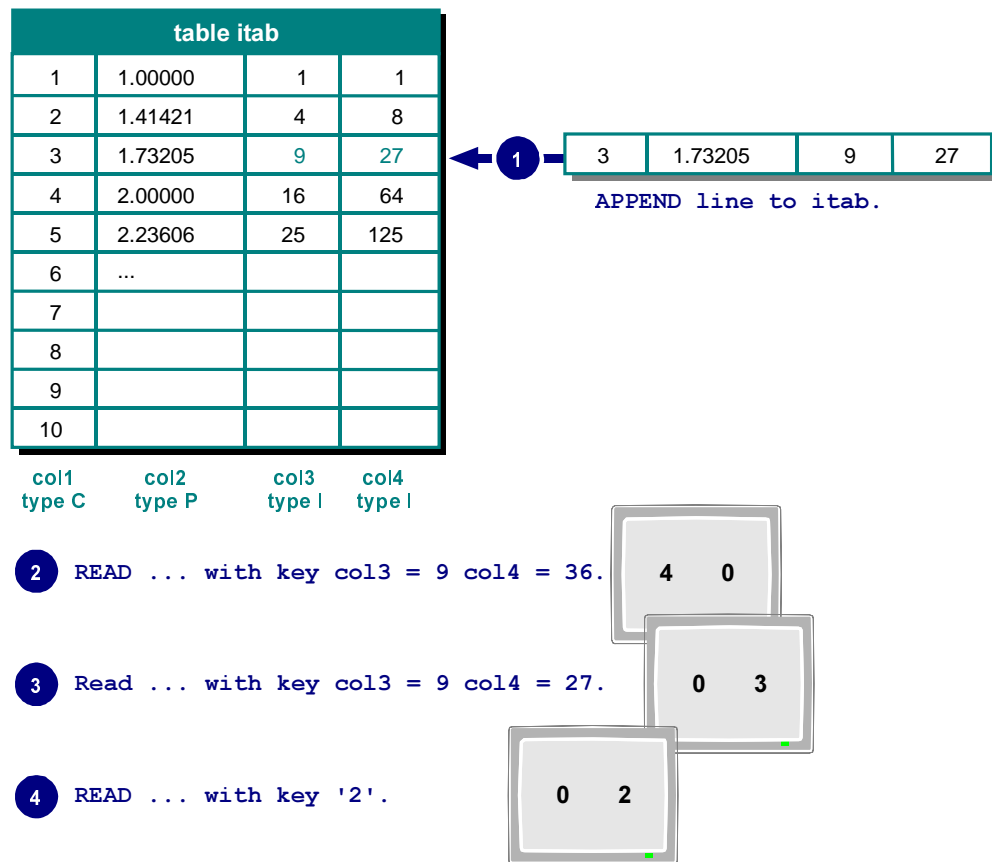
READ TABLE ITAB INTO LINE WITH KEY '2'.
WRITE: / SY-SUBRC, SY-TABIX.
```

The produces the following output:

```
4      0
0      3
0      2
```

Here, an internal table is created with four columns. After filling 10 lines of the table, single lines are read with user-defined keys. The first READ statement with the user-defined sequence of key fields COL3, COL4 fails, while the second READ statement finds the line with index 3. The third READ statement searches for a table line which starts with "2" and finds the line with index 2. The following diagram shows the main steps involved:

Reading Single Lines With User-Defined Keys



```
DATA ITAB TYPE I OCCURS 10,
DATA SQUARE TYPE I.
```

```
DO 30 TIMES.
```

```
  SQUARE = SY-INDEX ** 2.
```

```
  APPEND SQUARE TO ITAB.
```

```
ENDDO.
```

```
READ TABLE ITAB INTO SQUARE WITH KEY = 25.
```

```
WRITE: SY-SUBRC, SY-TABIX.
```

The produces the following output:

```
0      5
```

Here, an internal table is created with lines of elementary type I. After filling the table, one line with value 25 and index 5 is read.

Reading Single Lines with the Standard Key

To read the first line from an internal table with a particular standard key, you use the READ statement as follows:

Syntax

```
READ TABLE <itab> [INTO <wa>] [BINARY SEARCH].
```

You have to specify the key of the line to be read from <itab> in its table work area.

This variant of the read statement works only for internal tables with header lines.

The system searches for the first entry in the table to match all standard key fields in the table work area and reads this line into the table work area. If you use the INTO option, the line is read into the work area <wa>.

The standard key consists of all of the key fields described in [Operations on Internal Tables \[Page 266\]](#) that do not contain the values SPACE. For more information about the BINARY SEARCH addition, see [Binary Search \[Page 302\]](#)

If an entry with matching key was found, the system field SY-SUBRC is set to 0 and SY-TABIX contains the index of that line. Otherwise, SY-SUBRC is set to 4.

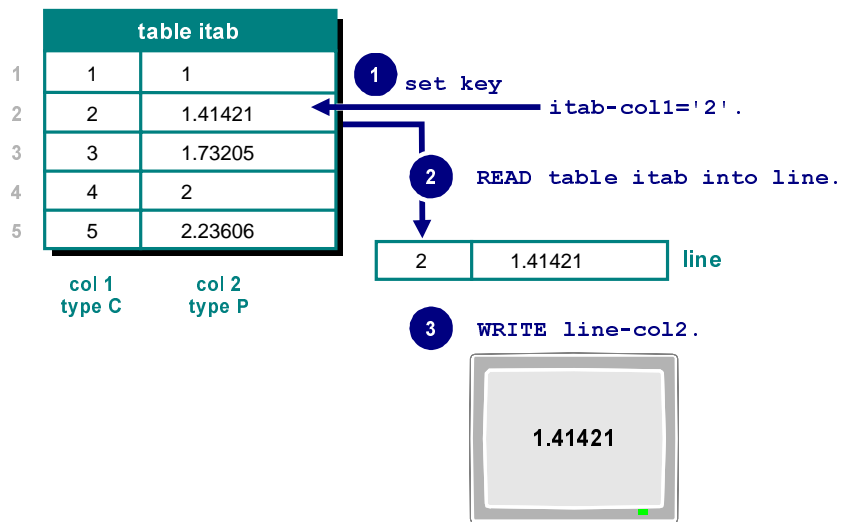
```
DATA: BEGIN OF LINE,  
      COL1 TYPE C,  
      COL2 TYPE P DECIMALS 5,  
      END OF LINE.  
  
DATA ITAB LIKE LINE OCCURS 10 WITH HEADER LINE.  
  
DO 5 TIMES.  
  LINE-COL1 = SY-INDEX. LINE-COL2 = SQRT( SY-INDEX).  
  APPEND LINE TO ITAB.  
ENDDO.  
  
ITAB-COL1 = '2'.  
  
READ TABLE ITAB INTO LINE.  
  
WRITE LINE-COL2.
```

The produces the following output:

```
1.41421
```

Here, an internal table ITAB is created with a header line and filled with square root values for the numbers between 1 and 5. The default key of this table is COL1 because it is of type C. After assigning "2" to COL1 of the table work area, the respective line of the internal table is read and the square root of 2 is output. The following diagram shows the main steps involved:

Reading Single Lines with the Standard Key



Binary Search

When using keys to read single lines, you can perform a binary search instead of the standard sequential search. To do this, use the BINARY SEARCH option of the READ statement.

Syntax

READ TABLE <itab>..... BINARY SEARCH.

If you use the BINARY SEARCH option, your internal table must be sorted in the order specified in the key.

If the system finds more than one line that matches the specified key, it reads the line with the lowest index.

Binary search is faster than linear search. Therefore, you should try to keep internal tables sorted and use the BINARY SEARCH option whenever possible.

Comparing the Contents of Single Lines

Comparing the Contents of Single Lines

To compare the contents of a single line which has been read using the READ statement with the contents of the target area, you can use the COMPARING option of the READ statement as follows:

Syntax

READ TABLE <itab> [INTO <wa>] <key-option> COMPARING <fields>.

The system reads a single line specified by a key or an index in <key option>. After reading the line, the components specified in <fields> are compared to the corresponding components of the target area. You can specify the target area <wa> by using the INTO option. In the case of tables with a header line, you can omit the INTO option. The table work area then becomes the target area.

For <fields>, you can write a list of components

... <f1>...<fn>.

or specify all components with

... ALL FIELDS.

If the system finds an entry with the specified <key-option> and if the contents of the compared fields are the same, SY-SUBRC is set to 0. If the contents of the compared fields are not the same, it returns the value 2. If the system cannot find an entry, SY-SUBRC is set to 4.

If the system finds an entry, it reads it into the target area, regardless of the result of the comparison. For more information about the READ statement, see the keyword documentation of READ.

```
DATA: BEGIN OF LINE,
      COL1 TYPE I,
      COL2 TYPE I,
      COL3 TYPE I,
      END OF LINE.

DATA ITAB LIKE LINE OCCURS 10.

DO 10 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  LINE-COL3 = SY-INDEX ** 3.
  APPEND LINE TO ITAB.
ENDDO.

LINE-COL1 = 2. LINE-COL2 = 4. LINE-COL3 = 8.

READ TABLE ITAB INTO LINE INDEX 4 COMPARING COL1 COL2.
WRITE: / SY-SUBRC, SY-TABIX.

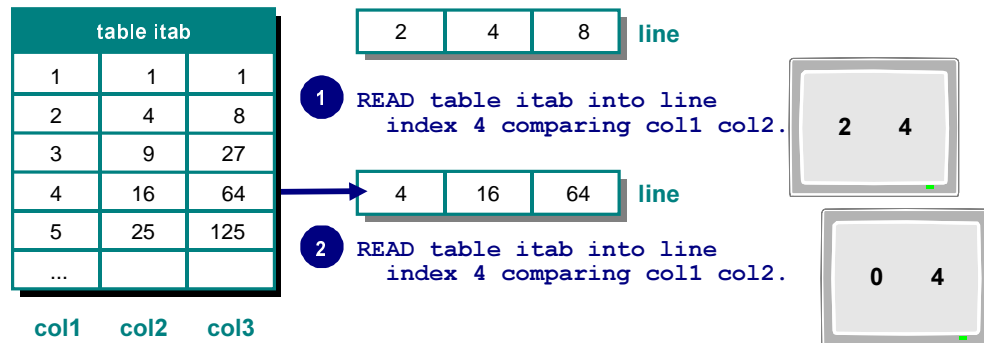
READ TABLE ITAB INTO LINE INDEX 4 COMPARING COL1 COL2.
WRITE: / SY-SUBRC, SY-TABIX.
```

The produces the following output:

```
2      4
0      4
```

Comparing the Contents of Single Lines

Here, an internal table ITAB is created and filled. After the first READ statement, SY-SUBRC is set to 2 because the line with index 4 was found, but the contents of COL1 and COL2 are not the same for the internal table and the target area LINE. However, the table line is read into LINE and SY-SUBRC returns 0 after the next READ statement. The following diagram shows the main steps involved:



Reading Parts of Single Lines

Reading Parts of Single Lines

To read parts of single lines, use the TRANSPORTING option of the READ statement as follows:

Syntax

```
READ TABLE <itab> [INTO <wa>] <key-option> TRANSPORTING <fields>.
```

The system reads a single line specified by a key or an index in <key option>. After reading the line, the components specified in <fields> are transported to the target area. You can specify the target area <wa> using the INTO option. In the case of tables with a header line, you can omit the INTO option. The table work area then becomes the target area.

For <fields>, you can write a list of components

... <f1>...<fn>.

or specify that no component should be transferred with

... NO FIELDS.

In the latter case, only the system fields SY-SUBRC and SY-TABIX are affected by the READ statement.

For more information about the READ statement, see the keyword documentation.

```
DATA: BEGIN OF LINE,
      COL1 TYPE I,
      COL2 TYPE I,
      COL3 TYPE I,
      END OF LINE.

DATA ITAB LIKE LINE OCCURS 10.

DO 10 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  LINE-COL3 = SY-INDEX ** 3.
  APPEND LINE TO ITAB.
ENDDO.

CLEAR LINE.
READ TABLE ITAB INTO LINE INDEX 5 TRANSPORTING COL2.
WRITE: / LINE-COL1, LINE-COL2, LINE-COL3.
```

The produces the following output:

```
0      25      0
```

Here, an internal table ITAB is created and filled. The READ statement reads only the COL2 field into the field string LINE.

Searching Internal Tables for Character Strings

You can search internal tables for a specific pattern with the statement SEARCH as follows:

Syntax

SEARCH <itab> FOR <str> <options>.

This statement searches the internal table <itab> for the character string <str>. If successful, the return code value of SY-SUBRC is set to 0. SY-TABIX is set to the index of the table line, in which the string was found. SY-FDPOS is set to the offset of the string in the table line. Otherwise, SY-SUBRC is set to 4.

This statement treats all operands as data type C. The system executes no type conversions.

In this statement, <itab> always denotes the actual internal table and not the header line. This is an exception to the general rule, which stipulates that statements normally interpret the table name as a table work area (see [Choosing a Table Type \[Page 267\]](#)).

The search string <str> can take the following forms (refer to [Searching for Strings \[Page 209\]](#)):

<str>	Function
<pattern>	<pattern> (any sequence of characters) is sought. Trailing blanks are ignored.
.<pattern>.	Searches for <pattern>. Trailing blanks are not ignored.
*<pattern>	A word ending with <pattern> is sought.
<pattern>*	A word starting with <pattern> is sought.

Words are separated by blanks, commas, periods, semicolons, colons, question marks, exclamation marks, parentheses, slashes, plus signs, and equal signs.

The different options (<options>) for the search in an internal table are:

- **ABBREVIATED**
Field <c> is searched for a word containing the string in <str>. The characters can be separated by other characters. The first letter of the word and the string <str> must be the same.
- **STARTING AT <lin₁>**
Searches table <itab> for <str>, starting at line <lin₁>. <lin₁> can be a variable.
- **ENDING AT <lin₂>**
Searches table <itab> for <str> up to line <lin₂>. <lin₂> can be a variable.
- **AND MARK**
If the search string is found, all the characters in the search string (and all the characters inbetween when using ABBREVIATED) are converted to upper case.

Searching Internal Tables for Character Strings

```
TYPES: BEGIN OF LINE,  
        INDEX  TYPE I,  
        TEXT(8) TYPE C,  
        END OF LINE.  
  
DATA: ITAB TYPE LINE OCCURS 10 WITH HEADER LINE,  
      NUM(2) TYPE N.  
  
DO 10 TIMES.  
  ITAB-INDEX = SY-INDEX.  
  NUM = SY-INDEX.  
  CONCATENATE 'string' NUM INTO ITAB-TEXT.  
  APPEND ITAB.  
ENDDO.  
  
SEARCH ITAB FOR 'string05' AND MARK.  
  
WRITE: / "'string05" found at line', (1) SY-TABIX,  
       'with offset', (1) SY-FDPOS.  
SKIP.  
  
READ TABLE ITAB INDEX SY-TABIX.  
WRITE: / ITAB-INDEX, ITAB-TEXT.
```

This example produces the following output:

```
'string05' found at line 5 with offset 4  
      5 STRING05
```

Note that the found string's offset is determined by the first column of the table, which is of type I with length 4.

The option AND MARK changes the table contents in the corresponding line.

Determining the Attributes of Internal Tables

If, in the course of processing, you want to find out how many lines are contained in your internal table or how large you have defined the OCCURS parameter, use the DESCRIBE statement as follows:

Syntax

```
DESCRIBE TABLE <itab> [LINES <lin>] [OCCURS <occ>].
```

If you use the LINES parameter, the number of filled lines is written to the variable <lin>. If you use the OCCURS parameter, the initial number of lines is written to the variable <occ>.

```
DATA: BEGIN OF LINE,
      COL1 TYPE I,
      COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE LINE OCCURS 10.

DATA: LIN TYPE I, OCC TYPE I.

DESCRIBE TABLE ITAB LINES LIN OCCURS OCC.
WRITE: / LIN, OCC.

DO 1000 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

DESCRIBE TABLE ITAB LINES LIN OCCURS OCC.
WRITE: / LIN, OCC.
```

The produces the following output:

0	10
1,000	10

Here, an internal table ITAB is created. The DESCRIBE statement is processed before and after the table is filled. The current number of lines changes, but the number of initial lines cannot change.

Changing and Deleting Lines of Internal Tables

Changing and Deleting Lines of Internal Tables

To modify the contents of filled internal tables, you can do the following

Changing lines

[Changing Lines with MODIFY \[Page 310\]](#)

[Changing Lines with WRITE TO \[Page 313\]](#)

Deleting lines

[Deleting Lines in a Loop \[Page 314\]](#)

[Deleting Lines Using the Index \[Page 315\]](#)

[Deleting Adjacent Duplicate Entries \[Page 316\]](#)

[Deleting Selected Lines \[Page 318\]](#)

Changing Lines with MODIFY

To change lines with the MODIFY statement use:

Syntax

```
MODIFY <itab> [FROM <wa>] [INDEX <idx>]
      [TRANSPORTING <f1>... <fn> [WHERE <condition>]].
```

The work area <wa> specified in the FROM option replaces a line in <itab>. In the case of tables with a header line, you can omit the FROM option: The table work area then replaces the line.

If you use the INDEX option, the new line replaces the existing line with index <idx>. If the replacement is successful, SY-SUBRC is set to 0. If the internal table contains fewer lines than <idx>, no line is changed and SY-SUBRC contains 4.

If you use the MODIFY statement without the INDEX option, the system can process it only within a LOOP - ENDLOOP block by changing the current line (i.e. the line which has an index returned by SY-TABIX).

With the TRANSPORTING option, the system transports only the components <f₁> to <f_n> from the work area into the internal table <itab>. You can specify the components dynamically by writing (<f_i>). In this case, the system reads the name of the component from field <f_i> at runtime. An invalid component name leads to a runtime error.

For tables with a complex line structure, the usage of the transporting option results in better performance, if the system must not transport unnecessary table-like components.

You can specify a WHERE condition with the TRANSPORTING option that determines the lines of the table into which the components <f₁> to <f_n> are to be transported. You can use any logical expression for <condition>, where the first operand is a component of the internal table's line structure.

You cannot use the WHERE condition together with the INDEX option.

```
DATA: BEGIN OF LINE,
      COL1 TYPE I,
      COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE LINE OCCURS 10.

DO 3 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

LOOP AT ITAB INTO LINE.
  IF SY-TABIX = 2.
    LINE-COL1 = SY-TABIX * 10.
    LINE-COL2 = ( SY-TABIX * 10 ) ** 2.
    MODIFY ITAB FROM LINE.
```

Changing Lines with MODIFY

```

ENDIF.
ENDLOOP.

LOOP AT ITAB INTO LINE.
  WRITE: / SY-TABIX, LINE-COL1, LINE-COL2.
ENDLOOP.

```

The produces the following output:

1	3	5
2	20	400
3	3	9

Here, an internal table ITAB is created and filled with three lines. Inside the first LOOP - ENDLOOP block, the second line is replaced by the contents of the field string LINE.

```

DATA: BEGIN OF LINE,
      COL1 TYPE I,
      COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE LINE OCCURS 10.

DO 3 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

LINE-COL1 = 10. LINE-COL2 = 10 ** 2.

MODIFY ITAB FROM LINE INDEX 2.

LOOP AT ITAB INTO LINE.
  WRITE: / SY-TABIX, LINE-COL1, LINE-COL2.
ENDLOOP.

```

The produces the following output:

1	1	1
2	10	100
3	3	9

Here, an internal table ITAB is created and filled with three lines. The second line is replaced by the contents of the field string LINE.

```

DATA NAME(4) VALUE 'COL2'.

DATA: BEGIN OF LINE,
      COL1 TYPE I,
      COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE LINE OCCURS 10.

```

```
DO 5 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

LINE-COL2 = 111.
MODIFY ITAB FROM LINE INDEX 2 TRANSPORTING (NAME).

LINE-COL1 = 11.
MODIFY ITAB FROM LINE TRANSPORTING COL1 WHERE COL2 = 25.

LOOP AT ITAB INTO LINE.
  WRITE: / SY-TABIX, LINE-COL1, LINE-COL2.
ENDLOOP.
```

The produces the following output:

1	1	1
2	2	111
3	3	9
4	4	16
5	11	25

This example creates an internal table ITAB filled with five lines. The components COL2 in the second line and COL1 in the fifth line are replaced. Note the dynamic specification of COL2 in the first MODIFY statement.

Changing Lines with WRITE TO

Changing Lines with WRITE TO

To change lines with the WRITE TO statement, use the following syntax:

Syntax

WRITE <f>[+<o1>][(<l1>)] TO <itab>[+<o2>][(<l2>)] INDEX <idx>.

The contents of the section with offset <o1> and length <l1> in field <f> are copied to the table line with index <idx>, where they **overwrite** the section with offset <o2> and length <l2>. Note that, even in the case of tables with a header line, the WRITE TO statement with the INDEX option does not access the table work area, but a line of the table.

This is a variant of the WRITE TO statement, which is described in the section [Assigning Values using Offset Specification \[Page 176\]](#)

The WRITE TO statement does not recognize the structure of table lines. SAP recommends that you use this statement only if you want (for example) to convert a flag whose exact position you know. Another possibility would be internal tables defined with one elementary character field. Tables with this structure are, for example, important when you generate programs dynamically (see [Generating and Starting Programs Dynamically \[Page 505\]](#)).

```
DATA CODE(72) OCCURS 10 WITH HEADER LINE.
```

```
CODE = 'This is the first line.'.
APPEND CODE.
```

```
CODE = 'This is the second line. It is ugly.'.
APPEND CODE.
```

```
CODE = 'This is the third and final line.'.
APPEND CODE.
```

```
WRITE 'nice.' TO CODE+31 INDEX 2.
```

```
LOOP AT CODE.
  WRITE / CODE.
ENDLOOP.
```

The produces the following output:

This is the first line.

This is the second line. It is nice.

This is the third and final line.

Here, an internal table CODE is defined with an elementary type C field which is 72 characters long. After filling the table with three lines, the second line is changed by using the WRITE TO statement. The word "ugly" is replaced by the word "nice".

Deleting Lines in a Loop

To delete lines from an internal table in a loop, use the DELETE statement as follows:

Syntax

DELETE <itab>.

The system can only process this statement in a LOOP... ENDLOOP block (see [Reading Internal Tables Line by Line \[Page 293\]](#)). It deletes the current line (i.e. the line which has the index returned by SY-TABIX).

After the first line has been deleted, the current line and its assignment to the contents of SY-TABIX can be undefined. To process further lines within this loop, use only statements with the INDEX option.

```
DATA: BEGIN OF LINE,
      COL1 TYPE I,
      COL2 TYPE I,
      END OF LINE.
DATA ITAB LIKE LINE OCCURS 10.
DO 30 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.
LOOP AT ITAB INTO LINE.
  IF LINE-COL1 < 28.
    DELETE ITAB.
  ENDIF.
ENDLOOP.
LOOP AT ITAB INTO LINE.
  WRITE: / SY-TABIX, LINE-COL1, LINE-COL2.
ENDLOOP.
```

The produces the following output:

1	28	784
2	29	841
3	30	900

Here, an internal table ITAB is created and filled with 30 lines. Inside the first LOOP - ENDLOOP block, all lines in the COL1 field that have any entry less than 28 are deleted.

Deleting Lines Using the Index

Deleting Lines Using the Index

To delete lines using the index, you use the INDEX option with the DELETE statement as follows:

Syntax

```
DELETE <itab> INDEX <idx>.
```

If you use the INDEX option, the line with index <idx> is deleted from ITAB. After deleting the line, the index of the following lines is decremented by 1.

If the operation is successful, SY-SUBRC is set to 0. Otherwise, if no line with index <idx> exists, SY-SUBRC contains 4.

If you delete an entry within a LOOP - ENDLOOP block, the current line and its assignment to the contents of SY-TABIX can become undefined. To process further lines within this loop, use only statements with the INDEX option.

```
DATA: BEGIN OF LINE,
      COL1 TYPE I,
      COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE LINE OCCURS 10.

DO 5 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

DELETE ITAB INDEX: 2, 3, 4.

WRITE: 'SY-SUBRC', SY-SUBRC.

LOOP AT ITAB INTO LINE.
  WRITE: / SY-TABIX, LINE-COL1, LINE-COL2.
ENDLOOP.
```

The produces the following output:

```
SY-SUBRC    4

      1      1      1
      2      3      9
      3      5     25
```

This example creates an internal table ITAB filled with five lines. Then a chain statement is processed to delete three lines with the indexes 2, 3, and 4. After deleting the line with index 2, the index of the following lines is decremented by one. Therefore, the next deletion removes the line with an index which was initially 4. Then, the third deletion fails because the table now consists of only 3 lines and SY-SUBRC returns 4.

Deleting Adjacent Duplicate Entries

To delete adjacent duplicate entries, use the DELETE statement as follows:

Syntax

DELETE ADJACENT DUPLICATE ENTRIES FROM <itab> [COMPARING <comp>].

The system deletes all adjacent duplicate entries from the internal table <itab>.

Entries are duplicate if they fulfill one of the following compare criteria:

- If you do not use the COMPARING addition, the contents of the standard key fields must be identical (see [Operations on Internal Tables \[Page 266\]](#)).
- With the COMPARING option
.... COMPARING <f₁> <f₂>...,
the contents of the specified fields <f₁> <f₂>... must be the same. You can also specify the field names at runtime in parentheses by writing (<name>) instead of <f₁>. The field <name> contains the name of the sort key field. If <name> is empty at runtime, the system ignores it. If it contains an invalid component name, a runtime error occurs.
- With the COMPARING option
.... COMPARING ALL FIELDS,
the contents of all fields must be the same.

If the system finds and deletes at least one duplicate entry, SY-SUBRC is set to 0. Otherwise, it is set to 4.

You can use this statement to delete **all** duplicate entries from an internal table if the table is sorted by the specified compare criterion.

```
DATA: BEGIN OF LINE,
      COL1 TYPE I,
      COL2 TYPE C,
      END OF LINE.

DATA ITAB LIKE LINE OCCURS 10.

LINE-COL1 = 1. LINE-COL2 = 'A'. APPEND LINE TO ITAB.
LINE-COL1 = 1. LINE-COL2 = 'A'. APPEND LINE TO ITAB.
LINE-COL1 = 1. LINE-COL2 = 'B'. APPEND LINE TO ITAB.
LINE-COL1 = 2. LINE-COL2 = 'B'. APPEND LINE TO ITAB.
LINE-COL1 = 3. LINE-COL2 = 'B'. APPEND LINE TO ITAB.
LINE-COL1 = 4. LINE-COL2 = 'B'. APPEND LINE TO ITAB.
LINE-COL1 = 5. LINE-COL2 = 'A'. APPEND LINE TO ITAB.

LOOP AT ITAB INTO LINE.
  WRITE: / LINE-COL1, LINE-COL2.
ENDLOOP.

DELETE ADJACENT DUPLICATES FROM ITAB COMPARING ALL FIELDS.
```

Deleting Adjacent Duplicate Entries

```
SKIP TO LINE 3.  
LOOP AT ITAB INTO LINE.  
  WRITE: /14 LINE-COL1, LINE-COL2.  
ENDLOOP.  
  
DELETE ADJACENT DUPLICATES FROM ITAB COMPARING COL1.  
  
SKIP TO LINE 3.  
LOOP AT ITAB INTO LINE.  
  WRITE: /28 LINE-COL1, LINE-COL2.  
ENDLOOP.  
  
DELETE ADJACENT DUPLICATES FROM ITAB.  
  
SKIP TO LINE 3.  
LOOP AT ITAB INTO LINE.  
  WRITE: /42 LINE-COL1, LINE-COL2.  
ENDLOOP.
```

The produces the following output:

1 A	1 A	1 A	1 A
1 A	1 B	2 B	2 B
1 B	2 B	3 B	5 A
2 B	3 B	4 B	
3 B	4 B	5 A	
4 B	5 A		
5 A			

Here, the first DELETE statement deletes the second line from ITAB because the second line has the same contents as the first line. The second DELETE statement deletes the second line from the remaining table because the contents of the field COL1 is the same as in the first line. The third DELETE statement deletes the third and fourth line from the remaining table because the contents of the default key field COL2 are the same as on the second line. Although the contents of the default key are the same for the first and the fifth line, the fifth line is not deleted because it is not adjacent to the first line.

Deleting Selected Lines

To delete a set of selected lines, use the DELETE statement as follows:

Syntax

DELETE <itab> [FROM <n₁>] [TO <n₂>] [WHERE <condition>].

You must specify at least one of the three options. If you use this statement without the WHERE option, the system deletes all lines from <itab> that have an index between <n₁> and <n₂>. If you do not use the FROM option, the system starts deleting from the first line. If you do not use the TO option, the system deletes all lines until the last line.

If you use the WHERE option, the system deletes only those lines from <itab> which satisfy the condition <condition>. For <condition>, use any logical expression described in [Programming Logical Expressions \[Page 235\]](#). The first operand must be a component of the internal table's line structure.

If the system deletes at least one line, SY-SUBRC is set to 0. Otherwise, it is set to 4.

```
DATA: BEGIN OF LINE,
      COL1 TYPE I,
      COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE LINE OCCURS 10.

DO 40 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

DELETE ITAB FROM 3 TO 38 WHERE COL2 > 20.

LOOP AT ITAB INTO LINE.
  WRITE: / LINE-COL1, LINE-COL2.
ENDLOOP.
```

The produces the following output:

1	1
2	4
3	9
4	16
39	1.521
40	1.600

Here, the system deletes all entries between line 3 and line 39 in ITAB if the values in COL2 are greater than 20.

Sorting Internal Tables

Sorting Internal Tables

To sort an internal table, use the SORT statement as follows:

Syntax

```
SORT <itab> [<order>] [AS TEXT]
  [BY <f1> [<order>] [AS TEXT]]... <fn> [<order>] [AS TEXT]].
```

With the SORT statement, you can sort an internal table its standard key or by self defined keys.

Sorting by Standard Key

If you do not use the BY option, the internal table <itab> is sorted by its standard key (see [Operations on Internal Tables \[Page 266\]](#)). The sort order depends from the sequence of the standard key fields in the internal table.

If possible, define the standard-key fields at the beginning of the internal table lines to get the best performance for sorting the table.

```
DATA: BEGIN OF ITAB OCCURS 10,
      COL1 TYPE C,
      COL2 TYPE I,
      COL3 TYPE N,
      END OF ITAB.
```

.....

```
SORT ITAB.
```

In this example, the internal table is sorted by its standard key or, in other words, by all non-numerical fields. The sort order is first COL1 and then COL3. The performance of the program can be improved by exchanging columns two and three.

Sorting by User-defined Key

To define a different sort key, use the BY option. The system then sorts the dataset according to the specified components <f₁>... <f_n>. These fields can be of any type, including type P, I, and F fields, or tables. The number of key fields is limited to 250. If you specify more than one key field, the system sorts the records first by <f₁>, then by <f₂>, and so on. The system uses the options you specify before BY as a default for all fields specified behind BY. The options that you specify behind individual fields overwrite for these fields the options specified before BY.

If a sort criterion is not known until runtime, you can set it dynamically by writing (<name>) instead of <f₁>. The field <name> contains the name of the sort key field. If <name> is empty at runtime, the system ignores it. If it contains an invalid component name, a runtime error occurs. You can specify offset and length for any component of the sort key (see [Specifying Offset Values for Data Objects \[Page 216\]](#)).

You can specify the sorting sequence by using DESCENDING or ASCENDING in the <order> option. The default is ascending.

You can influence the sorting method for character fields with the option AS TEXT. Without the option AS TEXT, the system sorts character fields binarily and according to their platform-

dependent internal coding. With the option AS TEXT, the system sorts character fields alphabetically according to the current text environment. By default, the text environment is set in the user's master record. As an exception, you can set the text environment with the statement SET LOCALE LANGUAGE. The option AS TEXT frees you from converting a character field into a sortable format before sorting (see [Converting to a Sortable Format \[Page 207\]](#)). Such a conversion is only necessary if you want to

- sort an internal table alphabetically first and search it binarily afterwards (see [Binary Search \[Page 302\]](#)). The order of an internal table after alphabetical sorting differs from the order after binary sorting.
- sort an internal table several times with alphabetical keys. In this case, the performance is better, because the conversion is processed only once.
- create alphabetical indexes for database tables in your program.

If you prefix BY with AS TEXT, the option only applies to type C fields of the sort key. If you place AS TEXT behind a field, the field must be of type C.

If you specify the sort key yourself, you can improve performance by keeping the key relatively short. However, if the sort key contains an internal table, the sorting process may be slowed down considerably.

Sorting is not stable. This means that the old sequence of lines with the same sort key may not necessarily be retained.

If the main storage space available is not enough for sorting the data, the system writes data to an external help file during the sorting process. This help file is determined by the SAP profile parameter DIR_SORTTMP.

```
DATA: BEGIN OF ITAB OCCURS 10,
      LAND(3) TYPE C,
      NAME(10) TYPE C,
      AGE TYPE I,
      WEIGHT TYPE P DECIMALS 2,
      END OF ITAB.

ITAB-LAND = 'USA'. ITAB-NAME = 'Nancy'.
ITAB-AGE = 35. ITAB-WEIGHT = '45.00'.
APPEND ITAB.

ITAB-LAND = 'USA'. ITAB-NAME = 'Howard'.
ITAB-AGE = 40. ITAB-WEIGHT = '95.00'.
APPEND ITAB.

ITAB-LAND = 'GB'. ITAB-NAME = 'Jenny'.
ITAB-AGE = 18. ITAB-WEIGHT = '50.00'.
APPEND ITAB.

ITAB-LAND = 'F'. ITAB-NAME = 'Michele'.
ITAB-AGE = 30. ITAB-WEIGHT = '60.00'.
APPEND ITAB.
```


Sorting Internal Tables

```

ITAB-LAND = 'G'.  ITAB-NAME = 'Karl'.
ITAB-AGE = 60.   ITAB-WEIGHT = '75.00'.
APPEND ITAB.

SORT ITAB.

LOOP AT ITAB.
  WRITE: / ITAB-LAND, ITAB-NAME, ITAB-AGE, ITAB-WEIGHT.
ENDLOOP.

SKIP.

SORT ITAB DESCENDING BY LAND WEIGHT ASCENDING.

LOOP AT ITAB.
  WRITE: / ITAB-LAND, ITAB-NAME, ITAB-AGE, ITAB-WEIGHT.
ENDLOOP.

```

The produces the following output:

F	Michele	30	60.00
G	Karl	60	75.00
GB	Jenny	18	50.00
USA	Howard	40	95.00
USA	Nancy	35	45.00
USA	Nancy	35	45.00
USA	Howard	40	95.00
GB	Jenny	18	50.00
G	Karl	60	75.00
F	Michele	30	60.00

Here, an internal table ITAB is created with a header line and filled with 5 lines. First, it is sorted by its standard key, which is LAND and NAME. Then, it is sorted by a sort key defined as LAND and WEIGHT. The general sort order is defined as descending, but for WEIGHT it is defined as ascending. This is why the line with the NAME field "NANCY" is output before the line with the NAME field "HOWARD" after the second SORT statement.

```

DATA: BEGIN OF ITAB OCCURS 10,
      TEXT(6),
      XTEXT(160) TYPE X,
      END OF ITAB.

```

```

ITAB-TEXT = 'Muller'.
CONVERT TEXT ITAB-TEXT INTO SORTABLE CODE ITAB-XTEXT.
APPEND ITAB.

```

```

ITAB-TEXT = 'Möller'.
CONVERT TEXT ITAB-TEXT INTO SORTABLE CODE ITAB-XTEXT.
APPEND ITAB.

```

```

ITAB-TEXT = 'Moller'.
CONVERT TEXT ITAB-TEXT INTO SORTABLE CODE ITAB-XTEXT.
APPEND ITAB.

```

```

ITAB-TEXT = 'Miller'.
CONVERT TEXT ITAB-TEXT INTO SORTABLE CODE ITAB-XTEXT.
APPEND ITAB.

```

```
SORT ITAB BY TEXT.  
LOOP AT ITAB.  
  WRITE / ITAB-TEXT.  
ENDLOOP.  
SKIP.  
  
SORT ITAB BY XTEXT.  
LOOP AT ITAB.  
  WRITE / ITAB-TEXT.  
ENDLOOP.  
SKIP.  
  
SORT ITAB AS TEXT BY TEXT.  
LOOP AT ITAB.  
  WRITE / ITAB-TEXT.  
ENDLOOP.
```

This example demonstrates alphabetical sorting of character fields. The internal table ITAB contains a column with character fields and a column with corresponding binary codes that are alphabetically sortable. The binary codes are created with the CONVERT statement (see [Converting to a Sortable Format \[Page 207\]](#)). The table is sorted three times. First, it is sorted binarily by the TEXT field. Second, it is sorted binarily by the XTEXT field. Third, it is sorted alphabetically by the TEXT field. If using a German text environment, the output looks as follows:

```
Miller  
Moller  
Muller  
Möller  
  
Miller  
Moller  
Möller  
Muller  
  
Miller  
Moller  
Möller  
Muller
```

After the first sorting, 'Möller' follows behind 'Muller' since the internal coding for the letter 'ö' comes behind the coding of 'u'. The other two sortings are alphabetical. The binary sorting by XTEXT has the same result as the alphabetical sorting by field TEXT.

To create a ranked list, you can also use the SORTED BY option of the APPEND statement instead of the SORT statement (see [Creating Ranked Lists \[Page 323\]](#)). For examples of how to use SORT to sort data from database tables, see [Formatting Data Using Internal Tables \[Page 911\]](#).

Creating Ranked Lists

Creating Ranked Lists

Internal tables are suitable for generating ranked lists. To do this, you start with an empty internal table and use the SORTED BY option of the APPEND statement as follows:

Syntax

APPEND [<wa> TO] <itab> SORTED BY <f>.

If you use the SORTED BY option with the APPEND statement (see [Appending Lines \[Page 279\]](#)), the new line is not appended as the last line of the internal table <itab>. Instead, the system inserts the new line so that the internal table <itab> is sorted in descending order by the field <f>.

To generate ranked lists containing up to 100 entries, you should use the APPEND statement. When dealing with larger lists, it is advisable to sort tables with the SORT statement for performance reasons (see [Sorting Internal Tables \[Page 319\]](#)).

If you use the SORTED BY option, the table can contain only the number of lines specified in the OCCURS parameter. This is an exception to the general rule (see [Creating Internal Table Data Types \[Page 271\]](#)). If you add more lines than specified, the last line is discarded. This is useful for creating ranked lists of limited length (for example "Top Ten").

The work area you specify when using the SORTED BY option of the APPEND statement must be compatible with the line type of the internal table. Convertibility is not sufficient for this option.

```
DATA: BEGIN OF ITAB OCCURS 2,
      COLUMN1 TYPE I,
      COLUMN2 TYPE I,
      COLUMN3 TYPE I,
      END OF ITAB.

ITAB-COLUMN1 = 1. ITAB-COLUMN2 = 2. ITAB-COLUMN3 = 3.
APPEND ITAB SORTED BY COLUMN2.

ITAB-COLUMN1 = 4. ITAB-COLUMN2 = 5. ITAB-COLUMN3 = 6.
APPEND ITAB SORTED BY COLUMN2.

ITAB-COLUMN1 = 7. ITAB-COLUMN2 = 8. ITAB-COLUMN3 = 9.
APPEND ITAB SORTED BY COLUMN2.

LOOP AT ITAB.
  WRITE: / ITAB-COLUMN2.
ENDLOOP.
```

The produces the following output:

```
8
5
```

Here, three lines are appended to an internal table with a header line using the SORTED BY option. Note that the line with the smallest contents of the COLUMN2 field is discarded because the number of lines is specified as 2 in the OCCURS parameter.

Loop Processing

You use the LOOP statement to read lines from an internal table into a work area (see [Reading Internal Tables Line by Line \[Page 293\]](#)). The following sections describe

[Calculating Totals \[Page 325\]](#)

[Using Control Levels for Groups of Lines \[Page 327\]](#)

Calculating Totals

Calculating Totals

To add up the contents of numeric fields in an internal table during loop processing, use the SUM statement as follows:

Syntax

SUM.

The system can process this statement only within a LOOP - ENDLOOP block.

If you use SUM in an AT - ENDAT block, the system calculates totals for the numeric fields of **all** lines in the current line group and writes them to the corresponding fields in the work area (see example in [Using Control Levels for Groups of Lines \[Page 327\]](#)).

If you use the SUM statement outside an AT - ENDAT block (see [Using Control Levels for Groups of Lines \[Page 327\]](#)), the system calculates totals for the numeric fields of **all** lines of the internal table in **each** loop pass and writes them to the corresponding fields of the work area (see example).

Therefore, you should use the SUM statement **only** in AT - ENDAT blocks.

If a component of an internal table line is another table, do not use the SUM statement.

This example shows how the SUM statement works outside an AT - ENDAT block. It should not be used in this way.

```
DATA: BEGIN OF LINE,
      COL1 TYPE I,
      COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE LINE OCCURS 10.

DO 3 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

LOOP AT ITAB INTO LINE.
  WRITE: / LINE-COL1, LINE-COL2.
  SUM.
  WRITE: / LINE-COL1, LINE-COL2.
ENDLOOP.
```

The produces the following output:

1	1
6	14
2	4
6	14
3	9

6 14

Here, an internal table ITAB is created and filled with three lines. In the LOOP - ENDLOOP block, the contents of the work area LINE are output before and after the SUM statement. The total for all lines is calculated within each loop pass.

Using Control Levels for Groups of Lines

Using Control Levels for Groups of Lines

This topic describes how to use control level statements to create statement blocks which process only specific table lines inside the LOOP - ENDLOOP block. You can thus use the SUM statement to calculate totals from subsets of all lines (see also [Calculating Totals \[Page 325\]](#)).

You open such a statement block with the control level statement AT and close it with the control level statement ENDAT.

Syntax

AT <line>.

 <statement block>

ENDAT.

The line condition <line>, at which the statement block within AT - ENDAT is processed, can be:

<line>	Meaning
FIRST	First line of the internal table
LAST	Last line of the internal table
NEW <f>	Beginning of a group of lines with the same contents in the field <f> and in the fields left of <f>
END Of <f>	End of a group of lines with the same contents in the field <f> and in the fields left of <f>

Statement blocks in AT - ENDAT blocks which use these line conditions represent predefined control structures. You can use them to react to control breaks in internal tables instead of programming them yourself with control statements and logical expressions.

To work with control levels in internal tables, you must pay attention to the following conditions:

- The sequence of the control levels in internal tables is static.
The occurrence of control breaks depends on the sequence of the columns. The first column defines the highest control level and so on. You must know the control level hierarchy already when defining the internal table.
- Sorting of the internal table by these columns.
Before working with control breaks, you should sort the internal exactly in the same order as its columns are defined. This means, sort first by column one, second by column two and so on. If you sort by the standard key (see [Sorting Internal Tables \[Page 319\]](#)) and all standard key fields (the non-numerical fields) are placed at the beginning of the table lines, the sort order is automatically suited for working with control breaks.
- Hierarchy of the AT-ENDAT statement blocks.
Within the LOOP-ENDLOOP loop, you must order the AT-ENDAT statement blocks according to the hierarchy of the control levels. If the

Using Control Levels for Groups of Lines

internal table has the columns <col1>, <col2>,....., and if it is sorted by these columns, you must program the loop as follows:

```

LOOP AT <itab>.
  AT FIRST.... ENDAT.
  AT NEW <col1>..... ENDAT.
  AT NEW <col2>..... ENDAT.
  .....
  <single line processing>
  .....
  AT END OF <col2>.... ENDAT.
  AT END OF <col1>.... ENDAT.
  AT LAST..... ENDAT.
ENDLOOP.

```

As the innermost step, you can place statements for table lines that do not create control breaks. You need not to use all line conditions. But you must place the used ones in the above sequence.

If a control break criterion <f> is not known until runtime, you can set it dynamically by writing (<name>) instead of <f>. The field <name> contains the name of the field <f>. If <name> is empty at runtime, the system ignores the control break criterion. If it contains an invalid component name, a runtime error occurs.

You can restrict group change criteria (both static and dynamic) by specifying offset and length (see [Specifying Offset Values for Data Objects \[Page 216\]](#)). You can also use field symbols for <f>. If a field symbol does not point to a valid component, a runtime error occurs.

In an AT - ENDAT statement block, the work area is not filled by the current table line. All fields which are not part of the standard key (see [Using Internal Tables \[Page 264\]](#)) are initialized. For the line-conditions FIRST and LAST, the system overwrites all standard key fields with asterisks (*). For the line-conditions NEW <f> and END OF <f>, the system overwrites all standard key fields which appear in the work area to the right of the specified field <f> with asterisks (*). You can fill the work area in the AT - ENDAT statement block according to your requirements.

Do not use control level statements in loops where the effect of the LOOP statement is restricted by FROM, TO or WHERE (see [Reading Internal Tables Line by Line \[Page 293\]](#)) because the interaction is then not well defined.

```

DATA: BEGIN OF LINE,
      COL1 TYPE C,
      COL2 TYPE I,
      COL3 TYPE I,
      END OF LINE.

DATA ITAB LIKE LINE OCCURS 10.

LINE-COL1 = 'A'.
DO 3 TIMES.
  LINE-COL2 = SY-INDEX.
  LINE-COL3 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

```


Using Control Levels for Groups of Lines

```

LINE-COL1 = 'B'.
DO 3 TIMES.
  LINE-COL2 = 2 * SY-INDEX.
  LINE-COL3 = ( 2 * SY-INDEX) ** 2.
  APPEND LINE TO ITAB.
ENDDO.

LOOP AT ITAB INTO LINE.
WRITE: / LINE-COL1, LINE-COL2, LINE-COL3.

  AT END OF COL1.
    SUM.
    ULINE.
    WRITE: / LINE-COL1, LINE-COL2, LINE-COL3.
    SKIP.
  ENDAT.

  AT LAST.
    SUM.
    ULINE.
    WRITE: / LINE-COL1, LINE-COL2, LINE-COL3.
  ENDAT.
ENDLOOP.

```

The produces the following output:

A	1	1
A	2	4
A	3	9

A	6	14
---	---	----

B	2	4
B	4	16
B	6	36

B	12	56
---	----	----

*	18	70
---	----	----

Here, an internal table ITAB is created and filled with six lines. In the LOOP - ENDLOOP block, the work area LINE is output for each loop pass. The total for all numeric fields is always calculated when the contents of COL1 change and when the system is in the last loop pass. Compare this example with the one in the section [Calculating Totals \[Page 325\]](#).

The following example shows how you can use control levels in report programs. You find more information about the SELECT statement in [Formatting Data \[Page 905\]](#).

Using Control Levels for Groups of Lines

```
TABLES SBOOK.

DATA: BEGIN OF ITAB OCCURS 10,
      CARRID LIKE SBOOK-CARRID,
      CONNID LIKE SBOOK-CONNID,
      FLDATE LIKE SBOOK-FLDATE,
      CUSTTYPE LIKE SBOOK-CUSTTYPE,
      CLASS LIKE SBOOK-CLASS,
      BOOKID LIKE SBOOK-BOOKID,
      END OF ITAB.

SELECT CARRID CONNID FLDATE CUSTTYPE CLASS BOOKID
      FROM SBOOK INTO CORRESPONDING FIELDS OF TABLE ITAB.

SORT ITAB.

LOOP AT ITAB.

  AT FIRST.
    WRITE / 'List of Bookings'.
    ULINE.
  ENDAT.

  AT NEW CARRID.
    WRITE: / 'Carrid:', ITAB-CARRID.
  ENDAT.

  AT NEW CONNID.
    WRITE: / 'Connid:', ITAB-CONNID.
  ENDAT.

  AT NEW FLDATE.
    WRITE: / 'Fdate:', ITAB-FLDATE.
  ENDAT.

  AT NEW CUSTTYPE.
    WRITE: / 'Custtype:', ITAB-CUSTTYPE.
  ENDAT.

    WRITE: / ITAB-BOOKID, ITAB-CLASS.

  AT END OF CLASS.
    ULINE.
  ENDAT.

ENDLOOP.
```

In this example, an internal table ITAB is filled with data from the database table SBOOK in a SELECT statement. The sequence of internal table columns defines the hierarchy of control levels. Since the table contains only non-numeric fields, the sequence of columns defines also the sort order for sorting by the standard key. Please note the sequence of the AT-ENDAT blocks within the LOOP and ENDLOOP statements.

The output looks as follows:

```
List of Bookings
Carrid: AA
Connid: 0017
Fdate: 1996/03/24
Custtype: B
```

Using Control Levels for Groups of Lines

00063509 C

00063517 C

...

00063532 F

00063535 F

...

Custtype: P

00063653 C

00063654 C

...

00063668 F

00063670 F

...

Fldate: 1996/05/06

Custtype: B

00064120 C

00064121 C

...

and so on.

Comparing Internal Tables

You can use internal tables as operands of logical expressions (see also [Programming Logical Expressions \[Page 235\]](#)):

Syntax

.... <itab1> <operator> <itab2>...

For <operator>, all operators listed in the table in [Comparisons with All Field Types \[Page 236\]](#) can be used (EQ, =, NE, <>, ><, GE, >=, LE, <=, GT, >, LT, <).

The first criterion for comparing internal tables is the number of lines they contain. The more lines an internal table contains, the larger it is. If two internal tables contain the same number of lines, they are compared line by line, component by component. If components of the table lines are themselves internal tables, they are compared recursively. If you use operators other than the equality operator, the system stops the comparison as soon as it finds a pair of components which are not equal and returns the result.

In the case of internal tables with a header line, you can use square brackets ([]) after the table name to distinguish the table body from the table work area.

```
DATA: BEGIN OF LINE,
      COL1 TYPE I,
      COL2 TYPE I,
      END OF LINE.

DATA: ITAB LIKE LINE OCCURS 10,
      JTAB LIKE LINE OCCURS 10.

DO 3 TIMES.
  LINE-COL1 = SY-INDEX.
  LINE-COL2 = SY-INDEX ** 2.
  APPEND LINE TO ITAB.
ENDDO.

MOVE ITAB TO JTAB.

LINE-COL1 = 10. LINE-COL2 = 20.
APPEND LINE TO ITAB.

IF ITAB GT JTAB.
  WRITE / 'ITAB GT JTAB'.
ENDIF.

APPEND LINE TO JTAB.

IF ITAB EQ JTAB.
  WRITE / 'ITAB EQ JTAB'.
ENDIF.

LINE-COL1 = 30. LINE-COL2 = 80.
APPEND LINE TO ITAB.

IF JTAB LE ITAB.
  WRITE / 'JTAB LE ITAB'.
ENDIF.
```

Comparing Internal Tables

```
LINE-COL1 = 50. LINE-COL2 = 60.  
APPEND LINE TO JTAB.  
  
IF ITAB NE JTAB.  
  WRITE / 'ITAB NE JTAB'.  
ENDIF.  
  
IF ITAB LT JTAB.  
  WRITE / 'ITAB LT JTAB'.  
ENDIF.
```

The produces the following output:

```
ITAB GT JTAB  
ITAB EQ JTAB  
JTAB LE ITAB  
ITAB NE JTAB  
ITAB LT JTAB
```

Here, two internal tables, ITAB and JTAB, are created. ITAB is filled with 3 lines and copied to JTAB. Then, another line is appended to ITAB and the first logical expression tests whether ITAB is greater than JTAB. After appending the same line to JTAB, the second logical expression tests whether both tables are equal. Then, another line is appended to ITAB and the third logical expressions tests whether JTAB is less than or equal to ITAB. Then, a line is appended to JTAB, where the contents of the components are not equal to the contents of the components in the last line of ITAB. The next logical expressions test whether ITAB is not equal to JTAB. The first component where a difference is found between ITAB and JTAB is COL1 in the last table line. It is 30 for ITAB and 50 for JTAB. Therefore, in the last logical expression, ITAB is less than JTAB.

Initializing Internal Tables

To initialize an internal table with or without a header line, you use the REFRESH statement as follows:

Syntax

REFRESH <itab>.

This statement resets an internal table to the state before it was filled. This means that the table contains no lines.

If you are working with an internal table without table work area, you can use the CLEAR statement instead of the REFRESH statement as follows:

Syntax

CLEAR <itab>.

If you are working with an internal table with a header line, the CLEAR statement clears only the table work area as explained in [Resetting Values to Initial Values \[Page 184\]](#). To reset the whole internal table without clearing the table work area, use either the REFRESH statement or the CLEAR statement as follows:

Syntax

CLEAR <itab>[].

The square brackets after the name of the internal table refer to the body of the internal table.

After using REFRESH or CLEAR to initialize an internal table, the system keeps the space in memory reserved. You can release the memory with the FREE statement as follows:

Syntax

FREE <itab>.

You can also use the FREE statement to reset an internal table and to release its memory directly, without using REFRESH or CLEAR beforehand. Like REFRESH, FREE works on the table body, not on the table work area.

After a FREE statement, you can address the internal table again. The system then reserves memory space again.

You can check whether an internal table is empty by using the following logical expression:

Syntax

... <itab> IS INITIAL...

(see [Checking for the Initial Value \[Page 244\]](#)).

```
DATA: BEGIN OF LINE,
      COL1,
      COL2,
      END OF LINE.

DATA ITAB LIKE LINE OCCURS 10.

LINE-COL1 = 'A'. LINE-COL2 = 'B'.
APPEND LINE TO ITAB.
```

Initializing Internal Tables

```
REFRESH ITAB.  
IF ITAB IS INITIAL.  
  WRITE 'ITAB is empty'.  
  FREE ITAB.  
ENDIF.
```

The produces the following output:

ITAB is empty.

In this program, an internal table ITAB is filled and then initialized with REFRESH. In an IF statement, a logical expression with the IS INITIAL parameter is used to check whether ITAB is empty. If so, the memory is released.

Working with Field Symbols

In ABAP programs, field symbols are placeholders for existing fields. A field symbol does not physically reserve space for a field, but points to a field which is not known until runtime of the program. Field symbols are comparable to the concept of pointers as used in the programming language C (i.e. pointers to which the contents operator * is applied). In ABAP, however, there is no real equivalent to pointers in the sense of variables which contain a memory address and can be used without the contents operator. You can only work with the data object to which a field symbol points.

In this section you learn about:

[Field Symbols - Concept \[Page 337\]](#)

[Defining Field Symbols \[Page 338\]](#)

[Assigning Data Objects to Field Symbols \[Page 346\]](#)

[Runtime Checks \[Page 360\]](#)

Field Symbols - Concept

Sometimes you only know which field you want to process, and how you want to process it, at runtime.

For this purpose, you can create field symbols in your program. At runtime, you can assign real fields to such field symbols. All operations which you have programmed with the field symbol are then carried out with the assigned field. After successful assignment, there is no difference in ABAP whether you reference the field symbol or the field itself.

Field symbols can point to any data object in ABAP and to structures defined in the ABAP Dictionary.

You can create field symbols either without or with type specifications. In the first case, the field symbol adopts all the attributes of the assigned field. In the second case, the system checks during the assignment process whether the assigned field matches the type of the field symbol. Whichever applies, you must first assign a field to your field symbol before you can work with it in your program.

Field symbols provide some features that make them very flexible:

- You can specify the offset and length of the assigned field as variables.
- You can assign field symbols to other field symbols and even specify offset and length there.
- Assignments to field symbols may extend beyond field boundaries. This allows effective access to regularly stored data.
- You can force a field symbol to be of a different type and to have a different number of decimal places than the assigned field.
- Field symbols may have a structure that you can use to point to individual components of structures.

The flexibility of field symbols provides elegant solutions to certain problems. However, the same flexibility makes them tricky to use. Since you can assign data objects to field symbols which may not be known until runtime, the effectiveness of syntax and security checks is very limited for operations involving field symbols. This can lead to runtime errors or incorrect data assignments.

While runtime errors indicate an obvious problem, incorrect data assignments are dangerous because they can be very difficult to detect. Therefore, you should use field symbols only in cases where you are absolutely sure of what you are doing, or if there are no other ABAP statements you can use to solve your problem.

If, for example, you are processing character strings, you may want to process only part of a string where the position and length depend on its contents. You can do this by using field symbols. However, since Release 3.0 of the R/3 system, you can also use the MOVE statement with variable offset and length specifications (see [Assigning Values with Offset Specifications \[Page 176\]](#)). The use of the MOVE statement (possibly in conjunction with some auxiliary variables) is safer than working with field symbols. The advantage of field symbols is that they can improve response times in some cases.

Defining Field Symbols

You can define field symbols for any internal data objects

[Defining Field Symbols for Internal Fields \[Page 339\]](#)

You can define structured field symbols for internal and external structures

[Defining Structured Field Symbols \[Page 343\]](#)

You can use field symbols locally in subroutines and function modules.

[Defining Local Field Symbols \[Page 345\]](#)

Defining Field Symbols for Internal Fields

Defining Field Symbols for Internal Fields

To define a field symbol for an internal data object, use the FIELD-SYMBOLS statement as follows:

Syntax

FIELD-SYMBOLS <FS> [<type>].

This statement defines a field symbol <FS>.

For field symbols, the angle brackets are part of the syntax. They identify field symbols in the program code.

You can define field symbols without or with type specifications.

[Field Symbols Without Type Specifications \[Page 340\]](#)

[Typing Field Symbols \[Page 341\]](#)

Field Symbols Without Type Specifications

To define a field symbol without type specification, you do not use the <type> option of the FIELD-SYMBOLS statement:

Syntax

FIELD-SYMBOLS <FS>.

The field symbol <FS> does not have any attributes. You can assign any data object to it at runtime (see [Assigning Data Objects to Field Symbols \[Page 346\]](#)). During the assignment process, the field symbol inherits all the attributes of the data object. The data type of the assigned data object becomes the actual data type of the field symbol.

Typing Field Symbols

Typing Field Symbols

You can type a field symbol using the <type> option of the FIELD-SYMBOLS statement as follows:

Syntax

FIELD-SYMBOLS <FS> <type>.

For the <type>, you can write either TYPE <t> or LIKE <f> (see [The Basic Form of the DATA Statement \[Page 120\]](#)).

When you assign a data object to the typed field symbol <FS> without making a type specification, the system checks whether the type of the assigned data object is compatible with the typing of the field symbol. The compatibility rules are set out in the table below. If types are incompatible, the system outputs an error message during the syntax check if possible, or reacts with a runtime error.

On the other hand, no error occurs if you want the field symbol to retain its specified type, regardless of the assigned data object. Here, you must assign a data object with a type specification to the typed field symbol. The type specified during the assignment must then be compatible with the typing of the field symbol (see [Determining the Data Type of a Field Symbol \[Page 356\]](#)).

The following compatibility rules apply for the typing of field symbols:

Typing	Syntax check for field assignment
no type specified TYPE ANY	The system accepts any field type. All attributes of the field are assigned to the field symbol.
TYPE TABLE	The system checks whether the field is an internal table. All attributes and the structure of the table are assigned to the field symbol.
TYPE C, N, P, or X	The system checks whether the field is of type C, N, P, or X. The field symbol inherits the length of the field and (for type P) its DECIMALS specification.
TYPE D, F, I, or T LIKE <f>, TYPE <ud>	These types are fully specified. The system checks whether the data type of the field agrees completely with the type of the field symbol.

(<ud> is user-defined)

You use the option <type> to perform the same checks as for the formal parameters of subroutines. (see [Typing Formal Parameters \[Page 461\]](#)).

```
DATA DAT(8) VALUE '09161995'.
```

```
FIELD-SYMBOLS <FS> TYPE D.
```

```
ASSIGN DAT TO <FS>.
```

```
WRITE <FS>.
```

This program results in an error during the syntax check of the ASSIGN statement because DAT is incompatible with the typing of the field symbol <FS>.

```
DATA DAT(8) VALUE '19951609'.
```

```
FIELD-SYMBOLS <FS> TYPE D.
```

```
ASSIGN DAT TO <FS> TYPE 'D'.
```

```
WRITE <FS>.
```

In this program, the ASSIGN statement specifies the same type as in the typing of <FS>. Therefore, the program is executable and produces the following output:

```
09161995
```

For further information about the ASSIGN statement, see [Assigning Data Objects to Field Symbols \[Page 346\]](#).

Defining Structured Field Symbols

Defining Structured Field Symbols

You can define a structured field symbol by using the FIELD-SYMBOLS statement as follows:

Syntax

FIELD-SYMBOLS <FS> STRUCTURE <s> DEFAULT <f>.

This statement defines a structured field symbol <FS> which points initially to the field <f>. You must assign an initial field to the structured field symbol, but you can change this assignment later (see [Assigning Data Objects to Field Symbols \[Page 346\]](#)). The field symbol <FS> inherits the structure of <s>. The structure <s> can be any field string or a structure defined in the ABAP Dictionary. You do not have to declare the dictionary structure with the TABLES statement. You must specify the structure <s> by entering its name without quotation marks. Specification at runtime is not allowed.

If <s> contains no components of type I or type F, <f> can be any internal field with at least the same length as the structure <s>. If <f> is shorter than <s>, an error occurs during the syntax check. If you assign another field to <FS> later in the program, the system also checks whether the field to be assigned is long enough. If a shorter field is assigned dynamically at runtime (see [Dynamic ASSIGN \[Page 351\]](#)), a runtime error occurs.

If <s> contains components of type I or type F, please note that this structure is aligned (see [Alignment of Data Objects \[Page 233\]](#)). If you assign a data object to a field symbol with such an aligned structure, the data object must also be aligned accordingly. Otherwise a runtime error occurs. In such cases, you are recommended to assign such data objects only to structured field symbols which retain the same structure as the field symbol at least over the length of the structure.

After defining a structured field symbol, you can reference the individual components of the structure symbolically. To do this, prefix the name of a component field with the name of the field symbol separated with a hyphen.

You can use structured field symbols, for example, for creating one work area for different structures or several work areas of the same structure.

```
DATA: WA(100) VALUE '001LH 123419950627'.
DATA: BEGIN OF LINE1,
      COL1(6),
      COL2(4),
      COL3(8),
      END OF LINE1.
DATA: BEGIN OF LINE2,
      COL1 TYPE I VALUE 1,
      COL2 TYPE I VALUE 2,
      END OF LINE2.
DATA LINE3 LIKE LINE2.
DATA ITAB LIKE LINE2 OCCURS 10 WITH HEADER LINE.
LINE3-COL1 = 11. LINE3-COL2 = 22.
FIELD-SYMBOLS: <F1> STRUCTURE SBOOK DEFAULT WA,
               <F2> STRUCTURE LINE1 DEFAULT WA,
               <F3> STRUCTURE ITAB DEFAULT LINE3.
```

Defining Structured Field Symbols

```
WRITE: / <F1>-MANDT, <F1>-CARRID, <F1>-CONNID, <F1>-FLDATE,  
      / <F2>-COL1, <F2>-COL2, <F2>-COL3.  
      / <F3>-COL1, <F3>-COL2.
```

The produces the following output:

```
001 LH 1234 27.06.1995  
001LH 1234 19950627  
      11      22
```

This example defines three field strings LINE1, LINE2, and LINE3, as well as an internal table ITAB (with header line). It then defines three field symbols <F1>, <F2>, and <F3> as follows:

- <F1> has the same structure as the ABAP Dictionary structure SBOOK and points to the character string WA.
- <F2> has the same structure as LINE1 and also points to WA.
- <F3> has the same structure as the header line ITAB and points to LINE3. <F3> cannot point to WA, because the structure of ITAB is aligned.

When you output the field symbols to the screen, the contents of the assigned fields are formatted according to the type of the field symbol. Note, for example, that <F1>-FLDATE has the data type D.

Defining Local Field Symbols

Defining Local Field Symbols

You can define local field symbols in subroutines and function modules in the same way that you can define local data types and objects (see [Defining Local Data Types and Objects in Subroutines \[Page 469\]](#)).

Any field symbols you declare in subroutines or function modules are local. For local field symbols, the following rules apply:

- You cannot reference local field symbols outside the subroutine or function module.
- Each time the subroutine or function module is called, **no** field is assigned to the local field symbol, even if an ASSIGN was executed the previous time.
- Local field symbols can have the same names as globally defined field symbols or local field symbols in other subroutines or function modules. Local field symbols hide global field symbols.
- Structured field symbols can be local as well. They can have local (internal) structures and you can assign local fields to them.

Assigning Data Objects to Field Symbols

You must assign a data object to a field symbol before you can work with it. With structured field symbols, you must include this assignment in the definition (see [Defining Structured Field Symbols \[Page 343\]](#)). In the case of non-structured field symbols, you are free to decide where and when to assign a data object for the first time. In the course of the program, you can assign different data objects to the same field symbol, regardless of whether it is structured or not.

To assign a data object to a field symbol, you use the ASSIGN statement. The ASSIGN statement has several variants and parameters. The subsequent topics describe

[The Basic Form of the ASSIGN Statement \[Page 347\]](#)

[Assigning Field Symbols to Other Field Symbols \[Page 354\]](#)

[Assigning Components of Structures \[Page 355\]](#)

[Defining the Data Type of a Field Symbol \[Page 356\]](#)

[Changing the Number of Decimal Places \[Page 358\]](#)

[Assigning a Local Copy of a Global Field \[Page 359\]](#)

Basic Form of the ASSIGN Statement

The basic forms of the ASSIGN statement comprise two static variants and two dynamic variants.

[Static ASSIGN \[Page 348\]](#)

[Static ASSIGN with Offset Specification \[Page 349\]](#)

[Dynamic ASSIGN \[Page 351\]](#)

[Dynamic ASSIGN of Table Work Areas \[Page 353\]](#)

Static ASSIGN

If you know the name of the data object you want to assign to a field symbol before runtime, use the ASSIGN statement as follows:

Syntax

ASSIGN <f> TO <FS>.

After the assignment, the field symbol <FS> has the attributes of the data object <f> and points to the same memory area.

```
FIELD-SYMBOLS: <F1>, <F2> TYPE I.
DATA: TEXT(20) TYPE C VALUE 'Hello, how are you?',
      NUM      TYPE I VALUE 5,
      BEGIN OF LINE1,
        COL1 TYPE F VALUE '1.1e+10',
        COL2 TYPE I VALUE '1234',
      END OF LINE1,
      LINE2 LIKE LINE1.
ASSIGN TEXT TO <F1>.
ASSIGN NUM TO <F2>.
DESCRIBE FIELD <F1> LENGTH <F2>.
WRITE: / <F1>, 'has length', NUM.

ASSIGN LINE1 TO <F1>.
ASSIGN LINE2-COL2 TO <F2>.
MOVE <F1> TO LINE2.
ASSIGN 'LINE2-COL2 =' TO <F1>.
WRITE: / <F1>, <F2>.
```

The produces the following output:

```
Hello, how are you? has length      20
LINE-COL2 =      1,234
```

This example defines two field symbols <F1> and <F2>. <F2> can point only to type I fields, because its type is specified as I. In the course of the example, <F1> and <F2> point to several different data objects.

Static ASSIGN with Offset Specifications

Static ASSIGN with Offset Specifications

You can specify an offset value for a field to be assigned to a field symbol by using the ASSIGN statement as follows:

Syntax:

ASSIGN <f>[+<o>][(<l>)] TO <FS>.

As described in [Specifying Offset Values for Data Objects \[Page 216\]](#), the part of <f> which has the offset <o> and the length <l> is assigned to the field symbol <FS>. The following special features apply for the offset specification in the ASSIGN statement:

- <o> and <l> can be variables.
- The system does not check whether the selected part lies inside the field <f>. Both offset <o> and length <l> can be larger than the length of <f>. You can refer to addresses beyond the limits of <f>, but not beyond the defined memory area (see [Runtime Checks \[Page 360\]](#)).
- If you do not specify a length <l>, the system automatically inserts the length of the field <f>. If then <o> is greater than zero, <FS> always points to an area beyond the limits of <f>.
- If <o> is smaller than the length of <f>, you can specify an asterisk (*) for <l> to prevent <FS> from referring to an address beyond the limits of <f>.

FIELD-SYMBOLS <FS>.

DATA: BEGIN OF LINE,
 STRING1(10) VALUE '0123456789',
 STRING2(10) VALUE 'abcdefghij',
 END OF LINE.

WRITE / LINE-STRING1+5.

ASSIGN LINE-STRING1+5 TO <FS>.
 WRITE / <FS>.

ASSIGN LINE-STRING1+5(*) TO <FS>.
 WRITE / <FS>.

The produces the following output:

56789

56789abcde

56789

In this example, you can see the difference between an offset specification in a WRITE statement and an offset specification in an ASSIGN statement. With WRITE, the output is truncated at the end of LINE-STRING1. Specifying an offset greater than 9 would lead to an error during the syntax check. In the first ASSIGN statement, the memory area of length 10 beginning with offset 5 in LINE-STRING1 is assigned to the field symbol <FS>. This results in meaningful output because the memory area behind LINE-STRING1 is clearly defined in the

Static ASSIGN with Offset Specifications

program. In the second ASSIGN statement, the assignment of memory behind the boundary of LINE-STRING1 is prevented.

```
FIELD-SYMBOLS <FS>.  
DATA: BEGIN OF LINE,  
      A VALUE '1', B VALUE '2', C VALUE '3', D VALUE '4',  
      E VALUE '5', F VALUE '6', G VALUE '7', H VALUE '8',  
      END OF LINE,  
      OFF TYPE I,  
      LEN TYPE I VALUE 2.  
DO 2 TIMES.  
  OFF = SY-INDEX * 3.  
  ASSIGN LINE-A+OFF(LEN) TO <FS>.  
  <FS> = 'XX'.  
ENDDO.  
DO 8 TIMES.  
  OFF = SY-INDEX - 1.  
  ASSIGN LINE-A+OFF(1) TO <FS>.  
  WRITE <FS>.  
ENDDO.
```

The produces the following output:

```
1 2 3 X X 6 X X
```

In this example, you can see how field symbols facilitate the access to and manipulation of regular field strings. Note, however, that this flexible method of manipulating memory contents beyond field limits also has its dangers and may lead to runtime errors.

Dynamic ASSIGN

Dynamic ASSIGN

If you know the name of the data object you want to assign to a field symbol only at runtime, use the ASSIGN statement as follows:

Syntax

ASSIGN (<f>) TO <FS>.

This statement assigns the field whose name is contained in <f> to the field symbol <FS>. You cannot specify an offset for (<f>).

At runtime, the system searches for the named field in the following sequence:

1. If the assignment is performed in a subroutine or function module, the system searches for the field in the subroutine or function module among the local data.
2. If the assignment is performed outside a subroutine or function module, or the field is not found there, the system searches for the field among the global data of the program.
3. If the field is not found among the global data, the system searches for it in the table work areas declared with the TABLES statement in the main program of the current program group. A program group consists of a main program and all programs containing definitions of the external subroutines called by the main program (including nested ones).

Searching in this way leads to slower response times. You should only use dynamic ASSIGNS when absolutely necessary. If you know before runtime that you want to assign only table work areas, you can use the variant of the ASSIGN statement described in [Dynamic ASSIGN for Table Work Areas \[Page 353\]](#).

If the search is successful and a field can be assigned to the field symbol, SY-SUBRC is set to 0. Otherwise, it returns 4. For security reasons, you should always check the value of SY-SUBRC after a dynamic ASSIGN to prevent the field symbol pointing to an undefined area.

Suppose the main program SAPMZTST is as follows:

```
PROGRAM SAPMZTST.  
TABLES SBOOK.  
SBOOK-FLDATE = SY-DATUM.  
PERFORM FORM1(MYFORMS1).
```

Suppose also that there are two programs called MYFORMS1 and MYFORMS2:

```
PROGRAM MYFORMS1.  
FORM FORM1.  
  PERFORM FORM2(MYFORMS2).  
ENDFORM.
```

and

```
PROGRAM MYFORMS2.  
FORM FORM2.  
  DATA NAME(20) VALUE 'SBOOK-FLDATE'.  
  FIELD-SYMBOLS <FS>.  
  ASSIGN (NAME) TO <FS>.
```

```
IF SY-SUBRC EQ 0.  
  WRITE / <FS>.  
ENDIF.  
ENDFORM.
```

After executing SAPMZTST, the output could look as follows:

08.02.1995

In this example, the program group consists of SAPMZTST, MYFORMS1, and MYFORMS2. The field symbol <FS> is defined in MYFORMS2. After the dynamic ASSIGN statement, it points to the component FLDATE of the table work area SBOOK declared in the main program SAPMZTST. This is not possible with a static ASSIGN statement. In other words, you cannot replace (NAME) by SBOOK-FLDATE because this results in an error during the syntax check.

Dynamic ASSIGN of Table Work Areas

If you know before runtime that you want to assign a table work area to a field symbol, but you do not know the name of the table area until runtime, use the following variant of the dynamic ASSIGN statement:

Syntax

ASSIGN TABLE FIELD (<f>) TO <FS>.

The system searches the data object you want to assign to the field symbol only among the table work areas declared by the TABLES statement in the main program of the current program group. In other words, the system performs only step 3 of the search described in [Dynamic ASSIGN \[Page 351\]](#).

If the search is successful and a field can be assigned to the field symbol, SY-SUBRC is set to 0. Otherwise, it returns 4.

```
TABLES SBOOK.  
DATA: NAME1(20) VALUE 'SBOOK-FLDATE',  
      NAME2(20) VALUE 'NAME1'.  
FIELD-SYMBOLS <FS>.  
ASSIGN TABLE FIELD (NAME1) TO <FS>.  
WRITE: / 'SY-SUBRC:', SY-SUBRC.  
ASSIGN TABLE FIELD (NAME2) TO <FS>.  
WRITE: / 'SY-SUBRC:', SY-SUBRC.
```

The produces the following output:

```
SY-SUBRC:    0  
SY-SUBRC:    4
```

In the first ASSIGN statement, the system finds the component FLDATE of the table work area SBOOK and SY-SUBRC is set to 0. In the second ASSIGN statement, the system does not find the field NAME1 because it is declared by the DATA statement and not by the TABLES statement. In this case, SY-SUBRC returns 4.

Assigning Field Symbols to Other Field Symbols

Instead of using the names of data objects, you can also assign field symbols to field symbols in all variants of the ASSIGN statement. To do this, code the static ASSIGN as follows:

Syntax

ASSIGN <FS1>[+<o>][(<l>)] TO <FS2>.

The dynamic ASSIGN could be coded in the following way:

Syntax

ASSIGN [TABLE FIELD] (<f>) TO <FS2>.

The field <f> contains the name of field symbol <FS1>.

<FS1> and <FS2> can be identical.

```
DATA: BEGIN OF S,
      A VALUE '1', B VALUE '2', C VALUE '3', D VALUE '4',
      E VALUE '5', F VALUE '6', G VALUE '7', H VALUE '8',
      END OF S.

DATA OFF TYPE I.
FIELD-SYMBOLS <FS>.
ASSIGN S-A TO <FS>.
DO 4 TIMES.
  OFF = SY-INDEX - 1.
  ASSIGN <FS>+OFF(1) TO <FS>.
  WRITE <FS>.
ENDDO.
```

The produces the following output:

```
1 2 4 7
```

In this example, eight fields are created as the components S-A to S-H of the field string S and filled with "1" to "8". These character strings are stored regularly in memory. The component S-A is assigned initially to the field symbol <FS>. The statements in the loop have the following effect:

Loop pass 1:
<FS> points to S-A, OFF is zero, and S-A is assigned to <FS>

Loop pass 2:
<FS> points to S-A, OFF is one, and S-B is assigned to <FS>

Loop pass 3:
<FS> points to S-B, OFF is two, and S-D is assigned to <FS>

Loop pass 4:
<FS> points to S-D, OFF is three, and S-G is assigned to <FS>

Assigning Components of Field Strings

Assigning Components of Field Strings

You can assign specific components of field strings to field symbols by using the ASSIGN statement as follows:

Syntax

ASSIGN COMPONENT <comp> OF STRUCTURE <s> TO <FS>.

The system assigns the component <comp> of structure <s> to the field symbol <FS>. You can specify <comp> either as a literal or as a variable. If <comp> is of type C or a structure which has no internal tables as components, it specifies the name of the component. If <comp> has any other elementary data type, it is converted to type I (see [Type Conversions \[Page 218\]](#)) and specifies the number of the component.

In the assignment is successful, SY-SUBRC is set to 0. Otherwise, it returns 4.

```
DATA: BEGIN OF LINE,
      COL1 TYPE I VALUE '11',
      COL2 TYPE I VALUE '22',
      COL3 TYPE I VALUE '33',
      END OF LINE.

DATA COMP(5) VALUE 'COL3'.
FIELD-SYMBOLS: <F1>, <F2>, <F3>.

ASSIGN LINE TO <F1>.
ASSIGN COMP TO <F2>.

DO 3 TIMES.
  ASSIGN COMPONENT SY-INDEX OF STRUCTURE <F1> TO <F3>.
  WRITE <F3>.
ENDDO.

ASSIGN COMPONENT <F2> OF STRUCTURE <F1> TO <F3>.
WRITE / <F3>.
```

The produces the following output:

```
11      22      33
33
```

In this example, <F1> points to the field string LINE and <F2> points to the field COMP. In the loop, the components of LINE are specified by their numbers and assigned one by one to <F3>. After the loop, the component COL3 of LINE is specified by its name and assigned to <F3>.

Defining the Data Type of a Field Symbol

You can define the data type of a field symbol by using the TYPE option of the ASSIGN statement as follows:

Syntax

ASSIGN..... TO <FS> TYPE <t>.

You can use the TYPE option with all variants of the ASSIGN statement.

When assigning a data object to a field symbol with a type specification, you must distinguish between the following:

- Untyped field symbols

With TYPE, an untyped field symbol <FS> does not inherit the data type and the output length of the assigned data object, but the data type specified in <t>. You can use any predefined data type for <t> (see [Predefined Elementary Data Types \[Page 106\]](#)). For further information about the output length of elementary data types, see [The WRITE Statement \[Page 891\]](#). The type <t> can be a literal or a variable.

- Typed field symbols

Using the TYPE options with a typed field symbol makes sense if the data type of the data object to be assigned is incompatible with the typing of the field symbol, but you want to avoid the resulting error message. In this case, the specified type <t> and the typing of the field symbol must be compatible. The field symbol then retains its data type, regardless of the assigned data object. For further information and an example, see [Typing Field Symbols \[Page 341\]](#).

If the field symbol is used after the assignment in the program, the assigned data object is not converted to the specified type <t>. Therefore, the contents of the data object should be interpretable as a field of the type <t>.

The system reacts with a runtime error if

- the specified data type is unknown,
- the length of the specified data type is incompatible with the type of the assigned field,
- an alignment error occurs (see [Aligning Data Objects \[Page 233\]](#)).

```
DATA TXT(8) VALUE '19950606'.
DATA MYTYPE(1) VALUE 'X'.
FIELD-SYMBOLS <FS>.
ASSIGN TXT TO <FS>.
WRITE / <FS>.

ASSIGN TXT TO <FS> TYPE 'D'.
WRITE / <FS>.

ASSIGN TXT TO <FS> TYPE MYTYPE.
WRITE / <FS>.
```

This produces the following output:

Defining the Data Type of a Field Symbol

19950606

06061995

3139393530363036

In this example, the character string TXT is assigned to <FS> three times. The first time the type of <FS> remains unchanged, the second time the type of <FS> is changed to type D, the third time the type of <FS> is changed to type X. Note that, when considered in pairs, the numbers in the last output line represent the hexadecimal code of the characters in TXT. The format of the second output line depends on the user's master record and that of the third output line on the character representation used by the operating system (ASCII in this case).

Changing the Number of Decimal Places

You can change the number of decimal places for field symbols which point to type P fields during the assignment. To do this, use the DECIMALS option of the ASSIGN statement as follows:

Syntax

ASSIGN..... TO <FS> DECIMALS <d>.

You can use the DECIMALS option with all variants of the ASSIGN statement. With the DECIMALS option, the number of decimal places of the field symbol <FS> becomes <d>. In general, this leads to different numerical values for the field symbol and the assigned field.

<d> can be a literal or a variable. A runtime error occurs, if <d> does not fall between 0 and 14, or if the assigned field is not of type P.

```
DATA: PACK1 TYPE P DECIMALS 2 VALUE '400',  
      PACK2 TYPE P DECIMALS 2,  
      PACK3 TYPE P DECIMALS 2.
```

```
FIELD-SYMBOLS: <F1>, <F2>.
```

```
WRITE: / 'PACK1', PACK1.
```

```
ASSIGN PACK1 TO <F1> DECIMALS 1.
```

```
WRITE: / '<F1> ', <F1>.
```

```
PACK2 = <F1>.
```

```
WRITE: / 'PACK2', PACK2.
```

```
ASSIGN PACK2 TO <F2> DECIMALS 4.
```

```
WRITE: / '<F2> ', <F2>.
```

```
PACK3 = <F1> + <F2>.
```

```
WRITE: / 'PACK3', PACK3.
```

```
<F2> = '1234.56789'.
```

```
WRITE: / '<F2> ', <F2>.
```

```
WRITE: / 'PACK2', PACK2.
```

The produces the following output:

```
PACK1      400.00
```

```
<F1>      4,000.0
```

```
PACK2      4,000.00
```

```
<F2>      40.0000
```

```
PACK3      4,040.00
```

```
<F2>      1,234.5679
```

```
PACK2      123,456.79
```

In this example, all type P fields have two decimal places. The field symbols <F1> and <F2> have one and four decimal places respectively. Note that the numeric values are different for the field symbols and for their assigned fields.

Assigning a Local Copy of a Global Field

Assigning a Local Copy of a Global Field

When working with [subroutines \[Page 444\]](#), you can create local copies of global data on a data stack. To do this, use the ASSIGN statement as follows:

Syntax

ASSIGN LOCAL COPY OF..... TO <FS>.

The system places a copy of the specified global data on the stack. In the subroutine, you can access and change this copy without changing the global data by addressing the field symbol <FS>.

You can use this option with all variants of the ASSIGN statement except for the one described in [Assigning Components of Structures \[Page 355\]](#).

Assume a main program SAPMZTST as follows:

```
PROGRAM SAPMZTST.  
DATA: BEGIN OF COMMON PART,  
      TEXT(5) VALUE 'Text1',  
      END OF COMMON PART.  
PERFORM ROUTINE(FORMPOOL).  
WRITE TEXT.
```

Assume a program FORMPOOL containing the subroutine ROUTINE as follows:

```
PROGRAM FORMPOOL.  
DATA: BEGIN OF COMMON PART,  
      TEXT(5) VALUE 'Text1',  
      END OF COMMON PART.  
FORM ROUTINE.  
  FIELD-SYMBOLS <FS>.  
  ASSIGN LOCAL COPY OF TEXT TO <FS>.  
  WRITE <FS>.  
  <FS> = 'Text2'.  
  WRITE <FS>.  
  ASSIGN TEXT TO <FS>.  
  WRITE <FS>.  
  <FS> = 'Text3'.  
ENDFORM.
```

SAPMZTST then produces the following output:

```
Text1 Text2 Text1 Text3
```

In this example, the field TEXT is declared as a common data area in the main program SAPMZTST and in the program FORMPOOL. By assigning TEXT to <FS> in the subroutine ROUTINE in the program FORMPOOL, you place a copy of the field TEXT on the local data stack. By addressing <FS>, you can read and change this copy. The global field TEXT is not affected by operations on the local field.

Runtime Checks

At runtime, the system performs certain checks to prevent any uncontrolled loss of data in memory resulting from erroneous assignments outside the defined data areas.

The following data areas are defined:

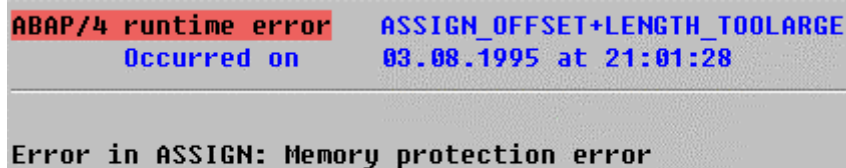
- The table storage area for internal tables. The size of this storage area depends on the number of table lines which is not fixed, but often extended dynamically at runtime (see [Creating and Processing Internal Tables \[Page 260\]](#)).
- The DATA storage area for other data objects. The size of this storage area is fixed during the data declaration (see [Declaring Data \[Page 103\]](#)).

Field symbols cannot point to addresses outside these areas. If the system encounters this at runtime, it stops processing the program.

Certain system information, such as the control blocks of internal tables, is also stored in the DATA storage area. Therefore, despite the runtime checks, you may unintentionally overwrite some of these fields and cause subsequent errors (e.g. destruction of the table header).

```
PROGRAM SAPMZTST.  
DATA: TEXT1(10), TEXT2(10), TEXT3(5).  
FIELD-SYMBOLS <FS>.  
DO 25 TIMES.  
  ASSIGN TEXT1+SY-INDEX(1) TO <FS>.  
ENDDO.
```

After starting SAPMZTST, a runtime error occurs. The system reacts as follows:



```
ABAP/4 runtime error    ASSIGN_OFFSET+LENGTH_TOOLARGE  
Occurred on           03.08.1995 at 21:01:28  
  
Error in ASSIGN: Memory protection error
```

The DATA storage area has a width of 25. The last loop pass attempts to assign an address outside this because the offset here is 25. Up to the 24th loop pass, no error occurs. If you tried to replace TEXT1 by TEXT2 in the ASSIGN statement, the error would occur in the 15th loop pass.

Saving and Reading Data

Storing Data Objects as Clusters

You can group any complex internal data objects of an ABAP program together in data clusters and store them temporarily in ABAP memory or for longer periods in databases.

In the following topics, you learn more about storing data clusters in memory and in databases

[Data Clusters in ABAP Memory \[Page 363\]](#)

[Data Clusters in Databases \[Page 368\]](#)

Data Clusters in ABAP Memory

You can store data clusters in ABAP memory. ABAP memory is a storage area assigned to a particular transaction and any modules called from there with the keywords CALL or SUBMIT. For further information about the flow of transactions, see [Dialog Mode \[Page 1309\]](#)

ABAP memory is not dependent on the ABAP program or program module which generates it during a transaction. This means that an object stored in ABAP memory can be read again by any ABAP program during the same transaction. However, the ABAP memory described in this section is not the same as the global SAP memory which extends beyond transaction limits (see, for example, [Transferring SPA/GPA Parameters to Transactions \[Page 1123\]](#)).

ABAP memory allows you to pass data between different modularization units across several levels of the program hierarchy. For example, you can pass data between:

- executable programs (reports) and other reports called with SUBMIT
- transactions and executable programs (reports)
- different dialog modules
- programs and function modules

and so on.

The memory is released again when you leave the transaction.

You store data objects in memory with the EXPORT TO MEMORY statement.

[Storing Data Objects in ABAP Memory \[Page 364\]](#)

You read data objects from memory with the IMPORT FROM MEMORY statement.

[Reading Data Objects from Memory \[Page 365\]](#)

You delete data clusters in memory with the FREE MEMORY statement.

[Deleting Data Clusters in Memory \[Page 367\]](#)

Storing Data Objects in ABAP Memory

To write data objects from an ABAP program to ABAP memory, you use the following statement:

Syntax

EXPORT <f₁> [FROM <g₁>] <f₂> [FROM <g₂>]... TO MEMORY ID <key>.

This statement stores the data objects specified in the list as a cluster in ABAP memory. If you omit the option FROM <g_i>, a data object <f_i> is stored under its own name. If you use the option, a data object <g_i> is stored under the name <f_i>. The ID <key>, which can be up to 32 characters long, identifies the data in memory.

The EXPORT statement always completely overwrites the contents of any existing data cluster with the same ID <key>.

In the case of internal tables with a header line, you can only store the table itself, not the header line. In the EXPORT statement, the table name is interpreted as a table. This is an exception to the general rule which stipulates that statements normally interpret the table name as a table work area (see [Choosing a Table Type \[Page 267\]](#)).

```
PROGRAM SAPMZTS1.

DATA TEXT1(10) VALUE 'Exporting'.

DATA ITAB LIKE SBOOK OCCURS 10 WITH HEADER LINE.

DO 5 TIMES.
  ITAB-BOOKID = 100 + SY-INDEX.
  APPEND ITAB.
ENDDO.

EXPORT TEXT1
  TEXT2 FROM 'Literal'
  TO MEMORY ID 'text'.

EXPORT ITAB
  TO MEMORY ID 'table'.
```

In this example, the text fields TEXT1 and TEXT2 are stored under the ID "text" and the internal table ITAB under the ID "table" in the ABAP memory of the program SAPMZTS1.

Reading Data Objects from Memory

Reading Data Objects from Memory

To read data objects from ABAP memory into an ABAP program, you use the following statement:

Syntax

```
IMPORT <f1> [TO <g1>] <f2> [TO <g2>]... FROM MEMORY ID <key>.
```

This statement reads the data objects specified in the list from a cluster in ABAP memory. If you omit the option TO <g_i>, the data object <f_i> is assigned from memory to the data object of the same name in the program. If you use this option, the data object <f_i> in memory is written to the field <g_i>. The ID <key>, which can be up to 32 characters long, identifies the data in memory.

You do not have to read all objects stored under a particular ID <key>. Instead, you can make a selection from the names <f_i>. If memory contains no objects under the specified ID <key>, SY-SUBRC is set to 4. However, if a data cluster with this ID exists in memory, the value of SY-SUBRC is always 0, regardless of whether the data object <f_i> also exists. If a data object <f_i> does not occur in the cluster, the target field remains unchanged.

This statement does not check whether the structure of the objects in memory matches those into which it is supposed to be written. Since the data is transported bit by bit, non-matching structures can cause inconsistencies.

```
PROGRAM SAPMZTS1.  
DATA TEXT1(10) VALUE 'Exporting'.  
DATA ITAB LIKE SBOOK OCCURS 10 WITH HEADER LINE.  
DO 5 TIMES.  
  ITAB-BOOKID = 100 + SY-INDEX.  
  APPEND ITAB.  
ENDDO.  
EXPORT TEXT1  
  TEXT2 FROM 'Literal'  
  TO MEMORY ID 'text'.  
EXPORT ITAB  
  TO MEMORY ID 'table'.  
SUBMIT SAPMZTS2 AND RETURN.  
SUBMIT SAPMZTS3.
```

The first part of this program corresponds to the example in [Storing Data Objects in ABAP Memory \[Page 364\]](#). The present example also calls the programs SAPMZTS1 and SAPMZTS2 with SUBMIT. You can generate and maintain the programs specified after SUBMIT by double-clicking on their names in the ABAP Editor. For more information about SUBMIT, refer to [Calling Executable Programs \(Reports\) \[Page 1113\]](#).

Example for SAPMZTS2:

```
PROGRAM SAPMZTS2.  
DATA: TEXT1(10),  
      TEXT3 LIKE TEXT1 VALUE 'Initial'.
```

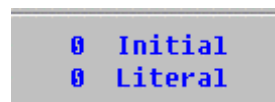
```
IMPORT TEXT3 FROM MEMORY ID 'text'.  
WRITE: / SY-SUBRC, TEXT3.
```

```
IMPORT TEXT2 TO TEXT1 FROM MEMORY ID 'text'.  
WRITE: / SY-SUBRC, TEXT1.
```

Example for SAPMZTS3:

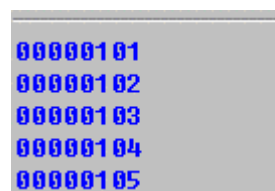
```
PROGRAM SAPMZTS3.  
DATA JTAB LIKE SBOOK OCCURS 10 WITH HEADER LINE.  
IMPORT ITAB TO JTAB FROM MEMORY ID 'table'.  
LOOP AT JTAB.  
  WRITE / JTAB-BOOKID.  
ENDLOOP.
```

Output is on two consecutive screens and looks as follows:



```
0 Initial  
0 Literal
```

and



```
00000101  
00000102  
00000103  
00000104  
00000105
```

SAPMZTS2 attempts to read a non-existent data object TEXT3 from the data cluster "text". Therefore, the target field TEXT3 remains unchanged. The existing data object TEXT2 is placed after TEXT1. In both cases, SY-SUBRC is set to 0 because the cluster "text" contains data.

SAPMZTS3 writes the internal table ITAB from the cluster "table" to the internal table JTAB. Both tables have the same structure, namely that of the ABAP Dictionary table SBOOK.

Deleting Data Clusters in Memory

Deleting Data Clusters in Memory

To delete data objects in ABAP memory, you use the following statement:

Syntax

FREE MEMORY [ID <key>].

Without the addition ID <key>, this statement deletes the entire memory, including all data clusters previously stored in ABAP memory with EXPORT. With the addition ID <key>, the statement deletes only the data cluster bearing this name.

You should use the FREE MEMORY statement **only** with the ID addition because deleting the entire memory also causes the memory contents of any system routines to be lost.

```
PROGRAM SAPMZTST.
DATA: TEXT(10) VALUE '0123456789',
      IDEN(3) VALUE 'XYZ'.
EXPORT TEXT TO MEMORY ID IDEN.
TEXT = 'xxxxxxxxxx'.
IMPORT TEXT FROM MEMORY ID IDEN.
WRITE: / SY-SUBRC, TEXT.

FREE MEMORY ID IDEN.
TEXT = 'xxxxxxxxxx'.
IMPORT TEXT FROM MEMORY ID IDEN.
WRITE: / SY-SUBRC, TEXT.
```

The output of this example is:

```
0 0123456789
4 xxxxxxxxxxxx
```

The FREE MEMORY ID IDEN statement deletes the data cluster "XYZ". Therefore, the system field SY-SUBRC is set to 4 after the next IMPORT statement and the target field remains unchanged.

Data Clusters in Databases

You can store data clusters in special databases of the ABAP Dictionary. These databases are known as ABAP cluster databases and have a predefined structure.

[Cluster Databases \[Page 369\]](#)

This method allows you to store complex data objects with any deep structure in a single step without having to adjust them to the flat structure of a relational database. As a result, your data objects are available throughout the system and every user has access to them. For a successful access, the data types of the stored objects must be known.

Storing data objects in cluster databases is useful for backing up the results of analyses made on information in relational databases. If, for example, you want to generate a list of the top customers in terms of sales or a complete address list from the personnel data of all branches, you can write ABAP programs which solve these problems and store the results as data clusters. If you need to refresh the stored data cluster, you can run these programs periodically in the background. To make use of the results, it is then possible to use other programs which only access the data cluster. This method can considerably reduce system response times because it is not necessary to access the distributed data in relational databases each time the results are used and also the result does not have to be regenerated each time.

Storing a data cluster is specific to ABAP. Although you can also access cluster databases using SQL statements, only ABAP statements are able to decode the structure of the stored data cluster.

You store data objects in cluster databases with the EXPORT TO DATABASE statement.

[Storing Data Objects in Cluster Databases \[Page 372\]](#)

You generate tables of contents for data clusters and read data objects from cluster databases with the IMPORT FROM DATABASE statement.

[Creating a Table of Contents for a Data Cluster \[Page 374\]](#)

[Reading Data Objects from Cluster Databases \[Page 376\]](#)

You delete data clusters from cluster databases with the DELETE FROM DATABASE statement.

[Deleting Data Clusters from Cluster Databases \[Page 378\]](#)

For information about accessing cluster databases with Open SQL statements, see

[Accessing Cluster Databases with Open SQL statements \[Page 379\]](#)

Cluster Databases

Cluster databases are special relational databases in the ABAP Dictionary. You use them to store data clusters. Their line structure is divided into a partly standardized start area consisting of several fields and one large field to store the data cluster.

The following topics describe the rules for setting up a cluster database and also discuss the system-defined cluster database INDX.

[Structure of Cluster Databases \[Page 370\]](#)

[Example of a Cluster Database \[Page 371\]](#)

Structure of Cluster Databases

The structure of a cluster database is as follows:

The rules for setting up a cluster database are described below. You must create the fields listed under points 1 to 4 as key fields. The data types mentioned are ABAP Dictionary types.

1. If the table is supposed to be client-specific, the first field must have the name MANDT, the type CHAR and a length of 3 bytes to store the client ID. When storing a data cluster, the system fills the field MANDT either automatically with the current client or uses the client explicitly specified in the EXPORT statement.
2. The next field (in the case of client-independent tables, this is the first field) must have the name RELID, the type CHAR and a length of 2 bytes. It contains an area ID. Cluster databases are divided into different areas. When storing a data cluster, the system fills the field RELID with the area ID specified in the EXPORT statement.
3. The next field is a type CHAR field of variable length. It contains the name <key> of the cluster which is specified in the program with the addition ID of the EXPORT statement when storing a data cluster. Since the subsequent field is aligned, the system might pad the end of the field RELID with up to 3 unused bytes. If you create your own cluster database, you should define the length of this field accordingly.
4. The next field must have the name SRTF2 and the type INT4 of length 4. Individual data clusters can extend across several lines of the database table. In theory, 2^{31} lines per cluster are possible. The field SRTF2 contains the sequential numbers of the lines within a stored data cluster and can contain any values between 0 and $2^{31}-1$. The system fills this field automatically when storing a data cluster (see 7. below).
5. SRTF2 can be followed by any number of user data fields with any permutation of name and type. The system does not fill these fields automatically when storing a data cluster. You must assign values to these fields explicitly before the EXPORT statement in the program. Normally, they contain control information like program name, user ID, etc.
6. The penultimate field on a line must have the name CLUSTR and the type INT2 of length 2. It contains the length of the data in the subsequent field CLUSTD. The system fills this field automatically when storing a data cluster.
7. The last field on a line must have the name CLUSTD and the type VARC. It can be any length, but usually occupies about 1000 bytes. When storing a data cluster, the system fills this field with the actual data in compressed form. If the length of CLUSTD is insufficient for a data cluster, it is distributed over several lines. These lines are numbered in the field SRTF2 (see 4. above).

You can either create your own cluster databases according to the above rules (in this case, see the documentation [BC ABAP Dictionary \[Ext.\]](#)) or use the system-defined cluster database INDX:

[Example of a Cluster Database \[Page 371\]](#)

Example of a Cluster Database

Example of a Cluster Database

The database INDX is an example of a cluster database and is included in your system as part of the standard installation. Intended for user applications, it is practical to use because you do not have to create a new one first. On the other hand, all users can access data you have stored there and change or delete it.

To view the structure of the database INDX in the ABAP Editor, choose *Edit* → *More functions* → *Command entry* and then enter SHOW INDX, or double-click on the word INDX, e.g. in a TABLES statement:

For each field, you then see the ABAP Dictionary data type and the corresponding data type of the ABAP programming language (i.e. the data type of the components in the table work area generated by TABLES).

The first four fields are the key fields of the table INDX and correspond exactly to the description in [Structure of Cluster Databases \[Page 370\]](#). The third field for the cluster name has the name SRTFD here and a length of 22 bytes, i.e. the name <key> specified in the addition ID of the EXPORT statement in the ABAP program can be up to 22 characters long for INDX.

The next seven fields are non-standard and intended for user input, e.g.:

- AEDAT: Date last changed
- USERA: User name
- PGMID: Program name

The last two fields are also predefined. The field CLUSTD for storing the actual data cluster is 2886 bytes long in the table INDX.

For examples of the usage of the table INDX, see

[Storing Data Objects in Cluster Databases \[Page 372\]](#)

[Creating a Table of Contents for a Data Cluster \[Page 374\]](#)

[Reading Data Objects from Cluster Databases \[Page 376\]](#)

[Deleting Data Clusters from Cluster Databases \[Page 378\]](#)

Storing Data Objects in Cluster Databases

To store data objects from an ABAP program in cluster databases, you use the following statement:

Syntax

```
EXPORT <f1> [FROM <g1>] <f2> [FROM <g2>]...
      TO DATABASE <dbtab>(<ar>) [CLIENT <cli>] ID <key>.
```

This statement stores the data objects specified in the list as a cluster in the cluster database <dbtab>. You must declare <dbtab> with the TABLES statement. Without the addition FROM <g_i>, a data object <f_i> is stored under its own name. With the addition, a data object <g_i> is stored under the name <f_i>.

<ar> is the two-character area ID for the cluster to be stored in the database.
(see point 2 under [Structure of Cluster Databases \[Page 370\]](#)).

<key>, which has a maximum length dependent on the length of the name field in <dbtab>, identifies the data in the database.
(see point 3 under [Structure of Cluster Databases \[Page 370\]](#)).

You can use the option CLIENT <cli> to switch off automatic client handling when you are dealing with client-specific cluster databases and specify the client yourself. You must specify this option immediately after the name of the database.
(see point 1 under [Structure of Cluster Databases \[Page 370\]](#)).

The EXPORT statement also transports the contents of the user fields of the table work area <dbtab> to the database table. You can fill these fields before, according to your requirements.
(see point 5 under [Structure of Cluster Databases \[Page 370\]](#)).

The EXPORT statement always overwrites the contents of any existing data cluster in the same area <ar> with the same name <key> and in the same client <cli> completely.

In the case of internal tables with a header line, you can only store the table itself, not the header line. In the EXPORT statement, the table name is interpreted as a table. This is an exception to the general rule which stipulates that statements normally interpret the table name as a table work area (see [Choosing a Table Type \[Page 267\]](#)).

```
PROGRAM SAPMZTS1.
TABLES INDX.
DATA: BEGIN OF ITAB OCCURS 100,
      COL1 TYPE I,
      COL2 TYPE I,
      END OF ITAB.
DO 3000 TIMES.
  ITAB-COL1 = SY-INDEX.
  ITAB-COL2 = SY-INDEX ** 2.
  APPEND ITAB.
ENDDO.
```

Storing Data Objects in Cluster Databases

```

INDX-AEDAT = SY-DATUM.
INDX-USERA = SY-UNAME.
INDX-PGMID = SY-REPID.

EXPORT ITAB TO DATABASE INDX(HK) ID 'Table'.

WRITE: '  SRTF2',
       AT 20 'AEDAT',
       AT 35 'USERA',
       AT 50 'PGMID'.
ULINE.

SELECT * FROM INDX WHERE RELID = 'HK'
       AND  SRTFD = 'Table'.

WRITE: / INDX-SRTF2 UNDER 'SRTF2',
       INDX-AEDAT UNDER 'AEDAT',
       INDX-USERA UNDER 'USERA',
       INDX-PGMID UNDER 'PGMID'.

ENDSELECT.

```

An internal table ITAB is filled with 3000 lines and, after the assignment of values to some user fields of INDX, ITAB is exported to INDX.

Since INDX is a relational database, you can address the individual lines with Open SQL statements. With the SELECT statement, the lines stored using EXPORT are selected by suitable WHERE conditions.

The output of database fields would be as follows:

SRTF2	AEDAT	USERA	PGMID
0	07.12.1995	KELLERH	SAPMZTST
1	07.12.1995	KELLERH	SAPMZTST
2	07.12.1995	KELLERH	SAPMZTST
3	07.12.1995	KELLERH	SAPMZTST
4	07.12.1995	KELLERH	SAPMZTST
5	07.12.1995	KELLERH	SAPMZTST

Here, the output shows that the user fields AEDAT, USERA, and PGMID were transported by EXPORT and that the data cluster containing ITAB is extended by 6 lines. If you change the number 3000 in the DO statement, the number of lines occupied by this data cluster also changes.

Creating a Table of Contents for a Data Cluster

To create the table of contents for a data cluster from an ABAP cluster database, you use the following statement:

Syntax

```
IMPORT DIRECTORY INTO <dirtab>
    FROM DATABASE <dbtab>(<ar>)
    [CLIENT <cli>] ID <key>.
```

This statement creates a list of the data objects in the data cluster stored in the database <dbtab> and places it in the table <dirtab>. You must declare <dbtab> with the TABLES statement.

To store a data cluster in a database, you normally use an EXPORT TO DATABASE statement (see [Storing Data Objects in Cluster Databases \[Page 372\]](#)). For information about the structure of the database table <dbtab>, see [Structure of Cluster Databases \[Page 370\]](#).

<ar> is the two-character area ID for the cluster to be stored in the database. <key>, which has a maximum length dependent on the length of the name field in <dbtab>, identifies the data in the database. You can use the option CLIENT <cli> to switch off automatic client handling when you are dealing with client-specific cluster databases and specify the client yourself. You must specify this option immediately after the name of the database.

The IMPORT statement automatically reads also the contents of the user fields of the table work area <dbtab> from the database table.

If a table of contents can be created, SY-SUBRC is set to 0. Otherwise, its value is 4.

You must set up the internal table <dirtab> according to the ABAP Dictionary structure CDIR. To do this, use the addition LIKE of the DATA statement (see [Basic Form of the DATA Statement \[Page 120\]](#)). The structure CDIR has the following components:

Field name	Type	Description
NAME	CHAR	Name of stored object in cluster.
OTYPE	CHAR	Type of object: F means elementary field R means field string T means internal table
FTYPE	CHAR	Data type of object. Structured data types are type C.
TFILL	INT4	Number of filled lines (for internal tables).
FLENG	INT2	Length of field or structure.

```
PROGRAM SAPMZTS2.
```

```
TABLES INDX.
```

```
DATA DIRTAB LIKE CDIR OCCURS 10 WITH HEADER LINE.
```

```
IMPORT DIRECTORY INTO DIRTAB FROM DATABASE
    INDX(HK) ID 'Table'.
```

Creating a Table of Contents for a Data Cluster

```
IF SY-SUBRC = 0.  
  WRITE: / 'AEDAT:', INDX-AEDAT,  
        / 'USERA:', INDX-USERA,  
        / 'PGMID:', INDX-PGMID.  
  WRITE / 'Directory:'.  
  LOOP AT DIRTAB.  
    WRITE: / DIRTAB-NAME, DIRTAB-OTYPE, DIRTAB-FTYPE,  
          DIRTAB-TFILL, DIRTAB-FLENG.  
  ENDLOOP.  
ELSE.  
  WRITE 'Not found'.  
ENDIF.
```

This program creates a table of contents for the data cluster stored with the example program in [Storing Data Objects in Cluster Databases \[Page 372\]](#). The output is as follows:

The table of contents DIRTAB contains a line showing that the data cluster contains an internal table called ITAB with 3000 filled lines with a length of 8 bytes.

Reading Data Objects from Cluster Databases

To read data objects from an ABAP cluster database into an ABAP program, you use the following statement:

Syntax

```
IMPORT <f1> [TO <g1>] <f2> [TO <g2>]...
      FROM DATABASE <dbtab>(<ar>)
      [CLIENT <cli>] ID <key>|MAJOR-ID <maid> [MINOR-ID <miid>].
```

This statement reads the data objects specified in the list from a data cluster in the database <dbtab>. You must declare <dbtab> with the TABLES statement. Without the addition TO <g_i>, the data object <f_i> from the database is assigned to the data object of the same name in the program. With this addition, the data object <f_i> of the database is written to the field <g_i>.

To store a data cluster in a database, you normally use an EXPORT TO DATABASE statement (see [Storing Data Objects in Cluster Databases \[Page 372\]](#)). For information about the structure of the database table <dbtab>, see [Structure of Cluster Databases \[Page 370\]](#).

<ar> is the two-character area ID for the cluster to be stored in the database. <key>, which has a maximum length dependent on the length of the name field in <dbtab>, identifies the data in the database. You can replace the addition ID <key> with MAJOR-ID <maid>. Then, the data cluster is selected whose first part of the name matches <maid>. If you specify the addition MINOR-ID <miid> with MAJOR-ID, the data cluster is selected whose second part of the name (i.e. the positions after the length of <maid>) will be greater than or equal to <miid>. You can use the option CLIENT <cli> to switch off automatic client handling when you are dealing with client-specific cluster databases and specify the client yourself. You must specify this option immediately after the name of the database.

The IMPORT statement automatically reads also the contents of the user fields of the table work area <dbtab> from the database table.

You do not have to read all the objects stored under a particular name <key>, but can make a selection using the names <f_i>. If the database contains no objects with the specified keys <ar>, <key>, and <cli>, SY-SUBRC is set to 4. However, if a data cluster with these keys exists in the database, the value of SY-SUBRC is always 0, regardless of whether the data object <f_i> also exists. If there is no data object <f_i> in the cluster, the target fields remains unchanged.

At runtime, the system checks in this statement that the structure of the objects in the database matches those into which they are supposed to be written. If this is not the case, a runtime error occurs. Exceptions to this rule are type C fields which can also appear at the end of a structured data object. These can be lengthened, shortened, added, or omitted.

```
PROGRAM SAPMZTS3.

TABLES INDX.

DATA: BEGIN OF JTAB OCCURS 100,
      COL1 TYPE I,
      COL2 TYPE I,
      END OF JTAB.

IMPORT ITAB TO JTAB FROM DATABASE INDX(HK) ID 'Table'.
```

Reading Data Objects from Cluster Databases

```
WRITE: / 'AEDAT:', INDX-AEDAT,  
      / 'USERA:', INDX-USERA,  
      / 'PGMID:', INDX-PGMID.  
  
SKIP.  
WRITE 'JTAB:'.  
  
LOOP AT JTAB FROM 1 TO 5.  
  WRITE: / JTAB-COL1, JTAB-COL2.  
ENDLOOP.
```

This program reads the internal table ITAB stored with the example program in [Storing Data Objects in Cluster Databases \[Page 372\]](#) from the cluster database INDX into the internal table JTAB. The output of some user fields of INDX and the first five lines of JTAB is as follows:

```
SY-SUBRC:      0  
AEDAT: 07.12.1995  
USERA: KELLERH  
PGMID: SAPM2TST  
  
JTAB:  
      1      1  
      2      4  
      3      9  
      4     16  
      5     25
```

Deleting Data Clusters from Cluster Databases

To delete data clusters from cluster databases, you use the following statement:

Syntax

```
DELETE FROM DATABASE <dbtab>(<ar>) [CLIENT <cli>] ID <key>.
```

This statement deletes the entire data cluster for the area <ar> and name <key> of the database table <dbtab>. You must declare <dbtab> with the TABLES statement.

You can use the option CLIENT <cli> to switch off automatic client handling when you are dealing with client-specific cluster databases and specify the client yourself. You must specify this option immediately after the name of the database.

To store a data cluster in a database, you normally use an EXPORT TO DATABASE statement (see [Storing Data Objects in Cluster Databases \[Page 372\]](#)). For information about the structure of the database table <dbtab>, see [Structure of Cluster Databases \[Page 370\]](#).

This DELETE statement deletes from the cluster database all the lines covered by the specified data cluster.

If a data cluster with the specified key can be deleted, SY-SUBRC is set to 0. Otherwise, its value is 4.

```
PROGRAM SAPMZTS4.
TABLES INDX.
DATA DIRTAB LIKE CDIR OCCURS 10.
IMPORT DIRECTORY INTO DIRTAB FROM DATABASE
      INDX(HK) ID 'Table'.
WRITE: / 'SY-SUBRC, IMPORT:', SY-SUBRC.
DELETE FROM DATABASE INDX(HK) ID 'Table'.
WRITE: / 'SY-SUBRC, DELETE:', SY-SUBRC.
IMPORT DIRECTORY INTO DIRTAB FROM DATABASE
      INDX(HK) ID 'Table'.
WRITE: / 'SY-SUBRC, IMPORT:', SY-SUBRC.
```

This program deletes the data cluster stored with the example program in [Storing Data Objects in Cluster Databases \[Page 372\]](#). If the data cluster exists when the program is started, the output is as follows:

```
SY-SUBRC,  IMPORT:      0
SY-SUBRC,  DELETE:      0
SY-SUBRC,  IMPORT:      4
```

In the first IMPORT statement, the data cluster still exists. The DELETE statement is then successfully executed. In the second IMPORT statement, the data cluster no longer exists.

Accessing Cluster Databases with Open SQL statements

Cluster databases are relational databases defined in the ABAP Dictionary which are used in a special way by ABAP. Therefore, you can in principal also access them with the Open SQL statements described in [Reading and Processing Database Tables \[Page 538\]](#).

To make meaningful use of Open SQL statements with cluster database tables, you must be aware of the special structure of these database tables (see [Structure of Cluster Databases \[Page 370\]](#)).

For example, it makes little sense to read the fields CLUSTR and CLUSTID with the SELECT statement or to change them with the UPDATE statement. These fields contain the data cluster coded by the system and can only be correctly handled by the EXPORT TO DATABASE and IMPORT FROM DATABASE statements.

You should only use the Open SQL statements UPDATE, MODIFY, and DELETE if certain combinations of the statements for data clusters result in excessive runtimes. You should not use the Open SQL statement INSERT in cluster databases at all.

You can use Open SQL statements to maintain your cluster database. For example, SELECT statements allow you to scan the cluster database table for particular data clusters. Here, you can also use information from the user data fields (see example in [Storing Data Objects in Cluster Databases \[Page 372\]](#)). The IMPORT FROM DATABASE statement is not suitable for this purpose.

```
PROGRAM SAPMZTS5.
DATA COUNT TYPE I VALUE 0.
TABLES INDX.
SELECT * FROM INDX WHERE RELID = 'HK'
      AND SRTF2 = 0
      AND USERA = SY-UNAME.
DELETE FROM DATABASE INDX(HK) ID INDX-SRTFD.
IF SY-SUBRC = 0.
  COUNT = COUNT + 1.
ENDIF.
ENDSELECT.
WRITE: / COUNT, 'Cluster(s) deleted'.
```

This example program deletes from the table INDX all data clusters in the area "HK" where the field USERA contains the name of the current program user. The field SRTFD of the table work area INDX is filled by the SELECT statement and used in the DELETE statement. Specifying SRTF2 = 0 in the WHERE clause ensures that each data cluster is only procoessed once.

Do not confuse the DELETE statement of the Open SQL command set (see [Deleting Lines from Database Tables \[Page 584\]](#)) with the DELETE statement for data clusters (see [Deleting Data Clusters from Cluster Databases \[Page 378\]](#)).

Accessing Cluster Databases with Open SQL statements

When deleting data in a data cluster, you should always delete all the lines, not just particular lines.

The following example demonstrates how you can change the name and the area of a data cluster in a database table with the Open SQL statement UPDATE. Solving this problem with the cluster statements EXPORT, IMPORT, and DELETE would be considerably more expensive.

```
PROGRAM SAPMZTS5.

TABLES INDX.

DATA DIRTAB LIKE CDIR OCCURS 10 WITH HEADER LINE.

UPDATE INDX SET RELID = 'NW'
      SRTFD = 'Internal'
      WHERE RELID = 'HK'
      AND SRTFD = 'Table'.

WRITE: / 'UPDATE:',
      / 'SY-SUBRC:', SY-SUBRC,
      / 'SY-DBCNT:', SY-DBCNT.

IMPORT DIRECTORY INTO DIRTAB FROM DATABASE
      INDX(NW) ID 'Internal'.

WRITE: / 'IMPORT:',
      / 'SY-SUBRC:', SY-SUBRC.
```

This program changes the data cluster stored with the example program in [Storing Data Objects in Cluster Databases \[Page 372\]](#). If the data cluster exists when the program is started, and there are no other errors in the UPDATE statement, the output is as follows:

UPDATE:

SY-SUBRC: 0

SY-DBCNT: 6

IMPORT:

SY-SUBRC: 0

The UPDATE statement changes the six lines of the database table INDX belonging to the specified data cluster, as described in [Changing Several Lines \[Page 579\]](#). Then, the IMPORT DIRECTORY statement finds the data cluster in the area "NW" under the name "Internal".

Working with Files

Working with Files

ABAP allows you to work with sequential files located on the application server or on the presentation server

These files can, for example, serve as temporary storage facilities for data or as an interface between local programs and the SAP system.

[Working with Files on the Application Server \[Page 382\]](#)

[Working with Files on the Presentation Server \[Page 412\]](#)

The physical addressing of files and file paths is platform-dependent. The R/3 System provides a function module and some transactions that allow you to work with platform-independent file names:

[Using Platform-Independent File Names \[Page 428\]](#)

Working with Files on the Application Server

ABAP provides some statements for working with data stored not in databases, but in sequential files on the application server. The following topics describe:

[File Handling in ABAP \[Page 383\]](#)

[Writing Data to Files \[Page 402\]](#)

[Reading Data from Files \[Page 404\]](#)

When working with sequential files on the application server, the system performs some automatic checks. These checks can possibly result in runtime errors.

[Automatic Checks before File Operations \[Page 406\]](#)

File Handling in ABAP

ABAP provides three statements for handling files:

- The OPEN DATASET statement opens a file.
- The CLOSE DATASET statement closes a file.
- The DELETE DATASET statement deletes a file.

[Opening a File \[Page 384\]](#)

[Closing a File \[Page 400\]](#)

[Deleting a File \[Page 401\]](#)

Opening a File

To open a file on the application server, use the OPEN DATASET statement. The basic form of the OPEN DATASET statement is described in:

[Basic Form of the OPEN DATASET Statement \[Page 385\]](#)

The OPEN DATASET statement has several options covering a number of tasks. These allow you

ABAP specific options:

[Opening a File for Reading \[Page 386\]](#)

[Opening a File for Writing \[Page 387\]](#)

[Opening a File for Appending \[Page 390\]](#)

[Specifying Binary Mode \[Page 392\]](#)

[Specifying Text Mode \[Page 394\]](#)

[Opening a File at a Specific Position \[Page 396\]](#)

Options for the operating system:

[Sending Operating System Commands \[Page 398\]](#)

[Receiving the Operating System Message \[Page 399\]](#)

For further options see the keyword documentation for the OPEN DATASET statement.

Basic Form of the OPEN DATASET Statement

Basic Form of the OPEN DATASET Statement

To open a file on the application server, use the OPEN DATASET statement as follows:

Syntax

OPEN DATASET <dsn> [options].

This statement opens file <dsn>. If you do not specify any options for the mode, the file is opened for reading in binary mode (see following topics). If the system can open the file, the system field SY-SUBRC is set to 0. Otherwise, SY-SUBRC returns 8.

You specify the filename <dsn> either as a literal or as a field which contains the file name. If you do not specify a path, the system opens the file in the directory where the SAP system is running on the application server. To open a file, the user under which the SAP system is running must have the appropriate authorizations at operating system level.

File names are platform-dependent. You must specify file and path names according to the rules of the operating system under which the SAP system is running. To write programs which are not dependent on the operating system, you can use logical file names (for more information on logical file names, see [Using Platform-Independent File Names \[Page 428\]](#)).

DATA FNAME(60).

FNAME = '/tmp/myfile'.

OPEN DATASET 'myfile'.

OPEN DATASET FNAME.

This example works if the SAP system is running under UNIX. The program opens the file "myfile" in the directory where the SAP system is running and opens the file "myfile" in the directory "/tmp". For other operating systems, you must substitute other file names. For OpenVMS, for example, you can specify the following:

FNAME = '[TMP]myfile.BIN'

OPEN DATASET 'myfile.BIN'.

Opening a File for Reading

To open a file for read access, use the FOR INPUT option of the OPEN DATASET statement as follows:

Syntax

```
OPEN DATASET <dsn> FOR INPUT.
```

This statement opens a file for reading. The file must already exist, otherwise the system sets SY-SUBRC to 8 and ignores the command.

If the file is already open (either for reading, writing, or appending), the system resets the positioning to the beginning of the file. However, it is better programming style to use the CLOSE statement for already open files before opening them again (for information on closing files, see [Closing a File \[Page 400\]](#)).

```
DATA FNAME(60) VALUE 'myfile'.  
OPEN DATASET FNAME FOR INPUT.  
IF SY-SUBRC = 0.  
  WRITE / 'File opened'.  
  ....  
ELSE.  
  WRITE / 'File not found'.  
ENDIF.
```

In this example, file "myfile" is opened for reading.

Opening a File for Writing

Opening a File for Writing

To open a file for write access, use the FOR OUTPUT option of the OPEN DATASET statement as follows:

Syntax

OPEN DATASET <dsn> FOR OUTPUT.

This statement opens a file for writing. If the file does not exist, it is created. If the file already exists and is closed, its contents are deleted. If the file already exists and is open (either for reading, writing, or appending), the positioning is reset to the beginning of the file. If the system can open the file, SY-SUBRC is set to 0. Otherwise, SY-SUBRC returns 8.

```
DATA: MESS(60),  
      FNAME(10) VALUE 'tmp'.  
  
OPEN DATASET FNAME FOR OUTPUT MESSAGE MESS.  
  
IF SY-SUBRC <> 0.  
  WRITE: 'SY-SUBRC:', SY-SUBRC,  
        / 'System Message:', MESS.  
ENDIF.
```

If the SAP System is running under UNIX, the output of this example appears as follows:

```
SY-SUBRC:      8  
System Message: Is a directory
```

The system cannot open the file, because it is a directory.

The following program demonstrates how the positioning is set when a file is opened for writing. However, it is better programming style to use the CLOSE statement for already open files before opening them again (for information on closing files, see [Closing a File \[Page 400\]](#)).

```
DATA FNAME(60) VALUE 'myfile'.  
DATA NUM TYPE I.  
  
OPEN DATASET FNAME FOR OUTPUT.  
  
DO 10 TIMES.  
  NUM = NUM + 1.  
  TRANSFER NUM TO FNAME.  
ENDDO.  
  
PERFORM INPUT.  
  
OPEN DATASET FNAME FOR OUTPUT.  
  
NUM = 0.  
DO 5 TIMES.  
  NUM = NUM + 10.  
  TRANSFER NUM TO FNAME.  
ENDDO.
```

```
PERFORM INPUT.  
CLOSE DATASET FNAME.  
OPEN DATASET FNAME FOR OUTPUT.  
NUM = 0.  
DO 5 TIMES.  
  NUM = NUM + 20.  
  TRANSFER NUM TO FNAME.  
ENDDO.  
PERFORM INPUT.  
FORM INPUT.  
  SKIP.  
  OPEN DATASET FNAME FOR INPUT.  
  DO.  
    READ DATASET FNAME INTO NUM.  
    IF SY-SUBRC <> 0.  
      EXIT.  
    ENDIF.  
    WRITE / NUM.  
  ENDDO.  
ENDFORM.
```

The output of this program appears as follows:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
  
10  
20  
30  
40  
50  
6  
7  
8  
9  
10  
  
20  
40  
60  
80  
100
```

In this example, file "myfile"

Opening a File for Writing

- is opened for writing.
- is filled with 10 integers (for information on the TRANSFER statement, see [Writing Data to Files \[Page 402\]](#)).
- is opened for reading as well. This resets the positioning to the beginning of the file.
- is read into field NUM (for information on the READ DATASET statement, see [Reading Data from Files \[Page 404\]](#)). The values of NUM are written to the output screen.
- is opened again for writing. This resets the positioning to the beginning of the file.
- is filled with 5 integers. These integers overwrite the old contents of the file.
- is opened again for reading. This resets the positioning to the beginning of the file.
- is read into field NUM. The values of NUM are written to the output screen.
- is closed (for information on the CLOSE DATASET statement, see [Closing a File \[Page 400\]](#)).
- is opened again for writing. This deletes the contents of the existing file.
- is filled with 5 integers.
- is opened for reading as well. This resets the positioning to the beginning of the file.
- is read into field NUM. The values of NUM are written to the output screen.

Opening a File for Appending

To open a file for appending data to the file, use the FOR APPENDING option of the OPEN DATASET statement as follows:

Syntax

OPEN DATASET <dsn> FOR APPENDING.

This statement opens a file for writing at the end of the file. If the file does not exist, it is created. If the file already exists and is closed, the system opens it and sets the positioning to the end of the file. If the file exists and is already open (either for reading, writing, or appending), the positioning is set to the end of the file. SY-SUBRC always returns 0.

It is better programming style to always use the CLOSE statement for already open files before opening them again (for information on closing files, see [Closing a File \[Page 400\]](#)).

```
DATA FNAME(60) VALUE 'myfile'.
DATA NUM TYPE I.
OPEN DATASET FNAME FOR OUTPUT.
DO 5 TIMES.
  NUM = NUM + 1.
  TRANSFER NUM TO FNAME.
ENDDO.
OPEN DATASET FNAME FOR INPUT.
OPEN DATASET FNAME FOR APPENDING.
NUM = 0.
DO 5 TIMES.
  NUM = NUM + 10.
  TRANSFER NUM TO FNAME.
ENDDO.
OPEN DATASET FNAME FOR INPUT.
DO.
  READ DATASET FNAME INTO NUM.
  IF SY-SUBRC <> 0.
    EXIT.
  ENDIF.
  WRITE / NUM.
ENDDO.
```

The output of this example appears as follows:

```
1
2
3
4
5
10
20
30
```

Opening a File for Appending

40

50

In this example, a file "myfile" is opened for writing and filled with five integer numbers from 1 to 5 (for information on the TRANSFER statement, see [Writing Data to Files \[Page 402\]](#)). The next OPEN DATASET statement resets the positioning to the beginning. Then, the file is opened for appending and the positioning is set to its end. Five integer numbers from 10 to 50 are written to the file. Finally, the contents of the file are read and written to the screen.

Specifying Binary Mode

To process a file in binary mode, use the IN BINARY MODE option of the OPEN DATASET statement as follows:

Syntax

OPEN DATASET <dsn> IN BINARY MODE [FOR....].

If you read data from a file or write data to a file that is opened in binary mode, the data are transmitted byte by byte. The contents of the file are not interpreted during the transmission. When you write the contents of a field into a file, the system transmits all bytes of the source field. When you read data from a file into a field, the number of transferred bytes depends on the target field length. When, after reading, you address the target field by another ABAP statement, the system interprets the contents of the field according to its data type.

```
DATA FNAME(60) VALUE 'myfile'.
DATA: NUM1    TYPE I,
      NUM2    TYPE I,
      TEXT1(4) TYPE C,
      TEXT2(8) TYPE C,
      HEX     TYPE X.

OPEN DATASET FNAME FOR OUTPUT IN BINARY MODE.

NUM1 = 111.
TEXT1 = 'TEXT'.
TRANSFER NUM1 TO FNAME.
TRANSFER TEXT1 TO FNAME.

OPEN DATASET FNAME FOR INPUT IN BINARY MODE.
READ DATASET FNAME INTO TEXT2.
WRITE / TEXT2.

OPEN DATASET FNAME FOR INPUT IN BINARY MODE.
READ DATASET FNAME INTO NUM2.
WRITE / NUM2.

OPEN DATASET FNAME FOR INPUT IN BINARY MODE.
SKIP.
DO.
  READ DATASET FNAME INTO HEX.
  If SY-SUBRC <> 0.
    EXIT.
  ENDIF.
  WRITE HEX.
ENDDO.
```

The output of this example appears as follows:

```
###oTEXT
```

```
111
```

```
00 00 00 6F 54 45 58 54
```


Specifying Binary Mode

After opening file "myfile" for writing in binary mode, the contents of fields NUM1 and TEXT1 are written into the file (for information on the TRANSFER statement, see [Writing Data to Files \[Page 402\]](#)). Then, the file is opened for reading and its complete contents are transferred into field TEXT2 (for information on the READ DATASET statement, see [Reading Data from Files \[Page 404\]](#)). The first four positions of character string TEXT2 make no sense, because the respective bytes are the (platform-dependend) representation of the integer number 111. The system tries to interpret all bytes as characters. This interpretation works only for the last four bytes. After resetting the positioning with an OPEN statement, the first four bytes of the file are read to NUM2. The value of NUM2 makes sense, because it has the same data type as NUM1. Finally, the eight bytes of the file are read one by one into field HEX. From the screen output of HEX, you can see the hexadecimal presentation of the actual file contents. The last four bytes are the ASCII representation of the characters in the word TEXT.

Specifying Text Mode

To process a file in text mode, use the IN TEXT MODE option of the OPEN DATASET statement as follows:

Syntax

OPEN DATASET <dsn> FOR.... IN TEXT MODE.

If you read data from a file or write data to a file that is opened in text mode, the data is transmitted line by line. The system assumes that the file has a line structure.

- For each TRANSFER statement (for information on the TRANSFER statement, see [Writing Data to Files \[Page 402\]](#)), the system transfers all bytes, except trailing blanks, to the file and places an end-of-line marker behind them.
- For each READ DATASET (for information on the READ DATASET statement, see [Reading Data from Files \[Page 404\]](#)), the system reads all data until the next end-of-line marker. If the target field is too small, the line is truncated. If the target field is longer than the line, it is filled with blanks from the right.

You should use the text mode if you want to write character strings to files or if you know that an existing file is formatted on a line basis. With text mode, you can read, for example, files that were created with an arbitrary text editor on your application server.

The following demonstration program is written for SAP Systems running on UNIX systems that use an ASCII representation.

```
DATA FNAME(60) VALUE 'myfile'.

DATA: TEXT(4),
      HEX TYPE X.

OPEN DATASET FNAME FOR OUTPUT IN TEXT MODE.

TRANSFER '12    ' TO FNAME.
TRANSFER '123456 9 ' TO FNAME.
TRANSFER '1234   ' TO FNAME.

OPEN DATASET FNAME FOR INPUT IN TEXT MODE.

READ DATASET FNAME INTO TEXT.
WRITE / TEXT.
READ DATASET FNAME INTO TEXT.
WRITE TEXT.
READ DATASET FNAME INTO TEXT.
WRITE TEXT.

OPEN DATASET FNAME FOR INPUT IN BINARY MODE.

SKIP.
DO.
  READ DATASET FNAME INTO HEX.
  If SY-SUBRC <> 0.
    EXIT.
  ENDIF.
  WRITE HEX.
ENDDO.
```

Specifying Text Mode

The output of this program appears as follows:

12 1234 1234

31 32 0A 31 32 33 34 35 36 20 20 39 0A 31 32 33 34 0A

In this example, a file "myfile" is opened for writing in text mode. Three string literals, each of length 10, are transferred to the file. After opening the file for reading in text mode, the stored lines are read into field TEXT of length 4. The first line is filled with two blanks from the right. The last five positions of the second line are truncated. Opening the file in binary mode and reading it into the hexadecimal field HEX shows its actual structure: The numbers between 31 and 36 are the ASCII representations of the numeric characters 1 to 6, while 20 represents the space character. The end of each line is marked by 0A. Note that the trailing spaces of the string literals are not written to the file.

Opening a File at a Specific Position

To open a file at a specific position, use the AT POSITION option of the OPEN DATASET statement as follows:

Syntax

OPEN DATASET <dsn> [FOR....] [IN... MODE] AT POSITION <pos>.

This statement opens a file <dsn> and sets the positioning to <pos> for reading or writing. The position <pos> is counted in bytes starting from the beginning of the file. You cannot specify a position before the beginning of the file.

The specification of a position <pos> makes sense mainly when you work in binary mode, because the physical representation of a text file depends on the operating system.

```
DATA FNAME(60) VALUE 'myfile'.
DATA NUM TYPE I.

OPEN DATASET FNAME FOR OUTPUT AT POSITION 16.
DO 5 TIMES.
  NUM = NUM + 1.
  TRANSFER NUM TO FNAME.
ENDDO.

OPEN DATASET FNAME FOR INPUT.
DO 9 TIMES.
  READ DATASET FNAME INTO NUM.
  WRITE / NUM.
ENDDO.

OPEN DATASET FNAME FOR INPUT AT POSITION 28.
SKIP.
DO 2 TIMES.
  READ DATASET FNAME INTO NUM.
  WRITE / NUM.
ENDDO.
```

The output of this example appears as follows:

```
0
0
0
0
1
2
3
4
5

4
5
```

In this example, file "myfile" is opened in the default binary mode. The starting position is specified as 16 for writing and as 28 for reading. As shown in this example, position specifications for writing and reading integer numbers make sense if the positions are divisible by four.

Sending Operating System Commands

When you work with the operating systems UNIX or WINDOWS NT, you can send an operating system command with the statement OPEN DATASET. To do so, use the option FILTER as follows:

Syntax

OPEN DATASET <dsn> FILTER <filt>.

The system processes the operating system command that is contained in the field <filt> when opening the file <dsn>.

The following example works with the operation system UNIX:

```
DATA DSN(20) VALUE '/usr/test.Z'.
```

```
OPEN DATASET DSN FOR OUTPUT FILTER 'compress'.
```

```
.....
```

```
OPEN DATASET DSN FOR INPUT FILTER 'uncompress'.
```

The first OPEN statement opens the file '/usr/test.Z' such, that data are written in a compressed format to this file.

The second OPEN statement opens the file '/usr/test.Z' such, that the system uncompresses data during reading from that file.

Receiving the Operating System Message

Receiving the Operating System Message

To receive the operating system message after trying to open a file, use the MESSAGE option of the OPEN DATASET statement as follows:

Syntax

OPEN DATASET <dsn> MESSAGE <msg>.

The system places the relevant operating system message in the variable <msg>.

Use this option together with the system field SY-SUBRC for error handling.

```
DATA: MESS(60),  
      FNAME(10) VALUE 'hugo.xyz'.  
OPEN DATASET FNAME MESSAGE MESS.  
IF SY-SUBRC <> 0.  
  WRITE: 'SY-SUBRC:', SY-SUBRC,  
        / 'System Message:', MESS.  
ENDIF.
```

If the SAP System is running under UNIX and the file "hugo.xyz" does not exist, the output of this example appears as follows:

```
SY-SUBRC:      8  
System Message: No such file or directory
```

Closing a File

To close a file on the application server, use the CLOSE DATASET statement as follows:

Syntax

CLOSE DATASET <dsn>.

This statement closes file <dsn>. The naming of the file is described in [Opening a File \[Page 384\]](#).

The closing of a file is only necessary if you want to delete its contents during its next opening for writing (for more information and for an example, see [Opening a File for Writing \[Page 387\]](#)).

To avoid errors and to make your programs easier to read, you should always close a file before you use the next OPEN DATASET statement. By using the CLOSE statement, you divide your program into logical blocks and make it easier to maintain.

```
DATA FNAME(60) VALUE 'myfile'.  
OPEN DATASET FNAME FOR OUTPUT.  
.....  
CLOSE FNAME.  
OPEN DATASET FNAME FOR INPUT.  
.....  
CLOSE FNAME.  
OPEN DATASET FNAME FOR INPUT AT POSITION <pos>.  
.....  
CLOSE FNAME.
```

Although the CLOSE statement is not necessary in this example, it improves the layout of the program.

Deleting a File

Deleting a File

To delete a file on the application server, use the DELETE DATASET statement as follows:

Syntax

```
DELETE DATASET <dsn>.
```

This statement deletes file <dsn>. The naming of the file is described in [Opening a File \[Page 384\]](#).

If the system can delete file <dsn>, SY-SUBRC returns 0. Otherwise, SY-SUBRC returns 4.

```
DATA FNAME(60) VALUE 'myfile'.  
OPEN DATASET FNAME FOR OUTPUT.  
OPEN DATASET FNAME FOR INPUT.  
IF SY-SUBRC = 0.  
  WRITE / 'File found'.  
ELSE.  
  WRITE / 'File not found'.  
ENDIF.  
DELETE DATASET FNAME.  
OPEN DATASET FNAME FOR INPUT.  
IF SY-SUBRC = 0.  
  WRITE / 'File found'.  
ELSE.  
  WRITE / 'File not found'.  
ENDIF.
```

The output of this example appears as follows:

File found

File not found

In this example, file "myfile" is created by opening it for writing, provided it did not exist. The system does find this file when opening it for reading. After the DELETE DATASET statement, the system does not find the file any more.

Writing Data to Files

To write data to a file on the application server, use the TRANSFER statement as follows:

Syntax

TRANSFER <f> to <dsn> [LENGTH <len>].

This statement writes the value of field <f> into file <dsn>. You can specify the transfer mode with the OPEN DATASET statement. If the file is not opened for writing, the system tries to open it in binary mode or with the options of the last OPEN DATASET statement for this file. However, it is good programming style to open a file only with the OPEN DATASET statement. The OPEN DATASET statement and the naming of the file are described in [Opening a File \[Page 384\]](#). The data type of field <f> can be elementary or it can be a field string that does not contain internal tables as components. Internal tables cannot be written to files in one execution..

With the LENGHT option, you can specify the length <len> of the data to be transferred. The system writes the first <len> bytes to the file. If <len> is too small, superfluous bytes are truncated. If <len> is too large, the system fills the transferred line with blanks from the right.

The following program demonstrates how to write an internal tables into a file:

```
DATA FNAME(60) VALUE 'myfile'.
TYPES: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
        END OF LINE.
TYPES ITAB TYPE LINE OCCURS 10.
DATA: LIN TYPE LINE,
      TAB TYPE ITAB.
DO 5 TIMES.
  LIN-COL1 = SY-INDEX.
  LIN-COL2 = SY-INDEX ** 2.
  APPEND LIN TO TAB.
ENDDO.
OPEN DATASET FNAME FOR OUTPUT.
LOOP AT TAB INTO LIN.
  TRANSFER LIN TO FNAME.
ENDLOOP.
CLOSE DATASET FNAME.
OPEN DATASET FNAME FOR INPUT.
DO.
  READ DATASET FNAME INTO LIN.
  IF SY-SUBRC <> 0.
    EXIT.
  ENDIF.
  WRITE: / LIN-COL1, LIN-COL2.
ENDDO.
CLOSE DATASET FNAME.
```

The output of this example appears as follows:

Writing Data to Files

1	1
2	4
3	9
4	16
5	25

In this example, a field string LIN and an internal table TAB are created with line type LINE. After filling internal table TAB, it is written line by line to file "myfile". Then the file is read into the field string LIN and the contents of LIN is written to the output screen.

The following demonstration program is written for SAP Systems running on UNIX systems that use an ASCII representation.

```
DATA FNAME(60) VALUE 'myfile'.

DATA: TEXT1(4) VALUE '1234',
      TEXT2(8) VALUE '12345678',
      HEX TYPE X.

OPEN DATASET FNAME FOR OUTPUT IN TEXT MODE.
TRANSFER: TEXT1 TO FNAME LENGTH 6,
          TEXT2 TO FNAME LENGTH 6.
CLOSE DATASET FNAME.

OPEN DATASET FNAME FOR INPUT.
DO.
  READ DATASET FNAME INTO HEX.
  IF SY-SUBRC <> 0.
    EXIT.
  ENDIF.
  WRITE HEX.
ENDDO.
CLOSE DATASET FNAME.
```

For this example, the output appears as follows:

```
31 32 33 34 20 20 0A 31 32 33 34 35 36 0A
```

In this example, two character strings TEXT1 and TEXT2 are written in text mode to file "myfile". The output length is specified to a value of 6. By reading the file byte by byte into the hexadecimal field, you see how the file is stored: The numbers 31 to 36 are the ASCII representations of the numeric characters 1 to 6. 20 represents a space and 0A marks the end of a line. TEXT1 is filled with two spaces from the right and two positions of TEXT2 are truncated.

Reading Data from Files

To read data from a file on the application server, use the READ DATASET statement as follows:

Syntax

READ DATASET <dsn> INTO <f> [LENGTH <len>].

This statement reads data from the file <dsn> into variable <f>. To decide into which variable you read data from a file, you must know the structure of the file.

You can specify the transfer mode with the OPEN DATASET statement. If the file is not opened for reading, the system tries to open it in binary mode or with the options of the last OPEN DATASET statement for this file. However, it is good programming style to open a file only with the OPEN DATASET statement. The OPEN DATASET statement and the naming of the file are described in [Opening a File \[Page 384\]](#).

After a successful reading operation, SY-SUBRC returns 0. When the end of the file is reached, SY-SUBRC returns 4. When the file cannot be opened, SY-SUBRC returns 8.

If you work in binary mode, you can use the LENGTH option to find out the length of the actual data transferred to <f>. The system sets the value of variable <len> to this length.

```
DATA FNAME(60) VALUE 'myfile'.
DATA: TEXT1(12) VALUE 'abcdefghijkl',
      TEXT2(5),
      LENG TYPE I.

OPEN DATASET FNAME FOR OUTPUT IN BINARY MODE.
TRANSFER TEXT1 TO FNAME.
CLOSE DATASET FNAME.

OPEN DATASET FNAME FOR INPUT IN BINARY MODE.
DO.
  READ DATASET FNAME INTO TEXT2 LENGTH LENG.
  WRITE: / SY-SUBRC, TEXT2, LENG.
  IF SY-SUBRC <> 0.
    EXIT.
  ENDIF.
ENDDO.
CLOSE DATASET FNAME.
```

The output of this example appears as follows:

```
0 abcde      5
0 fghij      5
4 kl###      2
```

In this example, file "myfile" is filled with 12 bytes from field TEXT1. Then, it is read in portions of 5 bytes into field TEXT2. Note that the system fills the last three bytes of TEXT2 with nulls after reaching the end of the file and that the number of actually transferred bytes is given in field LENG.

If you work in text mode, you can use the LENGTH option to find out the actual length of the current line in the file. The system sets the value of variable <len> to the line length. To do this, the system counts the number of bytes from the current position to the next end-of-line marker in the file.

Reading Data from Files

```
DATA FNAME(60) VALUE 'myfile'.  
DATA: TEXT1(4) VALUE '1234 ',  
      TEXT2(8) VALUE '12345678',  
      TEXT3(2),  
      LENG TYPE I.  
  
OPEN DATASET FNAME FOR OUTPUT IN TEXT MODE.  
  TRANSFER: TEXT1 TO FNAME,  
           TEXT2 TO FNAME.  
CLOSE DATASET FNAME.  
  
OPEN DATASET FNAME FOR INPUT IN TEXT MODE.  
  DO 2 TIMES.  
    READ DATASET FNAME INTO TEXT3 LENGTH LENG.  
    WRITE: / TEXT3, LENG.  
  ENDDO.  
CLOSE DATASET FNAME.
```

The output of this example appears as follows:

```
12      4  
12      8
```

In this example, the character strings TEXT1 and TEXT2 are written in text mode into file "myfile". Then they are read into the character string TEXT3 of length 2. The storage length of the lines is written into field LENG.

Automatic Checks before File Operations

The R/3 System performs the following checks automatically during operations with sequential files:

The system checks with the authorization object S_DATASET whether the current ABAP program is authorized to access the specified file.

[Authorization Check for particular Programs and Files \[Page 407\]](#)

By checking the contents of table SPTH, the system tests whether the specified file can be generally accessed from ABAP. Furthermore, table SPTH enables authorization checks of program users.

[General Check before File Access \[Page 409\]](#)

Authorization Check for particular Programs and Files

Authorization Check for particular Programs and Files

When an ABAP program accesses sequential files on the application server with the statements

- OPEN DATASET
- READ DATASET
- TRANSFER
- DELETE DATASET

the system performs an automatic authorization check with the authorization object S_DATASET.

You can use this object to assign authorizations to **specific** file access programs. You can also assign the authorization for using operating system commands as a file filter.

With the authorization object S_DATASET, you do **not** check the general permission for accessing files with ABAP and you **cannot** perform user-specific authorization checks. To do so, you use the table SPTH (see [General Check before File Access \[Page 409\]](#)).

The authorization object S_DATASET

The object consists of the following fields:

- ABAP program name
Name of the ABAP program containing the access. You can restrict file access to a few known access programs.
- Activity
The possible values are:
 - 33: Normal file read
 - 34: Normal file write or deletion
 - A6: Read file with filter (operating system command)
 - A7: Write to file with filter (operating system command)
- File name
Name of the operating system file. This allows you to restrict the accessible files.

You find more informations about authorization objects in the documentation [BC Users and Authorizations \[Ext.\]](#).

If the result of the automatic authorization check is negative, a run time error occurs.

Therefore, you should perform the authorization check in the ABAP program before the file access with the function module AUTHORITY_CHECK_DATASET.

The function module AUTHORITY_CHECK_DATASET

Authorization Check for particular Programs and Files

This function module allows you to check the user's authorization to access files before opening a file. By using this function module, you can avoid a run time error during the automatic authorization check.

The function module has the following import parameters:

- PROGRAM

Name of the ABAP program that contains the file access. If no program name is specified, the system assumes the current program.

- ACTIVITY

Access type. Possible values are:

- READ: read file
- WRITE: change file
- READ_WITH_FILTER: read file with filter function
- WRITE_WITH_FILTER: change file with filter function
- DELETE: delete file

These values are also defined as constants in the type group SABC:

TYPE-POOL SABC.

CONSTANTS:

SABC_ACT_READ(4)	VALUE 'READ',
SABC_ACT_WRITE(5)	VALUE 'WRITE',
SABC_ACT_READ_WITH_FILTER(16)	VALUE 'READ_WITH_FILTER',
SABC_ACT_WRITE_WITH_FILTER(17)	VALUE 'WRITE_WITH_FILTER',
SABC_ACT_DELETE(6)	VALUE 'DELETE',
SABC_ACT_INIT(4)	VALUE 'INIT',
SABC_ACT_ACCEPT(6)	VALUE 'ACCEPT',
SABC_ACT_CALL(4)	VALUE 'CALL'.

- FILENAME

Name of file to be accessed.

,

TYPE-POOLS SABC.

.....

```
CALL FUNCTION 'AUTHORITY_CHECK_DATASET'
  EXPORTING PROGRAM      = SY-REPID
            ACTIVITY     = SABC_ACT_READ
            FILENAME     = '/tmp/sapv01'
  EXCEPTIONS NO_AUTHORITY = 1
            ACTIVITY_UNKNOWN = 2.
```

.....

This call of the function module checks if the current program is authorized to access the file '/tmp/sapv01' for reading.

General Check before File Access

General Check before File Access

Before accessing sequential files on the application server with the statements

- OPEN DATASET
- TRANSFER
- DELETE DATASET

the system checks the table SPTH automatically.

Table SPTH controls generally read and write access of files from ABAP and whether files must be included in a backup routine.

The entries of table SPTH can prohibit generally and **independently** from the R/3 authorization concept the read and write access of files. The files can be specified **generically**. For the other files (for which read and write access is allowed in table SPTH), the system can perform an authorization check with from the R/3 authorization concept. For that, table SPTH can define authorization groups for program independent user authorization checks.

For this purposes, table SPTH has the following columns:

- PATH

This column contains generic file names. This means that the entries in a row of SPTH are valid for those files on the application servers, which match the generic file name in the PATH column most closely.

If SPTH contains the following three entries in the PATH column:

*

/tmp

/tmp/myfile

then the entries in the

- first row are valid for all files on the application server except those of the '/tmp' path.
- second row are valid for all files on the application server in the '/tmp' path except file '/tmp/myfile'.
- third row are valid for the file the '/tmp/myfile' on the application server.

- SAVEFLAG

This column contains flags that are set if they are marked with an 'X'.

If a flag is set, all files specified in the PATH column are included in a backup procedure.

- FS_NOREAD

This column contains flags that are set if they are marked with an 'X'.

General Check before File Access

If a flag is set, **no** ABAP program is allowed to access any of the files specified in PATH column. This flag overrides any user authorizations. If the FS_NOREAD flag is set, the FS_NOWRITE flag is set as well.

If a flag is not set, ABAP programs can access the files if they have the appropriate authorization (see column FSBGRU and [Authorization Check for particular Programs and Files \[Page 407\]](#)).

- FS_NOWRITE

This column contains flags that are set if they are marked with an 'X'.

If a flag is set, ABAP programs cannot change the files specified in the PATH column. This flag overrides any user authorizations.

If a flag is not set, ABAP programs can change the files if they have the appropriate authorization (see column FSBGRU and [Authorization Check for particular Programs and Files \[Page 407\]](#)).

- FSBGRU

This column contains the names of authorization groups.

An authorization group corresponds to the first field (RS_BRGRU) of the authorization object S_PATH. With the second field (ACTVT) of the authorization object S_PATH, you can check the authorization of a user for reading (value 3) or changing (value 2) the files of an authorization group.

Entries in column FSBGRU bundle files of the application servers in authorization groups. By defining authorizations for the authorization object S_PATH, you can control file access user specific.

Compared to authorization checks with the authorization object S_DATASET (see [Authorization Check for particular Programs and Files \[Page 407\]](#)), authorization checks with the authorization object S_PATH are independent from the current ABAP program. Furthermore, they are not restricted to single files but include all files that are specified generically in the column PATH.

If column FSBGRU is empty, the files specified in the PATH column do not belong to an authorization group and the system performs no authorization check with the authorization object S_PATH.

If the check on table SPTH yields a negative result, the system reacts with a runtime error.

Suppose table SPTH contains the following entries:

PATH	SAVEFLAG	FS_NOREAD	FS_NOWRITE	FSBGRU
*		X	X	
/tmp				
/tmp/files	X			FILE

General Check before File Access

With these settings, no ABAP program can access files on the application server other than those in the '/tmp' path.

All ABAP programs can read and change the files in the '/tmp' path.

Only users that have an authorization for the authorization group FILE can use ABAP programs that read and change files of path '/tmp/files'. These files are included in a backup procedure.

With the above table, the following program lines would result in a runtime error for all users:

```
DATA: FNAME(60).
```

```
FNAME = '/system/files'.
```

```
OPEN DATASET FNAME FOR OUTPUT.
```

Working with Files on the Presentation Server

To work with files on the presentation server, use special function modules that are provided in the function library. You must use an internal table as interface between program and function modules. The following topics describe:

[Writing Data to the Presentation Server With User Dialog \[Page 413\]](#)

[Writing Data to the Presentation Server Without User Dialog \[Page 416\]](#)

[Reading Data from the Presentation Server With User Dialog \[Page 419\]](#)

[Reading Data from the Presentation Server Without User Dialog \[Page 422\]](#)

[Checking Files on the Presentation Server \[Page 425\]](#)

The physical file names depend on the operating system of the presentation server. You can use logical file names to write platform-independent ABAP programs as described in [Using Platform-Independent File Names \[Page 428\]](#).

Writing Data to the Presentation Server With User Dialog

Writing Data to the Presentation Server With User Dialog

To write data from an internal table to the presentation server with user dialog, use the function module `DOWNLOAD`. The most important parameters are listed below. For more information, see the documentation of the function module in transaction `SE37`.

Important Import Parameters

Parameter	Function
<code>BIN_FILESIZE</code>	File length for binary files
<code>CODEPAGE</code>	Only for download in DOS: value IBM
<code>FILENAME</code>	Name of the file (default value for user dialog)
<code>FILETYPE</code>	File type (default value for user dialog)
<code>ITEM</code>	Header for user dialog window
<code>MODE</code>	Writing mode (empty = overwrite, 'A' = append)

With `FILETYPE` you specify the transfer mode. Possible values are:

- **BIN**
Binary files: you must specify the file length, and the internal table must consist of one column that has data type X.
- **ASC**
ASCII files.
- **DAT**
Excel files: the columns are separated by tabulators and the lines are separated by carriage returns and line feeds.
- **WK1**
Excel and Lotus files: the data are written to a WK1 spreadsheet.

Important Export Parameters

Parameter	Function
<code>ACT_FILENAME</code>	Name of the file (entered during user dialog)
<code>ACT_FILETYPE</code>	File type (entered during user dialog)
<code>FILESIZE</code>	Number of bytes transferred

Table Parameters

Parameter	Function
<code>DATA_TAB</code>	Internal source table

Exception Parameters

Parameter	Function
-----------	----------

Writing Data to the Presentation Server With User Dialog

INVALID_FILESIZE	Invalid parameter BIN_FILESIZE
INVALID_TABLE_WIDTH	Invalid table structure
INVALID_TYPE	Invalid value for parameter FILETYPE

Assume that the operating system used for presentation is WINDOWS NT and assume the following program:

```

PROGRAM SAPMZTST.

DATA: FNAME(128), FTYPE(3), FSIZE TYPE I.

TYPES: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
        END OF LINE.

TYPES ITAB TYPE LINE OCCURS 10.

DATA: LIN TYPE LINE,
      TAB TYPE ITAB.

DO 5 TIMES.
  LIN-COL1 = SY-INDEX.
  LIN-COL2 = SY-INDEX ** 2.
  APPEND LIN TO TAB.
ENDDO.

CALL FUNCTION 'DOWNLOAD'

  EXPORTING
    CODEPAGE      = 'IBM'
    FILENAME      = 'd:\temp\saptest.xls'
    FILETYPE      = 'DAT'
    ITEM          = 'Test for Excel File'

  IMPORTING
    ACT_FILENAME  = FNAME
    ACT_FILETYPE  = FTYPE
    FILESIZE      = FSIZE

  TABLES
    DATA_TAB     = TAB

  EXCEPTIONS
    INVALID_FILESIZE  = 1
    INVALID_TABLE_WIDTH = 2
    INVALID_TYPE      = 3.

WRITE: 'SY-SUBRC:', SY-SUBRC,
/ 'Name   : ', (60) FNAME,
/ 'Type   : ', FTYPE,
/ 'Size   : ', FSIZE.

```

After starting this program, the following user dialog window appears:

Writing Data to the Presentation Server With User Dialog

Transfer Test for Excel File to a Local File

File name: d:\temp\saptest.xls

Data format: DAT

OK Cancel

In this window, the user can change the default values. After clicking on **OK**, the system transfers the data from internal table **TAB**, which has been filled in the program, to file **d:\temp\saptest.xls**. If the file already exists, the system asks the user whether to replace it. The system places tabulators between the columns and carriage returns and line feeds at the end of each line.

The output of this example appears as follows:

SY-SUBRC: 0
Name : d:\temp\saptest.xls
Type : DAT
Size : 27

File **d:\temp\saptest.xls** can now be opened from Excel on the presentation server. The Excel screen then appears as follows:

	A	B	C
1	1	1	
2	2	4	
3	3	9	
4	4	16	
5	5	25	
6			

Writing Data to the Presentation Server Without User Dialog

To write data from an internal table to the presentation server without user dialog, use function module `WS_DOWNLOAD`. The most important parameters are listed below. For more information, see the documentation of the function module in transaction `SE37`.

Important Import Parameters

Parameter	Function
<code>BIN_FILESIZE</code>	File length for binary files
<code>CODEPAGE</code>	Only for download in DOS: value IBM
<code>FILENAME</code>	Name of the file
<code>FILETYPE</code>	File type
<code>MODE</code>	Writing mode (empty = overwrite, 'A' = append)

With `FILETYPE` you specify the transfer mode. Possible values are:

- **BIN**
Binary files: you must specify the file length, and the internal table must consist of one column that has data type X.
- **ASC**
ASCII files: the system places end-of-line markers behind each line.
- **DAT**
Excel files: the system separates columns by tabulators and lines by carriage returns and line feeds.
- **WK1**
Excel and Lotus files: the data are written to a WK1 spreadsheet.

Export Parameters

Parameter	Function
<code>FILELENGTH</code>	Number of bytes transferred

Table Parameters

Parameter	Function
<code>DATA_TAB</code>	Internal source table

Exception Parameters

Parameter	Function
<code>FILE_OPEN_ERROR</code>	System cannot open file
<code>FILE_WRITE_ERROR</code>	System cannot write to file

Writing Data to the Presentation Server Without User Dialog

INVALID_FILESIZE	Invalid parameter BIN_FILESIZE
INVALID_TABLE_WIDTH	Invalid table structure
INVALID_TYPE	Invalid value for parameter FILETYPE

Assume that the operating system used for presentation is WINDOWS NT and assume the following program:

```

PROGRAM SAPMZTST.

DATA: FLENGTH TYPE I.

DATA TAB(80) OCCURS 5.

APPEND 'This is the first line of my text. ' TO TAB.
APPEND 'The second line.                ' TO TAB.
APPEND '    The third line.              ' TO TAB.
APPEND '        The fourth line.         ' TO TAB.
APPEND '            Fifth and final line. ' TO TAB.

CALL FUNCTION 'WS_DOWNLOAD'

  EXPORTING
    CODEPAGE      = 'IBM'
    FILENAME      = 'd:\temp\saptest.txt'
    FILETYPE      = 'ASC'

  IMPORTING
    FILELENGTH    = FLENGTH

  TABLES
    DATA_TAB     = TAB

  EXCEPTIONS
    FILE_OPEN_ERROR   = 1
    FILE_WRITE_ERROR  = 2
    INVALID_FILESIZE  = 3
    INVALID_TABLE_WIDTH = 4
    INVALID_TYPE      = 5.

WRITE: 'SY-SUBRC : ', SY-SUBRC,
      / 'File length:', FLENGTH.

```

The output of this example appears as follows:

SY-SUBRC : 0

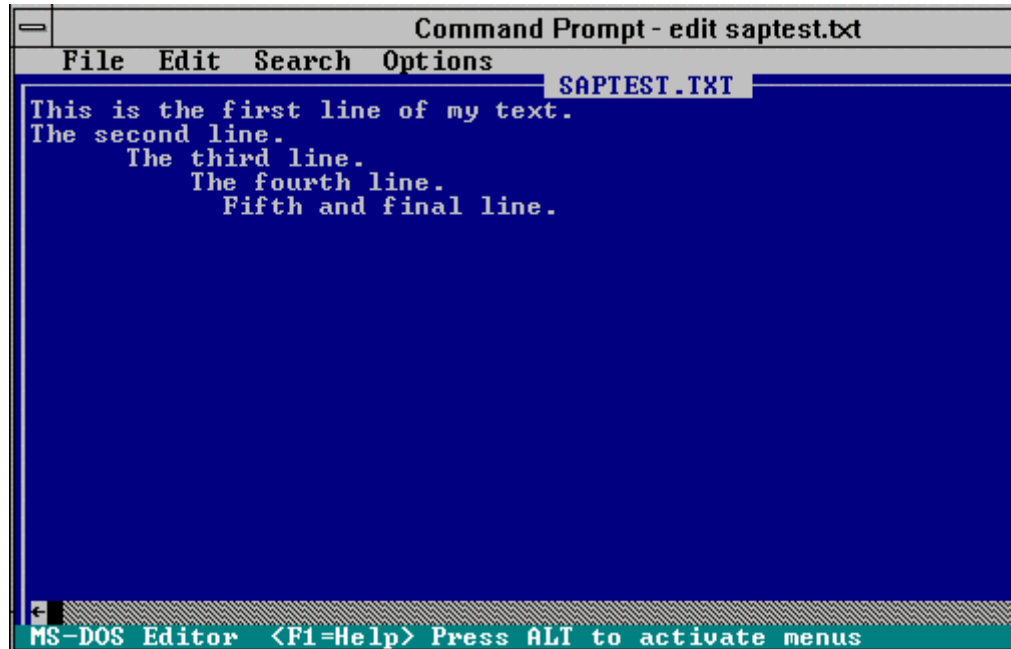
File length: 140

The system transferred the five lines of table TAB to the ASCII file d:\temp\saptest.txt. Using the WINDOWS File Manager, you can check the existence and length of the file as follows:



Writing Data to the Presentation Server Without User Dialog

The file can now be opened by any editor on the presentation server. In the following example, the DOS editor is used:



```
Command Prompt - edit saptest.txt
File Edit Search Options
SAPTEST.TXT
This is the first line of my text.
The second line.
    The third line.
        The fourth line.
            Fifth and final line.
MS-DOS Editor <F1=Help> Press ALT to activate menus
```

Reading Data from the Presentation Server With User Dialog

Reading Data from the Presentation Server With User Dialog

To read data from the presentation server into an internal table with user dialog, use function module `UPLOAD`. The most important parameters are listed below. For more information, see the documentation of the function module in transaction `SE37`.

Important Import Parameters

Parameter	Function
<code>CODEPAGE</code>	Only for upload in DOS: value IBM
<code>FILENAME</code>	Name of the file (default value for user dialog)
<code>FILETYPE</code>	File type (default value for user dialog)
<code>ITEM</code>	Header for user dialog window

With `FILETYPE` you specify the transfer mode. Possible values are:

- `BIN`
Binary files.
- `ASC`
ASCII files: text files with end-of-line markers.
- `DAT`
Excel files that are saved as text files, with columns that are separated by tabulators, and lines that are separated by carriage returns and line feeds.
- `WK1`
Excel and Lotus files that are saved as WK1 spreadsheet.

Important Export Parameters

Parameter	Function
<code>FILESIZE</code>	Number of bytes transferred
<code>ACT_FILENAME</code>	Name of the file (entered during user dialog)
<code>ACT_FILETYPE</code>	File type (entered during user dialog)

Table Parameters

Parameter	Function
<code>DATA_TAB</code>	Internal target table

Exception Parameters

Parameter	Function
<code>CONVERSION_ERROR</code>	Error in the data conversion
<code>INVALID_TABLE_WIDT</code>	Invalid table structure

Reading Data from the Presentation Server With User Dialog

H	
INVALID_TYPE	Incorrect parameter FILETYPE

Assume that the operating system used for presentation is WINDOWS NT and an Excel table as the one below:

	A	B	C
1	Billy	the	Kid
2	My	Fair	Lady
3	Herman	the	German
4	Conan	the	Barbarian

If this table is saved as a text file with tabulators between the columns to d:\temp\mytable.txt, the following program can read the table:

```

PROGRAM SAPMZTST.

DATA: FNAME(128), FTYPE(3), FSIZE TYPE I.

TYPES: BEGIN OF LINE,
        COL1(10) TYPE C,
        COL2(10) TYPE C,
        COL3(10) TYPE C,
        END OF LINE.

TYPES ITAB TYPE LINE OCCURS 10.

DATA: LIN TYPE LINE,
      TAB TYPE ITAB.

CALL FUNCTION 'UPLOAD'

  EXPORTING
    CODEPAGE      = 'IBM'
    FILENAME      = 'd:\temp\mytable.txt'
    FILETYPE      = 'DAT'
    ITEM          = 'Read Test for Excel File'

  IMPORTING
    FILESIZE      = FSIZE
    ACT_FILENAME  = FNAME
    ACT_FILETYPE  = FTYPE

  TABLES
    DATA_TAB     = TAB

  EXCEPTIONS
    CONVERSION_ERROR  = 1
    INVALID_TABLE_WIDTH = 2
    INVALID_TYPE      = 3.

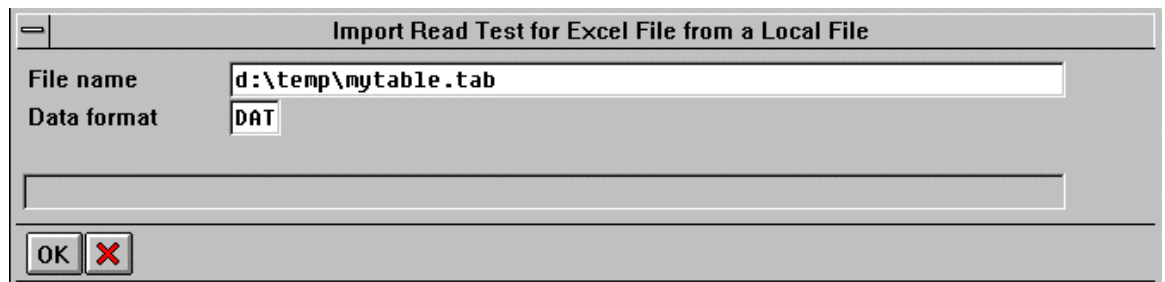
WRITE: 'SY-SUBRC:', SY-SUBRC,
      / 'Name   :', (60) FNAME,
      / 'Type   :', FTYPE,
      / 'Size   :', FSIZE.

```

Reading Data from the Presentation Server With User Dialog

```
SKIP.  
LOOP AT TAB INTO LIN.  
  WRITE: / LIN-COL1, LIN-COL2, LIN-COL3.  
ENDLOOP.
```

After starting this program, the following user dialog window appears:



Import Read Test for Excel File from a Local File

File name: d:\temp\mytable.tab

Data format: DAT

OK Cancel

In this window, the user can change the default values. After clicking on *OK*, the system transfers the data from file d:\temp\mytable.txt to internal table TAB.

The output of this example appears as follows:

```
SY-SUBRC: 0  
Name   : d:\temp\mytable.txt  
Type   : DAT  
Size   :      69  
  
Billy  the   Kid  
My     Fair  Lady  
Herman the   German  
Conan  the   Barbarian
```

The contents of internal table TAB are equal to the contents of the original Excel table.

Reading Data from the Presentation Server Without User Dialog

To read data from the presentation server into an internal table without user dialog, use function module `WS_UPLOAD`. The most important parameters are listed below. For more information, see the documentation of the function module in transaction `SE37`.

Important Export Parameters

Parameter	Function
<code>CODEPAGE</code>	Only for download in DOS: value IBM
<code>FILENAME</code>	Name of the file
<code>FILETYPE</code>	File type

With `FILETYPE` you specify the transfer mode. Possible values are:

- `BIN`
Binary files.
- `ASC`
ASCII files: text files with end-of-line markers.
- `DAT`
Excel files that are saved as text files, with columns that are separated by tabulators, and lines that are separated by carriage returns and line feeds.
- `WK1`
Excel and Lotus files that are saved as WK1 spreadsheet.

Export Parameters

Parameter	Function
<code>FILELENGTH</code>	Number of bytes transferred

Table Parameters

Parameter	Function
<code>DATA_TAB</code>	Internal target table

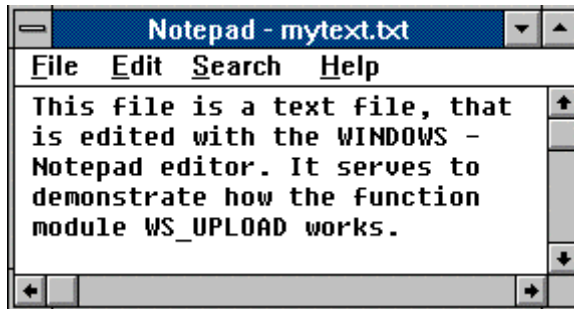
Exception Parameters

Parameter	Function
<code>CONVERSION_ERROR</code>	Error in the data conversion
<code>FILE_OPEN_ERROR</code>	System cannot open file
<code>FILE_READ_ERROR</code>	System cannot read from file
<code>INVALID_TABLE_WIDTH</code>	Invalid table structure

Reading Data from the Presentation Server Without User Dialog

INVALID_TYPE	Invalid value for parameter FILETYPE
--------------	--------------------------------------

Assume that the operating system used for presentation is WINDOWS NT and assume a text file as the one below:



The following program reads this text file:

```

PROGRAM SAPMZTST.

DATA: FLENGTH TYPE I.

DATA: TAB(80) OCCURS 5 WITH HEADER LINE.

CALL FUNCTION 'WS_UPLOAD'

  EXPORTING
    CODEPAGE      = 'IBM'
    FILENAME      = 'd:\temp\mytext.txt'
    FILETYPE      = 'ASC'

  IMPORTING
    FILELENGTH    = FLENGTH

  TABLES
    DATA_TAB     = TAB

  EXCEPTIONS
    CONVERSION_ERROR  = 1
    FILE_OPEN_ERROR   = 2
    FILE_READ_ERROR   = 3
    INVALID_TABLE_WIDTH = 4
    INVALID_TYPE       = 5.

WRITE: 'SY-SUBRC:', SY-SUBRC,
      / 'Length :', FLENGTH.

SKIP.
LOOP AT TAB.
  WRITE: / TAB.
ENDLOOP.

```

The output of this example appears as follows:

Reading Data from the Presentation Server Without User Dialog

```
SY-SUBRC:      0
Length   :      145

This file is a text file, that
is edited with the WINDOWS -
Notepad editor. It serves to
demonstrate how the function
module WS_UPLOAD works.
```


Checking Files on the Presentation Server

Checking Files on the Presentation Server

To get information about files on the presentation server and about the presentation server's operation system, use function module `WS_QUERY`. The most important parameters are listed below. For more information, see the documentation of the function module in transaction `SE37`.

Important Import Parameters

Parameter	Function
FILENAME	File name for query commands 'FE', 'FL', and 'DE'
QUERY	Query command

The import parameter `QUERY` defines the query command. In the following, some important query commands are listed:

- `CD`: query current directory
- `EN`: query environment variable
- `FL`: query length of file specified with `FILENAME`
- `FE`: query existence of file specified with `FILENAME`
- `DE`: query existence of directory specified with `FILENAME`
- `WS`: query windows system of presentation server
- `OS`: query operating system of presentation server

Export Parameters

Parameter	Function
RETURN	Result of query ('0' means 'no' and '1' means 'yes')

Exception Parameters

Parameter	Function
INV_QUERY	Wrong value for <code>QUERY</code> or <code>FILENAME</code>

Assume that the operating system used for presentation is `WINDOWS NT` and that the file `SYSTEM.INI` exists as shown here:



The following program determines some attributes of the operation system and of this file:

PROGRAM `SAPMZTST`.

DATA: `FNAME(60)`, `RESULT(30)`, `FLENGTH TYPE I`.

```
FNAME = 'C:\WINNT35\SYSTEM.INI'.

CALL FUNCTION 'WS_QUERY'
  EXPORTING
    QUERY      = 'OS'
  IMPORTING
    RETURN     = RESULT
  EXCEPTIONS
    INV_QUERY  = 1.

IF SY-SUBRC = 0.
  WRITE: / 'Operating System:', RESULT.
ENDIF.

CALL FUNCTION 'WS_QUERY'
  EXPORTING
    QUERY      = 'WS'
  IMPORTING
    RETURN     = RESULT
  EXCEPTIONS
    INV_QUERY  = 1.

IF SY-SUBRC = 0.
  WRITE: / 'Windows:', RESULT.
ENDIF.

CALL FUNCTION 'WS_QUERY'
  EXPORTING
    FILENAME   = FNAME
    QUERY      = 'FE'
  IMPORTING
    RETURN     = RESULT
  EXCEPTIONS
    INV_QUERY  = 1.

IF SY-SUBRC = 0.
  WRITE: / 'File exists ?', RESULT.
ENDIF.

CALL FUNCTION 'WS_QUERY'
  EXPORTING
    FILENAME   = FNAME
    QUERY      = 'FL'
  IMPORTING
    RETURN     = FLENGTH
  EXCEPTIONS
    INV_QUERY  = 1.

IF SY-SUBRC = 0.
  WRITE: / 'File Length:', FLENGTH.
ENDIF.
```

The output of this program appears as follows:

Operating System: NT

Windows: WN32

File exists ? 1

Checking Files on the Presentation Server

File Length: **210**

The windows system WN32 is the windows system of WINDOWS NT. For more information on the abbreviations, place the cursor on the QUERY field of the function module's documentation screen and choose *Help*.

Using Platform-Independent File Names

The file names that you specify in the ABAP statements for working with files are the physical file names. This means, that they must exactly fulfill the requirements of the operating system under which the SAP system is running. After creating a file from an ABAP program with a specific name and path, you can find this file exactly at this position after logging in on the operating system level.

Because the rules for the naming of files and paths differ from operating system to operating system, your ABAP programs are not portable from one operation system to another unless you use the tools that are described in the following.

To make ABAP programs portable, the SAP System provides the concept of logical file names and logical paths. Logical files and paths are connected to physical files and paths. This connection is established in special tables. You can maintain these tables according to your needs. In your ABAP programs, use the function module `FILE_GET_NAME` to create physical file names from the logical ones.

The maintenance of platform-independent file names is part of the customizing. You will find a comprehensive description by choosing *Tools → Business Engineering → Customizing → Implem. projects → Display complete IMG*. On the appearing screen choose *Basis → General basis → Platform independent assignment of filenames*.

For a description of function module `FILE_GET_NAME`, enter its name on the *ABAP function library: Introduction* screen (transaction SE37) and choose documentation there. On the appearing screen, click on *Function module doc*.

There is also a transaction named `FILE` to maintain platform-independent file names. The following topics provide a short overview on how to work with platform-independent file names by using the transaction `FILE`.

After calling the `FILE` transaction, you can maintain the above mentioned tables by selecting the respective lines in the *Navigation* frame. This is explained in the following topics.

[Maintaining Syntax Groups \[Page 429\]](#)

[Assigning Operating Systems to Syntax Groups \[Page 430\]](#)

[Creating and Defining Logical Paths \[Page 431\]](#)

[Creating and Defining Logical File Names \[Page 432\]](#)

Then, you learn how to use function module `FILE_GET_NAME` to convert logical file names into physical file names in ABAP programs.

[Using Logical Files in ABAP Programs \[Page 433\]](#)

You find further information about platform-independent file names in the documentation [BC Extended Applications Function Library \[Ext.\]](#).

Maintaining Syntax Groups

Maintaining Syntax Groups

Syntax groups comprise the names of all operating systems that follow the same syntax for the naming of files. Select *Syntax group definition* in the *Navigation* frame of the FILE transaction to display or maintain syntax groups. The following screen appears:

Syntax grp	Name	Leng.	Extension	Active
AS/400	AS/400	30	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
DOS	MS-DOS compatible	8	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
MACINTOSH	Apple Macintosh	32	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
UNIX	Unix compatible	50	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
WINDOWS NT	Microsoft Windows NT	255	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

On this screen, you can see a list of available syntax groups. You can define new syntax groups by clicking on *New entries*, if necessary. For information on how to assign an operating system to syntax groups, see [Assigning Operating Systems to Syntax Groups \[Page 430\]](#).

Assigning Operating Systems to Syntax Groups

Assigning Operating Systems to Syntax Groups

Select *Assignment of operating system to syntax group* in the *Navigation* frame of the FILE transaction to display or maintain the assignment of operating systems to existing syntax groups. The following screen appears:

Navigation

- Syntax group definition
- Assignment of operating system to syntax group

Level 5 of 6

OP system	Name
AIX	IBM Unix
BOS/X	Bull Unix
HP-UX	HP-UX
MC	Apple Macintosh
MF	Motif under VMS
OSF/1	DEC Unix
PM	Presentation Manager OS/2
SINIX	SNI Unix
SunOS	SunOS
WN	Windows /WFW
WN32	WN32
Windows NT	Microsoft Windows NT

On this screen, you see a list of currently supported operating systems. To create new entries, click on *New entries*. To assign existing entries to a syntax group, mark an operating system and click on *Details*. The following screen appears:

New entries

OP system: HP-UX

Name: HP-UX

Syntax group: UNIX Unix compatible

On this screen, the operating system HP-UX is assigned to the syntax group UNIX.

Creating and Defining Logical Paths

Creating and Defining Logical Paths

Each logical file name can be connected to a logical path. The logical path provides a logical file name with platform-dependent physical paths. To create a logical path, select *Logical file path definition* in the *Navigation* frame of the FILE transaction. On the appearing screen, choose *New entries* and define a new logical path as shown in the following example:

Navigation

- Logical file path definition
- >Assignment of logical to physical file paths
- Client-independent file name definition
- Definition of variables

Level 1 of 6

Create a logical file path

Logical file path	Name
TMP_SUB	Path for TMP directory

Save the logical path by choosing *Save*.

Select *→ Assignment of logical to physical file paths* in the *Navigation* frame of the FILE transaction to connect existing logical paths to physical paths and syntax groups. On the appearing screen, select either a logical path to maintain its connection to a syntax group or click on *New entries* to create new connections. The physical path for the syntax group UNIX can, for example, be defined as follows:

Logical path: TMP_SUB

Name:

Syntax group: UNIX

Physical path: /tmp/<FILENAME><PARAM_1>

For the input fields *Logical path* and *Syntax group* lists of possible entries are available. For creating the *Physical path*, you can use reserved words (enclosed by angle brackets) that are substituted with current values at runtime. You get a list of the reserved words by placing the cursor on the input field and choosing *Help*. The reserved word <FILENAME> must always be included in the physical path. Its actual value is defined with the logical file name that uses the logical path. The actual value of the reserved word <PARAM_1> that is used in the present example, is an import parameter of function module FILE_GET_NAME.

Creating and Defining Logical File Names

To create a logical file name, select *Client-independent file name definition* in the *Navigation* frame of the FILE transaction and choose *New entries*. Define a logical file name as shown in the following example:

Logical file	MYTEMP
Name	Example for ABAP/4 User's Guide
Physical file	TEST
Data format	BIN
ApplicationArea	BC
Logical path	TMP_SUB

You can either connect a logical file to a logical path, as done here, or enter the full physical file name in the input field *Phys. file*. In the latter case, the logical file name applies only to one operating system. The rules for entering a full physical file are the same as for defining the physical path for the logical path. Choose *Help* for further informations and a list of possible reserved words.

When connected to a logical path, the logical file applies to all syntax groups that are maintained for the logical path. The file name entered in *Phys. file* replaces the reserved word <FILENAME> in the physical paths assigned to the logical path. To keep this name independent of the operating system, choose names that start with a letter, comprise up to 8 characters, and do not contain special characters.

Save the definitions by choosing *Save*.

Using Logical Files in ABAP Programs

Using Logical Files in ABAP Programs

You use function module FILE_GET_NAME to create a physical file name from a logical file name in your ABAP programs. To insert the call of the function module in your program, choose *Edit* → *Insert statement* from the ABAP Editor screen. On the appearing dialog window, mark *Call Function* and type in FILE_GET_NAME. The parameters of this function module are listed below.

Import Parameters

Parameter	Function
CLIENT	The maintenance tables for the logical files and paths are client-dependent. Therefore, the desired client can be imported. The current client is stored in the system field SY-MANDT.
LOGICAL_FILENAME	Enter the logical file name in upper case letters that you want to convert.
OPERATING_SYSTEM	You can import any operating system that is contained in the list of transaction SF04 (see Assigning Operating Systems to Syntax Groups [Page 430]). The physical file name will be created according to the syntax group to which the operating system is connected. The default parameter is the value of the system field SY-OPSY.
PARAMETER_1 PARAMETER_2	If you specify these import parameters, the reserved words <PARAM_1> and <PARAM_2> in the physical path names will be replaced by the imported values.
USE_PRESENTATION_SERVER	With this flag you can decide whether to import the operating system of the presentation server instead of the operating system imported by the parameter OPERATING_SYSTEM.
WITH_FILE_EXTENSION	If you set this flag unequal to SPACE, the file format defined for the logical file name is appended to the physical file name.

Export Parameters

Parameter	Function
EMERGENCY_FLAG	If this parameter is unequal to SPACE, no physical name is defined in the logical path. An emergency physical name was created from table FILENAME and profile parameter DIR_GLOBAL.
FILE_FORMAT	This parameter is the file format defined for the logical file name. You can use this parameter, for example, to decide in which mode the file should be opened.
FILE_NAME	This parameter is the physical file name that you can use with the ABAP statements for working with files.

Exception Parameters

Parameter	Function
-----------	----------

Using Logical Files in ABAP Programs

FILE_NOT_FOUND	This exception is raised if the logical file is not defined.
OTHERS	This exception is raised if other errors occur.

Assume that the logical file MYTEMP and the logical path TMP_SUB are defined as in the preceding topics and assume the following program:

```

DATA: FLAG,
      FORMAT(3),
      FNAME(60).

WRITE SY-OPSY.

CALL FUNCTION 'FILE_GET_NAME'

  EXPORTING
    LOGICAL_FILENAME      = 'MYTEMP'
    OPERATING_SYSTEM      = SY-OPSY
    PARAMETER_1           = '01'

  IMPORTING
    EMERGENCY_FLAG        = FLAG
    FILE_FORMAT            = FORMAT
    FILE_NAME              = FNAME

  EXCEPTIONS
    FILE_NOT_FOUND        = 1
    OTHERS                 = 2.

IF SY-SUBRC = 0.
  WRITE: / 'Flag   : ', FLAG,
        / 'Format : ', FORMAT,
        / 'Phys. Name:', FNAME.
ENDIF.
```

The output of this program appears as follows:

HP-UX

FLAG :

FORMAT : BIN

Phys. Name: /tmp/TEST01

In this example, the SAP system is running under the operating system HP-UX, which is member of the syntax group UNIX. The logical file name MYTEMP with the logical path TMP_SUB is converted into a physical file name /tmp/TEST01 as defined for the syntax group UNIX. The field FNAME can be used in the further flow of the program to work with file TEST01 in directory /tmp.

Assume a logical file name EMPTY with the physical file name TEST, which is connected to a logical path that has no specification of a physical path. If you replace the EXPORTING parameter 'MYTEMP' with 'EMPTY' in the above example, the output appears as follows:

HP-UX

Using Logical Files in ABAP Programs

FLAG : X

FORMAT :

Phys. Name: /usr/sap/S11/SYS/global/TEST

The system created an emergency file name, whose path depends on the installation of the current SAP system.

Modularization

Modularizing ABAP Programs

Modularizing ABAP Programs

In the [Introduction to ABAP \[Ext.\]](#), you learned that modularization plays an essential role in ABAP.

Each ABAP program has a modular structure. ABAP programs are divided in:

- Processing blocks that are controlled by time events when the program is a report program (standalone program of type 1).

You find detailed information about that under [Controlling the Flow of ABAP Programs by Events \[Page 1208\]](#).

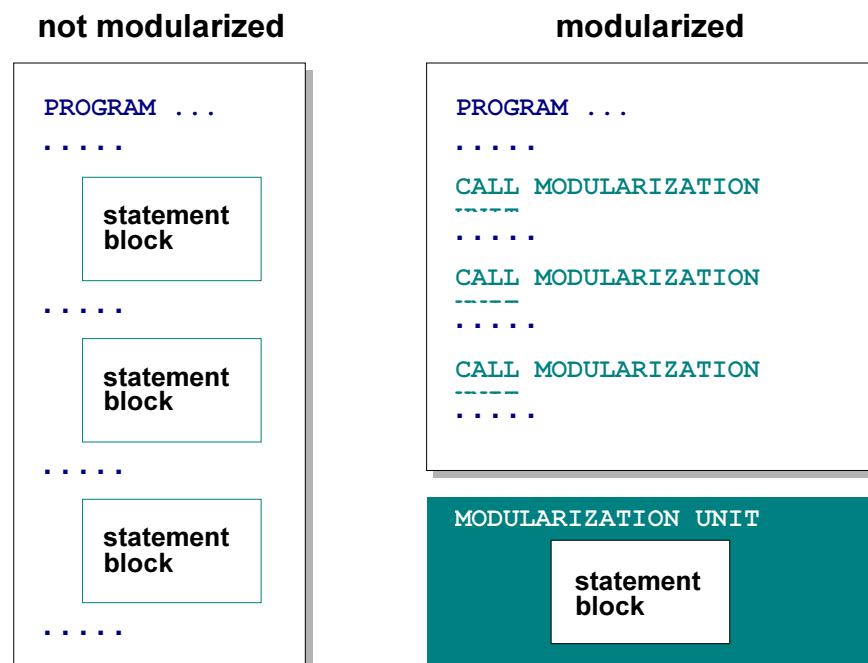
- Modules that are controlled by the screen flow logic when the program is a dialog program (module pool of type M).

You find detailed information about that under [Dialog Mode \[Page 1309\]](#).

In this section, you learn how you can modularize your ABAP programs beyond this basic concept. If your program contains the same or similar blocks of statements or you want to process the same function several times, you can avoid redundancy by using modularization techniques.

By modularizing your ABAP programs further, you make them easy to read and improve their structure. Modularized programs are also easier to maintain and to update than programs which have not been modularized.

Principle of modularization:



ABAP provides the following modularization techniques:

[Source Code Modules \[Page 439\]](#)

[Subroutines \[Page 444\]](#)

[Function Modules \[Page 477\]](#)

Source Code Modules

If you modularize your source code, your programs will be easier to maintain. You should use these modularization techniques if, for example, you want to avoid having to repeat the same statements several times in your program. However, you should not use this procedure for modularizing tasks and functions. For these purposes, you should use subroutines and function modules.

You can create callable modules of program code within your ABAP program by defining macros.

[Defining and Calling Macros \[Page 440\]](#)

You can also create include programs in the library. These contain modules of your source code.

[Include Programs \[Page 441\]](#)

Source code modularization with include programs is an essential aid for programming module pools and function modules. If you use the correct naming conventions, the ABAP Development Workbench supports this modularization via forward navigation.

Defining and Calling Macros

To define a macro which contains part of the source code, you use the DEFINE statement as follows:

Syntax

```
DEFINE <macro>.
```

```
    <statements>
```

```
END-OF-DEFINITION.
```

This defines the macro <macro>. You must specify complete statements between DEFINE and END-OF-DEFINITION. These statements can contain up to nine placeholders (&1, &2,..., &9).

After its definition, you can call a macro as follows:

Syntax

```
<macro> [<p1> <p2>... <p9>].
```

During generation of the program, the system replaces <macro> by the defined statements and each placeholder &i by the parameter <p_i>. You can call a macro from another one, but a macro cannot call itself.

```
DATA: RESULT TYPE I,
      N1  TYPE I VALUE 5,
      N2  TYPE I VALUE 6.

DEFINE OPERATION.
  RESULT = &1 &2 &3.
  OUTPUT  &1 &2 &3 RESULT.
END-OF-DEFINITION.

DEFINE OUTPUT.
  WRITE: / 'The result of &1 &2 &3 is', &4.
END-OF-DEFINITION.

OPERATION 4 + 3.
OPERATION 2 ** 7.
OPERATION N2 - N1.
```

This produces the following output:

```
The result of 4 + 3 is      7
The result of 2 ** 7 is   128
The result of N2 - N1 is    1
```

Here, two macros, OPERATION and OUTPUT, are defined. OUTPUT is nested in OPERATION. OPERATION is called three times with different parameters. Note how the place holders &1, &2,... are replaced in the macros.

Include Programs

Include Programs

If you want to use the same sequence of statements in several programs, you can code them once in an include program. This can be very important, for example, in the case of lengthy data declarations, which you want to use in different programs.

Furthermore, the include technique serves to modularize complex ABAP programs. Related parts from main programs of function modules and from module pools of dialog programs are principally coded as include programs. The ABAP Development Workbench supports you extensively when you create such complex programs by creating the include programs automatically and by naming them unambiguously.

The following topics describe

[Creating Include Programs \[Page 442\]](#)

[Using Include Programs \[Page 443\]](#)

You cannot pass data to and from include programs explicitly because their purpose is to modularize the source code. If you want to pass data to and from modules, use subroutines or function modules.

Creating Include Programs

To create an include program, follow the procedure explained in [Creating and Changing ABAP Programs \[Page 60\]](#).

For the program attribute type, you must use the type I as described in [Maintain Program Attributes \[Page 74\]](#).

You can also create or change an include program by double-clicking on the name of the program after the INCLUDE statement in your ABAP program (see [Using Include Programs \[Page 443\]](#)). By doing this, you can either create a new program or change an existing one.

An include program cannot run independently, but must be called from other programs. You can call include programs from other include programs. The only restrictions for writing the source code of include programs are:

- Include programs cannot contain PROGRAM or REPORT statements.
- Include programs cannot call themselves.
- Include programs must contain complete statements.

However, you must ensure that the statements of your include program fit logically into the source code of the programs from which it is called. Choosing *Check* while editing an include program in the ABAP Editor is normally not sufficient for this.

```
***INCLUDE INCL-TST.
```

```
TEXT = 'Hello!'.
```

Here, the syntax check reports an error because the field TEXT is not declared. However, you can call the program INCL-TST from any program in which the field TEXT is declared with a suitable type.

In order to obtain meaningful results from the syntax check, you must perform it for the program which calls the include program (see [Using Include Programs \[Page 443\]](#)).

Using Include Programs

Using Include Programs

To use an include program from another ABAP program, you use the INCLUDE statement as follows:

Syntax

```
INCLUDE <incl>.
```

This inserts the source code <incl> into the ABAP program during the syntax check and during generation. The INCLUDE statement performs the same function as if you were to copy the source code of <incl> to the position of the statement in the calling program.

The INCLUDE statement must be the only statement on one line and cannot extend over several lines.

Include programs are not loaded at runtime, but are resolved before the program is generated. After generation, the program contains the source code of all the include programs it uses.

Suppose you have written the following include program:

```
***INCLUDE STARTTXT.  
  
WRITE: / 'Program started by', SY-UNAME,  
       / 'on host', SY-HOST,  
       'date:', SY-DATUM, 'time:', SY-UZEIT.  
ULINE.
```

You can call this program from any other ABAP program. If, for example, you want to write a standard header for the output list, insert the INCLUDE statement after the PROGRAM or REPORT statement as follows:

```
PROGRAM SAPMZTST.  
INCLUDE STARTTXT.
```

.....

This could produce the following output:

```
Program started by FRED  
on host hs1077  date: 07/19/1995 time: 09:00:39
```

.....

Subroutines

Subroutines are program modules which can be called from ABAP programs. You define subroutines so that you only have to write frequently used parts of a program or algorithms once. You can explicitly pass data to and from subroutines.

There are two types of subroutines.

- Internal subroutines:

The source code of internal subroutines is in the same ABAP program as the calling procedure (internal call).

- External subroutines:

The source code of external subroutines is in an ABAP program other than the calling procedure (external call).

Although you use internal subroutines mainly to modularize and structure individual programs, a subroutine which is called internally in one ABAP program can be called externally from another ABAP program. On the other hand, it is also possible to create ABAP programs which contain only subroutines. These programs cannot run on their own, but are used by other ABAP programs as pools of external subroutines.

When developing larger applications, you should use function modules instead of external subroutines. In contrast to subroutines, function modules offer a defined interface, they are stored in a central library, and they obey a save release procedure.

The following topics describe

[Defining Subroutines \[Page 445\]](#)

[Calling Subroutines \[Page 446\]](#)

[Passing Data Between Calling Programs and Subroutines \[Page 451\]](#)

[Defining Local Data Types and Objects in Subroutines \[Page 469\]](#)

[Terminating Subroutines \[Page 474\]](#)

Defining Subroutines

Defining Subroutines

A subroutine is a block of code introduced by FORM and concluded by ENDFORM. To define a subroutine, use the following syntax:

Syntax

```
FORM <subr> [<pass>].
```

```
  <statement block>
```

```
ENDFORM.
```

The name of the subroutine is defined by <subr>. In the <pass> option, you specify how to pass data to and from the subroutine.

- In the case of internal subroutines, you do not have to use the <pass> option, but the subroutine can access all data objects declared in the main ABAP program.
- In the case of external subroutines, you must decide whether to use the <pass> option or declare data objects in common parts of the memory.

For more information about passing data to and from the subroutine, see [Passing Data Between Calling Programs and Subroutines \[Page 451\]](#)

If you define internal subroutines and do not use event keywords in the same program, you should group them together at the end of the program so as not to affect the program flow (see [Defining Processing Blocks \[Page 1209\]](#)).

A subroutine cannot contain nested FORM-ENDFORM blocks.

FORM HEADER.

```
  WRITE: / 'Program started by', SY-UNAME,  
         / 'on host', SY-HOST,  
         'date:', SY-DATUM, 'time:', SY-UZEIT.  
  ULINE.
```

ENDFORM.

A subroutine HEADER is created, which can be used to create a header in an output list (compare to example in [Using Include Programs \[Page 443\]](#)).

Calling Subroutines

You can call subroutines which are coded in the same ABAP program (internal calls), or subroutines which are coded in other ABAP programs (external calls).

[Calling Internal Subroutines \[Page 447\]](#)

[Calling External Subroutines \[Page 448\]](#)

You can specify the name of the subroutine at runtime or call a subroutine from a given list.

[Specifying the Subroutine Name at Runtime \[Page 449\]](#)

[Calling Subroutines from a List \[Page 450\]](#)

You can not only call subroutines from subroutines (nested calls), but subroutines can also call themselves (recursive calls).

Calling Internal Subroutines

Calling Internal Subroutines

To call an internal subroutine, use the PERFORM statement as follows:

Syntax

PERFORM <subr> [<pass>].

The subroutine <subr> is called. In the <pass> option, you specify how to pass data to and from the subroutine. Also if you do not use the <pass> option, the subroutine can access all data types and objects declared in the main ABAP program. This data is called global data and is visible to a subroutine if not hidden by local data definitions with the same names (for more information about passing data, see [Passing Data Between Calling Programs and Subroutines \[Page 451\]](#)).

```
PROGRAM SAPMZTST.
DATA: NUM1 TYPE I,
      NUM2 TYPE I,
      SUM TYPE I.
NUM1 = 2. NUM2 = 4.
PERFORM ADDIT.
NUM1 = 7. NUM2 = 11.
PERFORM ADDIT.
FORM ADDIT.
  SUM = NUM1 + NUM2.
  PERFORM OUT.
ENDFORM.
FORM OUT.
  WRITE: / 'Sum of', NUM1, 'and', NUM2, 'is', SUM.
ENDFORM.
```

The output appears as follows:

```
Sum of      2 and      4 is      6
Sum of      7 and     11 is     18
```

In this example, two internal subroutines ADDIT and OUT are defined at the end of the program. ADDIT is called from the program and OUT is called from ADDIT. The subroutines automatically have access to fields NUM1, NUM2, and SUM.

Calling External Subroutines

To call an external subroutine, use the PERFORM statement as follows:

Syntax

PERFORM <subr>(<prog>) [<pass>] [IF FOUND].

The subroutine <subr> which is defined in program <prog> is called. If you want to pass data to and from the subroutine, you must use the <pass> option or work with common parts (for further information about passing data, see [Passing Data Between Calling Programs and Subroutines \[Page 451\]](#)).

If you use the IF FOUND option and there is no subroutine <sub> in the program <prog>, the system ignores the PERFORM statement.

When you start a program which calls an external subroutine, ABAP loads the program in which the subroutine is defined into memory, if it is not already there. In order to save storage space, you should limit the number of subroutines that are defined in different programs to a minimum.

Suppose a program contains the following subroutine:

```
PROGRAM FORMPOOL.  
  
FORM HEADER.  
  WRITE: / 'Program started by', SY-UNAME,  
        / 'on host', SY-HOST,  
        'date:', SY-DATUM, 'time:', SY-UZEIT.  
  ULINE.  
ENDFORM.
```

You can call the subroutine from a program as follows:

```
PROGRAM SAPMZTST.  
  
PERFORM HEADER(FORMPOOL) IF FOUND.
```

In this example, no data is passed between calling program and subroutine. The subroutine performs output statements as in the example in [Using Include Programs \[Page 443\]](#).

Specifying the Subroutine Name at Runtime

Specifying the Subroutine Name at Runtime

You can specify the name of a subroutine you want to call and the name of the program in which it is stored at runtime. To do so, use the PERFORM statement as follows:

Syntax

PERFORM (<fsubr>) [IN PROGRAM (<fprog>)] [<pass>] [IF FOUND].

The system executes the subroutine stored in the field <fsubr>. If you use the IN PROGRAM option, the system searches for the subroutine in the program stored in the field <fprog> (external call). Otherwise, the system searches for the subroutine in the current program (internal call).

With this statement, you can also specify the names of subroutine and external program in the program code. To do so, omit the parentheses.

The <pass> option specifies how to pass data to and from the subroutine (for more information about passing data, see [Passing Data Between Calling Programs and Subroutines \[Page 451\]](#)). If you use the IF FOUND option, the system ignores the PERFORM statement if the subroutine <fsubr> is not found.

Assume a program that contains subroutines as follows:

```
PROGRAM FORMPOOL.
```

```
FORM SUB1.
```

```
  WRITE: / 'Subroutine 1'.
```

```
ENDFORM.
```

```
FORM SUB2.
```

```
  WRITE: / 'Subroutine 2'.
```

```
ENDFORM.
```

You can specify the name of these subroutines at runtime as follows:

```
PROGRAM SAPMZTST.
```

```
DATA: PROGNAME(8) VALUE 'FORMPOOL',  
      SUBRNAME(8).
```

```
SUBRNAME = 'SUB1'.
```

```
PERFORM (SUBRNAME) IN PROGRAM (PROGNAME) IF FOUND.
```

```
SUBRNAME = 'SUB2'.
```

```
PERFORM (SUBRNAME) IN PROGRAM (PROGNAME) IF FOUND.
```

The output appears as follows:

Subroutine 1

Subroutine 2

The character field PROGNAME contains the name of the program, in which the subroutines are contained. The names of the subroutines are assigned to the character field SUBRNAME.

Calling Subroutines from a List

To call a particular subroutine from a list, use the PERFORM statement as follows:

Syntax

PERFORM <idx> OF <form₁> <form₂>.... <form_n>.

The system executes the subroutine specified in the subroutine list in position <idx>. This variant of the PERFORM statement is only possible for internal calls. The field <idx> can be a variable or a literal.

```
PROGRAM SAPMZTST.  
DO 2 TIMES.  
  PERFORM SY-INDEX OF SUB1 SUB2.  
ENDDO.  
  
FORM SUB1.  
  WRITE / 'Subroutine 1'.  
ENDFORM.  
  
FORM SUB2.  
  WRITE / 'Subroutine 2'.  
ENDFORM.
```

The output appears as follows:

Subroutine 1

Subroutine 2

In this example, the two internal subroutines, SUB1 and SUB2, are called consecutively from a list.

Passing Data Between Calling Programs and Subroutines

The default handling of data is different for internal and external subroutines.

- In the case of internal subroutines, you can access all global data of the calling program directly from your subroutine, provided it is not hidden by local data with the same name. This global data consists of internal data objects and types, as well as all ABAP Dictionary fields which are referenced with the TABLES statement in the program (for more information about this statement, see [The TABLES Statement \[Page 129\]](#)).
- If you want to access the calling program data from external subroutines in the same way as from internal subroutines, you have to declare the data as common part in the calling program as well as in the programs that contain the external subroutines (see [Declaring Data as Common Part \[Page 452\]](#)).

When you work with global data in subroutines, you can put a copy of the global data on a local data stack. This is valid for global data in internal subroutines and for data declared as common part. To do so, you must work with field symbols (for more details about this topic, see [Assigning a Local Copy of a Global Field \[Page 359\]](#)).

You should use common parts only for simple modularizations. Especially when using function modules in nested subroutine calls (for more information about function modules, see [Function Modules \[Page 477\]](#)), the rules for accessing common parts can become complicated.

To make programs more transparent and flexible, you should choose the following possibility to pass data between calling programs and subroutines:

Specify explicitly the data that you need and may change in internal as well as in external subroutines. To do so, you can use parameters during defining and calling subroutines. These parameters are defined in the <pass> option of the FORM (see [Defining Subroutines \[Page 445\]](#)) and PERFORM (see [Calling Subroutines \[Page 446\]](#)) statements.

The following topics describe

[Declaring Data as Common Part \[Page 452\]](#)

[Passing Data by Parameters \[Page 454\]](#)

Declaring Data as Common Part

To declare data objects as common part, use the DATA statement as follows:

Syntax

```
DATA: BEGIN OF COMMON PART [<name>],
      <data declaration>,
      .....
      END OF COMMON PART [<name>].
```

In <data declaration>, you declare all data objects to be included in the common part as explained in [The DATA Statement \[Page 119\]](#).

Table work areas that are defined with the TABLES statement are automatically shared by subroutines and calling programs.

To use common parts in calling programs and external subroutines, you have to use exactly the same declarations in all programs that are involved. Therefore, you should place the declaration of common parts in INCLUDE programs (see [Include Programs \[Page 441\]](#)).

You can use several common parts in one program. In this case, you must assign a name <name> to each common part. If you use only one common part per program, the name <name> is optional.

To avoid conflicts between programs that have different common part declarations, you should always assign unique names to common parts.

Assume an INCLUDE program INCOMMON contains the declaration of a common part NUMBERS. The common part comprises three numeric fields: NUM1, NUM2, and SUM:

```
***INCLUDE INCOMMON.

DATA: BEGIN OF COMMON PART NUMBERS,
      NUM1 TYPE I,
      NUM2 TYPE I,
      SUM TYPE I,
      END OF COMMON PART NUMBERS.
```

Assume a program FORMPOOL includes INCOMMON and contains the subroutines ADDIT and OUT:

```
PROGRAM FORMPOOL.

INCLUDE INCOMMON.

FORM ADDIT.
  SUM = NUM1 + NUM2.
  PERFORM OUT.
ENDFORM.

FORM OUT.
  WRITE: / 'Sum of', NUM1, 'and', NUM2, 'is', SUM.
ENDFORM.
```

Declaring Data as Common Part

Assume a calling program SAPMZTST includes INCOMMON and calls the subroutine ADDIT from the program FORMPOOL.

```
PROGRAM SAPMZTST.
```

```
INCLUDE INCOMMON.
```

```
NUM1 = 2. NUM2 = 4.
```

```
PERFORM ADDIT(FORMPOOL).
```

```
NUM1 = 7. NUM2 = 11.
```

```
PERFORM ADDIT(FORMPOOL).
```

After starting SAPMZTST, the output appears as follows:

```
Sum of      2 and      4 is      6
```

```
Sum of      7 and     11 is     18
```

This example has the same function as the example in [Calling Internal Subroutines \[Page 447\]](#). In the present example, the data objects used in the programs must be declared as common parts because the subroutines ADDIT and OUT are called externally.

Passing Data by Parameters

You can pass data between calling programs and subroutines by using parameters.

- Parameters which are defined during the definition of a subroutine with the FORM statement are called **formal parameters**.
- Parameters which are specified during the call of a subroutine with the PERFORM statement are called **actual parameters**.

You can distinguish between different kinds of parameters:

- Input parameters are used to pass data to subroutines.
- Output parameters are used to pass data from subroutines.
- Input/output parameters are used to pass data to and from subroutines.

You define the parameters in the <pass> option of the FORM and PERFORM statements as follows:

Syntax

```
FORM <subr> [TABLES <formal table list>]
           [USING <formal input list>]
           [CHANGING <formal output list>]....

PERFORM <subr>[(<prog>)] [TABLES <actual table list>]
           [USING <actual input list>]
           [CHANGING <actual output list>]....
```

The options TABLES, USING, and CHANGING must be written in the shown sequence.

The parameters in the lists behind USING and CHANGING can be data objects of all types (see [Declaring Data \[Page 103\]](#)) and field symbols (see [Working with Field Symbols \[Page 336\]](#)). The parameters in the lists behind TABLES can be internal tables with or without a header line. You can transfer internal tables by using TABLES, USING, or CHANGING.

You can specify actual parameters with **variable** offset and length specifications (for more information about offset specifications, see [Specifying Offset Values for Data Objects \[Page 216\]](#)). You cannot do this for formal parameters. To refer to a part of the formal parameter, assign this part to a field symbol and then pass the field symbol (see [Working with Field Symbols \[Page 336\]](#)).

Offset specifications for actual parameters function as offset specifications for field symbols (see [Static ASSIGN with Offset Specifications \[Page 349\]](#)). You can select memory areas that lie outside the boundaries of the specified actual parameter.

For each **formal** parameter in a list behind USING and CHANGING in the FORM statement, you can specify different methods of passing data:

- **Calling by Reference** : During a subroutine call, only the address of the actual parameter is transferred to the formal parameters. The formal parameter has no memory of its own. You work with the field of the calling program within the

Passing Data by Parameters

subroutine. If you change the formal parameter, the field contents in the calling program also change.

[Passing by Reference \[Page 456\]](#)

- **Calling by Value** : During a subroutine call, the formal parameters are created as copies of the actual parameters. The formal parameters have memory of their own. Changes to the formal parameter have no effect on the actual parameter.

[Passing by Value \[Page 458\]](#)

- **Calling by Value and Result** : During a subroutine call, the formal parameters are created as copies of the actual parameters. The formal parameters have their own memory space. Changes to the formal parameters are copied to the actual parameter at the end of the subroutine.

[Passing by Value and Result \[Page 459\]](#)

Internal tables that are passed by TABLES, are always called by reference.

For information on how to specify the data types of formal parameters, see

[Typing Formal Parameters \[Page 461\]](#)

Specifying the data types of formal parameters is important for passing structured data to subroutines. How to pass structured data (field strings and internal tables) is explained especially in

[Passing Structures to Subroutines \[Page 463\]](#)

[Passing Internal Tables to Subroutines \[Page 466\]](#)

Passing by Reference

To pass data between calling programs and subroutines by reference, use USING or CHANGING for the <pass> option of the FORM and PERFORM statements as follows:

Syntax

FORM..... [USING <fi₁>... <fi_n>] [CHANGING <fo₁>... <fo_n>]...

PERFORM... [USING <ai₁>... <ai_n>] [CHANGING <ao₁>... <ao_n>]...

You specify the formal and actual parameters in the list behind USING and CHANGING without any addition.

The formal parameters in the form statement can have different names than the actual parameters <ai₁>... <ai_n> and <ao₁>... <ao_n> in the PERFORM statement. The first parameter of a list in the PERFORM statement is passed to the first parameter of the corresponding list in the FORM statement and so on.

For calling by reference, USING and CHANGING are equivalent. For documentation purposes, you should use USING for input parameters which are not changed in the subroutine, and CHANGING for output parameters which are changed in the subroutine.

Input parameters which are changed in the subroutine are also changed in the calling program. To prevent this, you must pass the parameter by value.

Assume a program FORMPOOL containing two subroutines, ADDIT and OUT:

```
PROGRAM FORMPOOL.
```

```
FORM ADDIT USING ADD_NUM1 ADD_NUM2 CHANGING ADD_SUM.
```

```
  ADD_SUM = ADD_NUM1 + ADD_NUM2.
```

```
  PERFORM OUT USING ADD_NUM1 ADD_NUM2 ADD_SUM.
```

```
ENDFORM.
```

```
FORM OUT USING OUT_NUM1 OUT_NUM2 OUT_SUM.
```

```
...WRITE: / 'Sum of',OUT_NUM1,'and',OUT_NUM2,'is',OUT_SUM.
```

```
ENDFORM.
```

Assume a calling program that calls ADDIT and OUT:

```
PROGRAM SAPMZTST.
```

```
DATA: NUM1 TYPE I,
```

```
      NUM2 TYPE I,
```

```
      SUM  TYPE I.
```

```
NUM1 = 2. NUM2 = 4.
```

```
PERFORM ADDIT(FORMPOOL) USING NUM1 NUM2 CHANGING SUM.
```

```
NUM1 = 7. NUM2 = 11.
```

```
PERFORM ADDIT(FORMPOOL) USING NUM1 NUM2 CHANGING SUM.
```

After starting SAPMZTST the output appears as follows:

Sum of	2 and	4 is	6
Sum of	7 and	11 is	18

Passing by Reference

This example has the same function as the example in [Declaring Data as Common Part \[Page 452\]](#). In the present example, the actual parameters NUM1, NUM2, and SUM are passed by reference from SAPMZTST to the formal parameters of the subroutine ADDIT. After changing ADD_SUM, the latter parameters are then passed to the formal parameters OUT_NUM1, OUT_NUM2, and OUT_SUM of the subroutine OUT.

Passing by Value

To ensure that an input parameter is not changed in the calling program, even if it is changed in the subroutine, you can pass data to a subroutine by value. For this, use USING for the <pass> option of the FORM and PERFORM statements as follows:

Syntax

FORM..... USING...VALUE(<fi>)..

PERFORM... USING.....<ai>..

By writing VALUE(<fi>) instead of <fi> for a formal input parameter in the list behind USING (for further information about USING, see [Passing by Reference \[Page 456\]](#)) in the FORM statement, the corresponding parameter is passed by value. A new local field <fi> is created when the subroutine is called by the PERFORM statement that has the same attributes as the actual field <ai> (for information about local fields, see [Defining Local Data Types and Objects in Subroutines \[Page 469\]](#)). The system processes this field independent from the reference field in the calling program.

Assume a program FORMPOOL that contains the subroutine FACT:

```
PROGRAM FORMPOOL.

FORM FACT USING VALUE(F_NUM) CHANGING F_FACT.
  F_FACT = 1.
  WHILE F_NUM GE 1.
    F_FACT = F_FACT * F_NUM.
    F_NUM = F_NUM - 1.
  ENDWHILE.
ENDFORM.
```

Assume a program SAPMZTST that calls the subroutine FACT:

```
PROGRAM SAPMZTST.

DATA: NUM TYPE I VALUE 5,
      FAC TYPE I VALUE 0.

PERFORM FACT(FORMPOOL) USING NUM CHANGING FAC.

WRITE: / 'Factorial of', NUM, 'is', FAC.
```

After starting SAPMZTST, the output appears as follows:

```
Factorial of      5 is      120
```

In this example, the factorial of a number NUM is calculated. The input parameter NUM is passed to the subroutine's formal parameter F_NUM. Although F_NUM is changed in the subroutine, the actual parameter NUM keeps its old value. The output parameter FAC is passed by reference.

Passing by Value and Result

Passing by Value and Result

If you want to return a changed output parameter from a subroutine to the calling program only after the subroutine has run successfully, use CHANGING for the <pass> option of the FORM and PERFORM statements as follows:

Syntax

```
FORM..... CHANGING...VALUE(<fii>)..
```

```
PERFORM... CHANGING.....<aii>..
```

By writing VALUE(<f_i>) instead of <f_i> for a formal output parameter in the list behind CHANGING (see [Passing by Reference \[Page 456\]](#)) in the FORM statement, the corresponding parameter is called by value and result. A new local field <f_i> is created when the subroutine is called by the PERFORM statement that has the same attributes as the actual field <a_i> (for further information about local fields, see [Defining Local Data Types and Objects in Subroutines \[Page 469\]](#)). The system processes this field independent from the reference field in the calling program.

Only when reaching the ENDFORM statement does the system pass the current value of <f_i> to <a_i>. If the subroutine processing is stopped because of a dialog message (for more information about messages, see [Handling Errors and Messages \[Page 1172\]](#)), the actual parameter <a_i> remains unchanged.

Terminating a subroutine by dialog messages is only meaningful when you write dialog programs (see [Dialog Mode \[Page 1309\]](#)). In report programs, dialog messages in subroutines terminate the entire program. Exceptions from this rule exist during the time event AT SELECTION SCREEN (see [Controlling the Flow of ABAP Programs by Events \[Page 1208\]](#)) or during interactive reporting (see [Messages in Lists \[Page 1195\]](#)). The selection screen and the tools for interactive reporting use ready-made dialog programs that are provided as keywords in ABAP.

When calling the subroutine with PERFORM, you can replace CHANGING by USING. But for documentation purposes, you should use the same word as in the FORM statement.

```
PROGRAM SAPMZTST.

DATA: OP1 TYPE I,
      OP2 TYPE I,
      RES TYPE I.

OP1 = 3.
OP2 = 4.

PERFORM MULTIP USING OP1 OP2 CHANGING RES.

WRITE: / 'After subroutine:',
       / 'RES=' UNDER 'RES=', RES.

FORM MULTIP USING VALUE(O1) VALUE(O2) CHANGING VALUE(R).
  R = O1 * O2.
  WRITE: / 'Inside subroutine:',
        / 'R=', R, 'RES=', RES.
ENDFORM.
```

After starting SAPMZTST the output appears as follows:

Inside subroutine:

R= 12 RES= 0

After subroutine:

RES= 12

In this example, an internal subroutine MULTIP is called from the calling program. The input parameters OP1 and OP2 are passed by value to the formal parameters O1 and O2. The output parameter RES is passed by value and result to the formal parameter R. By writing R and RES onto the screen from within the subroutine, it is demonstrated that RES has not changed its contents before the ENDFORM statement. After returning from the subroutine, its contents have changed.

Typing Formal Parameters

Typing Formal Parameters

To ensure that a formal parameter of a subroutine is of a certain type, you can specify that type in the FORM statement. To do this, enter TYPE <t> or LIKE <f> (see [Basic Form of the DATA Statement \[Page 120\]](#)) after the formal parameter in the list after TABLES, USING, or CHANGING. This type specification is optional.

When you call a subroutine with PERFORM, the system checks whether the types of the actual parameters in the PERFORM statement are compatible with the types assigned to the formal parameters. The compatibility rules are set out in the table below. There are no type conversions. If types are incompatible, the system outputs an error message during the syntax check for internal subroutine calls or reacts with a runtime error in the case of external subroutine calls.

The following compatibility rules apply for the typing of formal parameters:

Typing	Syntax check for actual parameters
No type specification, TYPE ANY	The system accepts actual parameters of any type. All attributes of the actual parameter are passed to the formal parameter.
TYPE TABLE	The system checks whether the actual parameter is an internal table. All attributes and the structure of the table are passed from the actual parameter to the formal parameter.
TYPE C, N, P, or X	The system checks whether the actual parameter is of type C, N, P, or X. The length of the parameter and its DECIMALS specification (for type P) are passed from the actual parameter to the formal parameter.
TYPE D, F, I, or T, LIKE <f>, TYPE <ud>	These types are fully determined. The system checks whether the data type of the actual parameter agrees completely with the type of the formal parameter.

(<ud> is user-defined)

```
REPORT SAPMZTST.
```

```
DATA:
```

```
  DATE1  TYPE D,      DATE2  TYPE T,
  STRING1(6) TYPE C,   STRING2(8) TYPE C,
  NUMBER1 TYPE P DECIMALS 2, NUMBER2 TYPE P,
  COUNT1  TYPE I,     COUNT2  TYPE I.
```

```
PERFORM TYPETEST USING DATE1 STRING1 NUMBER1 COUNT1.
```

```
SKIP.
```

```
PERFORM TYPETEST USING DATE2 STRING2 NUMBER2 COUNT2.
```

```
FORM TYPETEST USING NOW
```

```
  TXT TYPE C
  VALUE(NUM) TYPE P
  INT TYPE I.
```

```
DATA: T.
```

```
DESCRIBE FIELD NOW TYPE T.
```

```
WRITE: / 'Type of NOW is', T.
```

```
DESCRIBE FIELD TXT LENGTH T.
```

```
WRITE: / 'Length of TXT is', T.
```

```
DESCRIBE FIELD NUM DECIMALS T.  
WRITE: / 'Decimals of NUM are', T.  
DESCRIBE FIELD INT TYPE T.  
WRITE: / 'Type of INT is', T.  
ENDFORM.
```

After SAPMZTST, the output appears as follows:

```
TYPE of NOW is D  
Length of TXT is 6  
Decimals of NUM are 2  
Type of INT is I
```

```
TYPE of NOW is T  
Length of TXT is 8  
Decimals of NUM are 0  
Type of INT is I
```

An internal subroutine TYPETEST is called two times with different actual parameters. All actual and formal parameters are compatible and no error message occurs during the syntax check. If you would declare, for example, COUNT2 with TYPE F instead of TYPE I in the calling program, the syntax check would report an error because the formal parameter INT is specified with TYPE I. Note that, depending on their type specifications, the formal parameters can have different types and attributes for each call of the subroutine.

For more information, see the keyword documentation on FORM.

Passing Structures to Subroutines

Passing Structures to Subroutines

If you want to pass a structure to a subroutine and access the components of the structure in the subroutine, you must specify the type of the corresponding formal parameter (see [Typing Formal Parameters \[Page 461\]](#)). The data type used here must have the same structure as the structure.

With internal subroutines, you can use TYPE or LIKE to reference the structure of the structure you want to pass directly. With external subroutines, you must also specify the definition of the structure in the program which contains the subroutine. To do this, you can use any of the following:

- Include programs
- Type groups
- ABAP Dictionary structures

These are explained in the following examples:

Include programs

You can define the structure in an include program (see [Include Programs \[Page 441\]](#)). This method is suitable for structures which are only used in a few subroutines and only by a few developers.

An include program called DECLARE:

```
***INCLUDE DECLARE.
```

```
TYPES: BEGIN OF LINE,
       NAME(10) TYPE C,
       AGE(2) TYPE N,
       COUNTRY(3) TYPE C,
       END OF LINE.
```

A program called FORMPOOL includes the program DECLARE and contains a subroutine called COMPONENTS:

```
PROGRAM FORMPOOL.
```

```
INCLUDE DECLARE.
```

```
FORM COMPONENTS CHANGING VALUE(PERSON) TYPE LINE.
  WRITE: / PERSON-NAME, PERSON-AGE, PERSON-COUNTRY.
  PERSON-NAME = 'Mickey'.
  PERSON-AGE = '60'.
  PERSON-COUNTRY = 'USA'.
ENDFORM.
```

A program called SAPMZTST includes the program DECLARE and calls the subroutine COMPONENTS:

```
REPORT SAPMZTST.
```

```
INCLUDE DECLARE.
```

```
DATA WHO TYPE LINE.
```

```
WHO-NAME = 'Karl'. WHO-AGE = '10'. WHO-COUNTRY = 'D'.
```

PERFORM COMPONENTS(FORMPOOL) CHANGING WHO.

WRITE: / WHO-NAME, WHO-AGE, WHO-COUNTRY.

SAPMZTST then produces the following output:

KARL 10 D

MICKY 60 USA

The actual parameter WHO with the user-defined, structured data type LINE is passed to the formal parameter PERSON. The formal parameter PERSON is typed with TYPE LINE. Since LINE is a user-defined data type, the type of PERSON is completely specified. The subroutine accesses and changes the components of PERSON. They are then returned to the components of WHO in the calling program.

Type groups

You can define the structure in a type group (see [Type Groups \[Page 137\]](#)). This method is suitable for structures used in very large programs where several developers are working.

A type group called DECLA:

TYPE-POOL DECLA.

```
TYPES: BEGIN OF DECLA_LINE,
      NAME(10) TYPE C,
      AGE(2) TYPE N,
      COUNTRY(3) TYPE C,
      END OF DECLA_LINE.
```

A program called FORMPOOL uses the type group DECLA and contains a subroutine called COMPONENTS:

PROGRAM FORMPOOL.

TYPE-POOLS DECLA.

```
FORM COMPONENTS CHANGING VALUE(PERSON) TYPE DECLA_LINE.
  WRITE: / PERSON-NAME, PERSON-AGE, PERSON-COUNTRY.
  PERSON-NAME = 'Mickey'.
  PERSON-AGE = '60'.
  PERSON-COUNTRY = 'USA'.
ENDFORM.
```

A program called SAPMZTST uses the type group DECLA and calls the subroutine COMPONENTS:

REPORT SAPMZTST.

TYPE-POOLS DECLA.

DATA WHO TYPE DECLA_LINE.

WHO-NAME = 'Karl'. WHO-AGE = '10'. WHO-COUNTRY = 'D'.

PERFORM COMPONENTS(FORMPOOL) CHANGING WHO.

WRITE: / WHO-NAME, WHO-AGE, WHO-COUNTRY.

SAPMZTST then produces the following output:

Passing Structures to Subroutines

KARL 10 D

MICKEY 60 USA

This example works exactly like the one above, except that the type definition does not appear in an include program but in a type group called DECLA.

ABAP Dictionary structures

You can use table structures from the ABAP Dictionary. This method is always a possible option, since ABAP Dictionary structures can always be addressed.

A program called FORMPOOL contains a subroutine called COMPONENTS:

PROGRAM FORMPOOL.

FORM COMPONENTS CHANGING VALUE(CITIES) LIKE SPFLI.

WRITE: / CITIES-CITYFROM, CITIES-CITYTO.

CITIES-CITYFROM = 'New York'.

CITIES-CITYTO = 'San Francisco'.

ENDFORM.

A program called SAPMZTST calls the subroutine COMPONENTS:

REPORT SAPMZTST.

DATA FLIGHT LIKE SPFLI.

FLIGHT-CITYFROM = 'Berlin'. FLIGHT-CITYTO = 'London'.

PERFORM COMPONENTS(FORMPOOL) CHANGING FLIGHT.

WRITE: / FLIGHT-CITYFROM, FLIGHT-CITYTO.

SAPMZTST then produces the following output:

Berlin	London
New York	San Francisco

In the subroutine COMPONENTS, the formal parameter CITIES is typed with LIKE SPFLI. CITIES has the same structure as the table SPFLI in the ABAP Dictionary. In the program SAPMZTST, a structure FLIGHT is declared with the same structure, filled and passed to the components of the structure CITIES of the subroutine COMPONENTS. This subroutine processes the components and passes them to the calling program.

Passing Internal Tables to Subroutines

Passing with USING and CHANGING

You can pass internal tables as parameters in the list after USING or CHANGING in the FORM and PERFORM statements. If you want to access the components of the internal table, you must specify the type of the corresponding formal parameter (see [Typing Formal Parameters \[Page 461\]](#)). Otherwise, you can perform only line operations in the subroutine.

You also must distinguish between internal tables with or without header lines. For internal tables with header lines, you must specify the table body by using square brackets ([]) after the table name to distinguish it from the header line (see [Choosing a Table Type \[Page 267\]](#)).

With internal subroutines, you can use TYPE or LIKE to refer to the internal table you want to pass directly. With external subroutines, you must also define the structure in the program which contains the subroutine. To do this, proceed as you would for structures (see [Passing Structures to Subroutines \[Page 463\]](#)).

```

PROGRAM SAPMZTST.

DATA: BEGIN OF LINE,
      COL1 TYPE I,
      COL2 TYPE I,
      END OF LINE.

DATA ITAB LIKE LINE OCCURS 10.

PERFORM FILL CHANGING ITAB.
PERFORM OUT USING ITAB.

FORM FILL CHANGING F_ITAB LIKE ITAB.
  DATA F_LINE LIKE LINE OF F_ITAB.
  DO 3 TIMES.
    F_LINE-COL1 = SY-INDEX.
    F_LINE-COL2 = SY-INDEX ** 2.
    APPEND F_LINE TO F_ITAB.
  ENDDO.
ENDFORM.

FORM OUT USING VALUE(F_ITAB) LIKE ITAB.
  DATA F_LINE LIKE LINE OF F_ITAB.
  LOOP AT F_ITAB INTO F_LINE.
    WRITE: / F_LINE-COL1, F_LINE-COL2.
  ENDLOOP.
ENDFORM.

```

After starting SAPMZTST the output appears as follows:

```

  1      1
  2      4
  3      9

```

In this example, two internal subroutines, FILL and OUT, are defined and called. An internal table without header line ITAB is passed to the subroutines. It is passed to FILL by reference and it is passed to OUT by value. Note that in both subroutines a work area F_LINE is created as a local data object (for information

Passing Internal Tables to Subroutines

about local fields, see [Defining Local Data Types and Objects in Subroutines \[Page 469\]](#)). If ITAB would be a table with a header line, you would have to replace ITAB by ITAB[] in the PERFORM and FORM statements.

Passing with TABLES

You can pass all internal tables as parameters in the list after TABLES in the FORM and PERFORM statements. If you want to access the components of the table lines in the subroutine, you must specify the data types of the formal parameters (see [Passing Structures to Subroutines \[Page 463\]](#)). Otherwise, you can only perform operations on the whole line. Internal tables passed with TABLES are always called by reference.

If you pass an internal table with a header line, the table body and the table work area are passed to the subroutine. If you pass an internal table without a header line, a header line is created automatically as a local data object in the subroutine (for further information about local fields, see [Defining Local Data Types and Objects in Subroutines \[Page 469\]](#)).

```
PROGRAM SAPMZTST.

TYPES: BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
      END OF LINE.

DATA: ITAB TYPE LINE OCCURS 10 WITH HEADER LINE,
      JTAB TYPE LINE OCCURS 10.

PERFORM FILL TABLES ITAB.
MOVE ITAB[] TO JTAB.
PERFORM OUT TABLES JTAB.

FORM FILL TABLES F_ITAB LIKE ITAB[].
  DO 3 TIMES.
    F_ITAB-COL1 = SY-INDEX.
    F_ITAB-COL2 = SY-INDEX ** 2.
    APPEND F_ITAB.
  ENDDO.
ENDFORM.

FORM OUT TABLES F_ITAB LIKE JTAB.
  LOOP AT F_ITAB.
    WRITE: / F_ITAB-COL1, F_ITAB-COL2.
  ENDLOOP.
ENDFORM.
```

After starting SAPMZTST the output appears as follows:

```
1      1
2      4
3      9
```

In this example, an internal table ITAB is declared with a header line and an internal table JTAB is declared without a header line. ITAB is passed to the subroutine FILL, where it is filled using the table work area F_ITAB. After copying the table body of ITAB to JTAB in the calling program, JTAB is passed to the subroutine OUT. Note that, in this case, an actual table without a header line is

Passing Internal Tables to Subroutines

passed to a formal table with a header line and that the table work area F_ITAB is used in the subroutine.

Defining Local Data Types and Objects in Subroutines

Defining Local Data Types and Objects in Subroutines

Local data types and objects are visible only in the procedure where they are declared. For subroutines, you can distinguish between

- dynamic data types and objects that exist only while the subroutine is performed,
[Defining Dynamic Local Data Types and Objects \[Page 470\]](#)
- static data objects that exist beyond a subroutine call and keep their value until the next call of the same subroutine,
[Defining Static Local Data Objects \[Page 471\]](#)
- explicitly defined local data objects that can be used to preserve the values of global data objects.
[Defining Local Data Objects Explicitly \[Page 472\]](#)

A special kind of local data are copies of global data on a local data stack. They are defined and accessed using field symbols (for more details on this subject, see [Assigning a Local Copy of a Global Field \[Page 359\]](#)).

Defining Dynamic Local Data Types and Objects

You can create local data types and data objects within subroutines using the TYPES and DATA statements as described in [Creating Data Objects and Data Types \[Page 118\]](#). Such types and objects are newly defined for each subroutine call and deleted after exiting the subroutines.

Every subroutine has its own local naming space. If you define a local data type or object with the same name as a global data type or object, you cannot access the global data type or object within the subroutine. Local data types or data objects hide identically named global data types or objects. This means that if you use the name of a data type or object in the subroutine, you always address a locally declared object - if this exists - and otherwise a globally declared one.

```
PROGRAM SAPMZTST.  
TYPES WORD(10) TYPE C.  
DATA TEXT TYPE WORD.  
TEXT = '1234567890'. WRITE / TEXT.  
PERFORM DATATEST.  
WRITE / TEXT.  
FORM DATATEST.  
  TYPES WORD(5) TYPE C.  
  DATA TEXT TYPE WORD.  
  TEXT = 'ABCDEFGHJK'. WRITE / TEXT.  
ENDFORM.
```

After starting SAPMZTST, the output appears as follows:

```
1234567890  
ABCDE  
1234567890
```

In this example, a data type WORD and a data object TEXT with type WORD are declared globally in the calling program. After assigning a value to TEXT and writing it onto the screen, the internal subroutine DATATEST is called. Inside the subroutine, a data type WORD and a data object TEXT with type WORD are declared locally. They hide the global type and object. Only after exiting the subroutine the global definitions are valid again.

If you want to prevent the hiding of global data types and objects, you must assign different names to the local types and objects. For example, you can start all local names with a prefix 'F_'.

Defining Static Local Data Objects

Defining Static Local Data Objects

If you want to keep the value of a local data object after exiting the subroutine, you must use the `STATICS` statement instead of the `DATA` statement for its declaration (see [The `STATICS` Statement](#) [\[Page 128\]](#)). With `STATICS` you declare a data object that is globally defined, but only locally visible from the subroutine in which it is defined.

```
PROGRAM SAPMZTST.

PERFORM DATATEST1.
PERFORM DATATEST1.

SKIP.

PERFORM DATATEST2.
PERFORM DATATEST2.

FORM DATATEST1.
  TYPES F_WORD(5) TYPE C.
  DATA F_TEXT TYPE F_WORD VALUE 'INIT'.
  WRITE F_TEXT.
  F_TEXT = '12345'.
  WRITE F_TEXT.
ENDFORM.

FORM DATATEST2.
  TYPES F_WORD(5) TYPE C.
  STATICS F_TEXT TYPE F_WORD VALUE 'INIT'.
  WRITE F_TEXT.
  F_TEXT = 'ABCDE'.
  WRITE F_TEXT.
ENDFORM.
```

After starting `SAPMZTST`, the output appears as follows:

INIT 12345 INIT 12345

INIT ABCDE ABCDE ABCDE

In this example, two similar subroutines `DATATEST1` and `DATATEST2` are defined. In `DATATEST2`, the `STATICS` statement is used instead of the `DATA` statement to declare the data object `F_TEXT`. During each call of `DATATEST1`, `F_TEXT` is initialized again, but it keeps its value for `DATATEST2`. The `VALUE` option of the `STATICS` statement functions only during the first call of `DATATEST2`.

Defining Local Data Objects Explicitly

To preserve the value of a global data object from being changed inside a subroutine, use the LOCAL statement as follows:

Syntax

LOCAL <f>.

You can use this statement only between a FORM and ENDFORM statement. With LOCAL, you can preserve the values of global data objects which cannot be hidden by a data declaration inside the subroutine (see [Defining Dynamic Local Data Types and Objects \[Page 470\]](#)).

For example, you cannot declare a table work area that is defined by the TABLES statement (see [The TABLES Statement \[Page 129\]](#)) with another TABLES statement inside a subroutine. If you want to use the table work area locally, but preserve its contents outside the subroutine, you must use the LOCAL statement.

For more information about this statement, see the keyword documentation on LOCAL.

Assume a program FORMPOOL that contains two subroutines TABTEST1 and TABTEST2 as follows:

```
PROGRAM FORMPOOL.

TABLES SFLIGHT.

FORM TABTEST1.
  SFLIGHT-PLANETYPE = 'A310'.
  SFLIGHT-PRICE = '150.00'.
  WRITE: / SFLIGHT-PLANETYPE, SFLIGHT-PRICE.
ENDFORM.

FORM TABTEST2.
  LOCAL SFLIGHT.
  SFLIGHT-PLANETYPE = 'B747'.
  SFLIGHT-PRICE = '500.00'.
  WRITE: / SFLIGHT-PLANETYPE, SFLIGHT-PRICE.
ENDFORM.
```

Assume a program SAPMZTST that calls TABTEST1 and TABTEST2 as follows:

```
PROGRAM SAPMZTST.

TABLES SFLIGHT.

PERFORM TABTEST1(FORMPOOL).
WRITE: / SFLIGHT-PLANETYPE, SFLIGHT-PRICE.

PERFORM TABTEST2(FORMPOOL).
WRITE: / SFLIGHT-PLANETYPE, SFLIGHT-PRICE.
```

After starting SAPMZTST, the output appears as follows:

A310	150.00
A310	150.00
B747	500.00
A310	150.00

Defining Local Data Objects Explicitly

In this example, a table work area SFLIGHT is created by referring to the ABAP Dictionary structure SFLIGHT. Different values are assigned to the table work area SFLIGHT in TABTEST1 and TABTEST2. While the values assigned in TABTEST1 are globally valid, the values assigned in TABTEST2 are only locally valid.

Terminating Subroutines

To terminate the processing of a subroutine, you can proceed in a similar way as for terminating a loop by using the EXIT or CHECK statements (see [Terminating Loops \[Page 256\]](#)).

- Use EXIT to terminate a subroutine unconditionally.

[Terminating Subroutines Unconditionally \[Page 475\]](#)

- Use CHECK to terminate a subroutine according to a condition.

[Terminating Subroutines Conditionally \[Page 476\]](#)

If you terminate a subroutine using EXIT or CHECK, the system terminates the processing of the subroutine at this point, passes the parameters, and continues with the statement after the PERFORM statement.

If you use EXIT or CHECK inside loops in a FORM routine, the termination conditions for loops apply, as described in [Terminating Loops \[Page 256\]](#).

The EXIT and CHECK statements work differently for terminating loops, but in the same way for terminating subroutines.

Terminating Subroutines Unconditionally

Terminating Subroutines Unconditionally

To terminate a subroutine without any condition, use the EXIT statement as follows:

Syntax

EXIT.

After an EXIT statement, the system immediately exits the subroutine and resumes the processing after the PERFORM statement.

```
PROGRAM SAPMZTST.  
PERFORM TERMINATE.  
WRITE 'The End'.  
FORM TERMINATE.  
  WRITE '1'.  
  WRITE '2'.  
  WRITE '3'.  
  EXIT.  
  WRITE '4'.  
ENDFORM.
```

After starting SAPMZTST, the output appears as follows:

1 2 3 The End

In this example, the subroutine TERMINATE is terminated after the third WRITE statement.

Terminating Subroutines Conditionally

To terminate a subroutine conditionally, use the CHECK statement as follows:

Syntax

CHECK <condition>.

If the condition is **not** satisfied, the system leaves the subroutine and resumes the processing after the PERFORM statement. For <condition>, you can use any logical expression described in [Programming Logical Expressions \[Page 235\]](#).

```
PROGRAM SAPMZTST.
DATA: NUM1 TYPE I,
      NUM2 TYPE I,
      RES TYPE P DECIMALS 2.
NUM1 = 3. NUM2 = 4.
PERFORM DIVIDE USING NUM1 NUM2 CHANGING RES.
NUM1 = 5. NUM2 = 0.
PERFORM DIVIDE USING NUM1 NUM2 CHANGING RES.
NUM1 = 2. NUM2 = 3.
PERFORM DIVIDE USING NUM1 NUM2 CHANGING RES.
FORM DIVIDE USING N1 N2 CHANGING R.
  CHECK N2 NE 0.
  R = N1 / N2.
  WRITE: / N1, '/', N2, '=', R.
ENDFORM.
```

After starting SAPMZTST, the output appears as follows:

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

In this example, the system leaves the subroutine DIVIDE during the second call after the CHECK statement because the value of N2 is zero.

Function Modules

Function modules are special external subroutines stored in a central library. The R/3 System provides numerous predefined function modules that you can call from your ABAP programs, and you can create your own function modules.

The main difference between a function module and a normal ABAP subroutine is a clearly defined interface for passing data to and from the function modules. Declaring data as common parts is not possible for function modules. The calling program and the called function module have separate work areas for ABAP Dictionary tables.

The interface facilitates and standardizes the passing of input and output parameters. For example, you can assign default values to the input parameters of function modules. You can decide whether to pass data by value or by reference (for more information about passing data, see [Passing Data by Parameters \[Page 454\]](#)). The interface also supports exception handling. With exceptions, you can handle possible errors.

You use the ABAP Workbench to call, create, and maintain function modules. For example, you can test a function module by calling it from a transaction screen outside an ABAP program.

You can combine several function modules to form a function group in the function library. Since you can define global data that can be accessed from all function modules of one function group, it is reasonable to include function modules that operate with the same data, for example an internal table, in one function group.

The function group is the main program for the function modules contained in the group and for further program modules, as for example, subroutines. Function groups are stored in the program library with the attribute **type F**. See also [Function Groups \[Page 497\]](#) for an overview how function modules are organized.

The following topics explain the principles of working with existing function modules and how to create new ones. For detailed information about function modules and the function library, see the documentation [BC ABAP Workbench Tools \[Ext.\]](#).

[Working with Existing Function Modules \[Page 478\]](#)

[Creating and Programming Function Modules \[Page 491\]](#)

Working with Existing Function Modules

The following topics describe

[Calling a List of Available Function Modules \[Page 479\]](#)

[Displaying Attributes of Function Modules \[Page 481\]](#)

[Testing Function Modules \[Page 487\]](#)

[Calling Function Modules \[Page 488\]](#)

Calling a List of Available Function Modules

Calling a List of Available Function Modules

To obtain a list of available function modules:

1. From the *ABAP Development Workbench* screen, choose *Development* → *Function Library* or select *Function Library* in the application toolbar.

The *Function Library: Maintain Function Modules* screen displays.

The screenshot shows the 'Function Library: Maintain Function Modules' screen. At the top is a toolbar with icons for navigation and actions like 'Rename' and 'Reassign'. Below the toolbar, there is a 'Function module' text field and two buttons: 'Create' and 'Test'. A section titled 'Object components' contains a list of radio buttons: 'Administration' (selected), 'Import/export parameter interface', 'Table parameters/exceptions interface', 'Documentation', 'Source code', 'Global data', and 'Main program'. At the bottom of this section are two buttons: 'Display' and 'Change'.

2. Choose *Find*.

The *ABAP Repository Information System: Function Module* screen displays.

3. Enter all the search criteria you know, for example `string_sp*`, and choose *Enter*.

Calling a List of Available Function Modules

Standard selections

Function module	STRING_SP*	
Short description		
Function group		
Development class		

☐ Take into account generated function modules

A list of all function modules matching the entries you made displays.

CSTR	GROSSMANN	String functions for type C fields
<input type="checkbox"/> STRING_SPLIT		Splits a string into header and tail, according to a delimiter

4. Via this list, select the desired function module by clicking the corresponding check box.

Now, for example

- To change the function module, choose *Change* (for further information about this topic, see the documentation [BC ABAP Workbench Tools \[Ext.\]](#)).
- To display the documentation on the function module, choose *Display* (for further information about this topic, see [Displaying Attributes of Function Modules \[Page 481\]](#)).
- To test the function module, choose *Execute* (for further information about this topic, see [Testing Function Modules \[Page 487\]](#)).

Displaying Attributes of Function Modules

Displaying Attributes of Function Modules

To display the attributes of a function module, enter the name of the function module onto the *Function Library: Maintain Function Modules* screen. For example:

Function module:

Create

Test

Object components

- ☒ Administration
- ☐ Import/export parameter interface
- ☐ Table parameters/exceptions interface
- ☐ Documentation
- ☐ Source code
- ☐ Global data
- ☐ Main program

Display Change

Then, select the radio button of the attribute you want to display and choose *Display*. If you want to change an attribute, you can choose *Change* on the *ABAP Function Library: Initial Screen* or *Display* <-> *Change* on the following screen.

Important attributes are, for example:

[Documentation \[Page 482\]](#)

[Interfaces \[Page 483\]](#)

[Source Code \[Page 486\]](#)

Documentation

The *Documentation* screen appears as follows:

Short text:
Splits a string into header and tail, according to a delimiter

Parameter name		Parameter type
DELIMITER	String which is used as delimiter	I
STRING	Character string to be demounted	I
HEAD	Head of STRING in front of DELIMITER	E
TAIL	Tail of STRING after DELIMITER	E

Exception	Short text
NOT_FOUND	Delimiter string not found
NOT_VALID	Invalid delimiter string
TOO_LONG	DELIMITER too long (more than 40 characters)
TOO_SMALL	Output field HEAD or TAIL too short

The documentation describes the purpose of the function module, lists the parameters for passing data to and from the module, and the exceptions. The parameters of the parameter type I are import parameters, which are used to pass data to the function module. Parameters of the parameter type E are export parameters, which are used to pass data from the function module to the calling program. Exceptions describe error scenarios which can occur in function modules.

Interfaces

Interfaces

When you select *Import/export parameter interface*, you see the following screen:

Import parameter	Reference field	Reference type	Proposal	Optional	Reference
DELIMITER				<input type="checkbox"/>	<input type="checkbox"/>
STRING				<input type="checkbox"/>	<input type="checkbox"/>
				<input type="checkbox"/>	<input type="checkbox"/>
				<input type="checkbox"/>	<input type="checkbox"/>
				<input type="checkbox"/>	<input type="checkbox"/>
				<input type="checkbox"/>	<input type="checkbox"/>

Export parameters	Reference field	Reference type	Reference
HEAD			<input type="checkbox"/>
TAIL			<input type="checkbox"/>
			<input type="checkbox"/>
			<input type="checkbox"/>
			<input type="checkbox"/>
			<input type="checkbox"/>

CHANGING parameter	Reference field	Reference type	Proposal	Optional	Reference
				<input type="checkbox"/>	<input type="checkbox"/>
				<input type="checkbox"/>	<input type="checkbox"/>
				<input type="checkbox"/>	<input type="checkbox"/>
				<input type="checkbox"/>	<input type="checkbox"/>
				<input type="checkbox"/>	<input type="checkbox"/>
				<input type="checkbox"/>	<input type="checkbox"/>

When you select *Table parameters/exceptions interface*, you see the following screen:

Table parameters	Ref. structure	Reference type	Optional
			<input type="checkbox"/>
			<input type="checkbox"/>
			<input type="checkbox"/>
			<input type="checkbox"/>
			<input type="checkbox"/>
			<input type="checkbox"/>
			<input type="checkbox"/>
			<input type="checkbox"/>
			<input type="checkbox"/>
			<input type="checkbox"/>

Exception
NOT_FOUND
NOT_VALID
TOO_LONG
TOO_SMALL

These screens list all the formal parameters used to pass data in function modules. The procedure for passing parameters is similar to the procedure for subroutines (see [Passing Data by Parameters \[Page 454\]](#)). If the checkbox *Reference* is marked, the parameter is passed by reference. Otherwise, if the checkbox is not marked, the parameter is passed by value.

- Import parameters correspond to the formal input parameters of subroutines. They pass data from the calling program to the function module. Import parameters cannot be overwritten, even when passed by reference.
- Export parameters correspond to the formal output parameters of subroutines. They pass data from the function module back to the calling program.
- Changing parameters are passed by reference or by value and result (see [Passing by Value and Result \[Page 459\]](#)). Changing parameters act simultaneously as import and export parameters. They change the value passed to the function module and return it to the calling program.
- Table parameters are internal tables. Internal tables are treated like changing parameters and are always passed by reference.
- Exceptions are used to handle error scenarios which can occur in function modules. The calling program checks whether any errors have occurred and then takes action accordingly.

Import parameters, changing parameters, and table parameters can be *Optional*. This means that you can omit the corresponding actual parameter when you call the function in the calling program. If the parameter is optional and the actual parameter is not specified, you can specify a default value for use in the function module. Export parameters are always optional.

Interfaces

As with subroutines (see [Typing Formal Parameters \[Page 461\]](#)), you can specify the data types of the formal parameters in the field *Reference type*. In the field *Ref. structure*, you can specify ABAP Dictionary reference structures or fields. Then, the system checks the current parameter against the structure or field at runtime.

Source Code

The ABAP Editor screen displays the ABAP source code of the function module. You can work with the source code in the same way as described for normal ABAP programs in [Open Programs via Forward Navigation \[Page 69\]](#).

Testing Function Modules

Testing Function Modules

You can test a function module without calling it from an ABAP program via the *Function Library: Maintain Function Modules* screen by choosing *Single test*. You can assign values to the import parameters on the *Test Function Modules* screen.

Test for function group	CSTR
Function module	STRING_SPLIT
Upper/lower case	<input type="checkbox"/>

Import parameters	Value
DELIMITER	-
STRING	Function-Module

After choosing *Execute*, the *Test Function Module: Result Screen* appears:

Runtime: 2.747 Microseconds	
-----------------------------	--

Import parameters	Value
DELIMITER	-
STRING	FUNCTION-MODULE

Export parameters	Value
HEAD	FUNCTION
TAIL	MODULE

Now you can see that the function module STRING_SPLIT works in the same way as the ABAP keyword SPLIT (see [Splitting Character Strings \[Page 214\]](#)). The value of the import parameter STRING "FUNCTION-MODULE" is split at the DELIMITER "-" into "FUNCTION" and "MODULE", which are assigned to the export parameters HEAD and TAIL. Note that the contents of STRING is changed to uppercase because the *Upper/lowercase* checkbox on the *Test Function Modules* screen is blank.

Calling Function Modules

To call a function module from an ABAP program, use the CALL statement as follows:

Syntax

```
CALL FUNCTION <module>
  [EXPORTING f1 = a1.... fn = an]
  [IMPORTING f1 = a1.... fn = an]
  [CHANGING f1 = a1.... fn = an]
  [TABLES f1 = a1.... fn = an]
  [EXCEPTIONS e1 = r1.... en = rn [ERROR_MESSAGE = rE]
    [OTHERS = ro]].
```

You can specify the name of the function module <module> as a literal or as a variable. Parameters are passed to and from the function module by explicitly assigning the actual parameters to the formal parameters in the lists after the EXPORTING, IMPORTING, TABLES, or CHANGING options.

The assignments must always have the form
<formal parameter> = <actual parameter>

- The EXPORTING option enables you to pass actual parameters a_i to the formal input parameters f_i. The formal parameters must be declared as **import** parameters in the function module.
- The IMPORTING option enables you pass to formal **output** parameters f_i to the actual parameters a_i. The formal parameters must be declared as **export** parameters in the function module.
- The CHANGING option enables you to pass actual parameters a_i to the formal parameters f_i and after processing the function module, the system passes the (changed) formal parameters f_i back to the actual parameters a_i. The formal parameters must be declared as changing parameters in the function module.
- The TABLES option enables you to pass internal tables between the actual and formal parameters. With this option, internal tables are always passed by reference.

The parameters of the EXPORTING, IMPORTING, and CHANGING options can be data objects of any type. The function of these options is similar to the USING and CHANGING options of the FORM and PERFORM statements for subroutines (see [Passing Data by Parameters \[Page 454\]](#)). The parameters of the TABLES option must be internal tables. The TABLES option corresponds to the TABLES option of the FORM and PERFORM statements (see [Passing Internal Tables to Subroutines \[Page 466\]](#)).

By using the EXCEPTIONS option, you can handle exceptions that are raised during the processing of the function module. Exceptions are special parameters of function modules. How to define and raise exceptions is explained in [Creating and Programming Function Modules \[Page 491\]](#). If an exception e_i is raised, the system stops performing the function module and does not pass any values from the function module to the program, except those that are passed by reference. If e_i is specified in the EXCEPTION option, the calling program handles the exception by assigning r_i to SY-SUBRC. You must specify r_i as a number literal.

Calling Function Modules

With the specification of `ERROR_MESSAGE` in the exception list you can influence the message handling of function modules. Normally, messages in function modules should be called only via the exception handling (Syntax `MESSAGE.... RAISING`, see [Creating and Programming Function Modules \[Page 491\]](#)). With `ERROR_MESSAGE` you can force the system to treat messages that are called without the `RAISING` option in a function module as follows:

- Messages of classes S, I, and W are ignored (but put to the protocol for background jobs).
- Messages of classes E and A stop the execution of the function module as if the exception `ERROR_MESSAGE` was raised (`SY-SUBRC` is set to `rE`).

You can use `OTHERS` in the `EXCEPTION` list to handle all exceptions that are not specified explicitly in the list.

You can use the same number `ri` for several exceptions.

The simplest way to include the call of a function module in your program is to choose *Edit → Insert statement...* via the ABAP Editor screen. On the dialog window that appears, you can select *Call Function* and enter the name of the function module that you want to call.



When you then choose *Enter*, the system inserts the `CALL` statement with all options that are possible for the specified function module into your program code.

For the function module `STRING-SPLIT`, the inserted code appears as follows:

```
CALL FUNCTION 'STRING_SPLIT'
  EXPORTING
    DELIMITER =
    STRING    =
  IMPORTING
    HEAD     =
    TAIL     =
  EXCEPTIONS
    NOT_FOUND = 01
    NOT_VALID = 02
    TOO_LONG  = 03
    TOO_SMALL = 04.
```

Now, you can change this code according to your needs.

```
REPORT SAPMZTST.

DATA: TEXT(20),
      FRONT(20),
      END(20).

TEXT = 'Testing:String_Split'.

CALL FUNCTION 'STRING_SPLIT'
  EXPORTING DELIMITER = ':' STRING = TEXT
  IMPORTING HEAD = FRONT TAIL = END
  EXCEPTIONS NOT_FOUND = 1 OTHERS = 2.

CASE SY-SUBRC.
  WHEN 1.      WRITE / 'Not found'.
```

```

        WHEN 2.          WRITE / 'Other errors'.
        WHEN OTHERS.     WRITE: / FRONT, / END.
    ENDCASE.

```

After starting SAPMZTST, the output appears as follows:

```

Testing
String_Split.

```

In this example, the function module `STRING_SPLIT` is called. By using the `EXPORTING` option, the actual parameters `:` and `TEXT` are passed to the formal parameters `DELIMITER` and `STRING`. In the case of successful completion of the procedure, the formal parameters `HEAD` and `TAIL` are passed to the actual parameters `FRONT` and `END` by using the `IMPORTING` option.

If exception `NOT_FOUND` is raised in `STRING_SPLIT`, the value of `SY-SUBRC` is set to 1. If another exception is raised, the value of `SY-SUBRC` is set to 2. The exceptions are handled with a `CASE` statement. If, for example, a literal that is not contained in `TEXT` is assigned to the formal parameter `DELIMITER` in the `EXPORTING` list, the output would appear as follows:

Not found

Creating and Programming Function Modules

The following topics describe how to create a simple function module. For more information about this subject, see the documentation [BC ABAP Workbench Tools \[Ext.\]](#).

[Creating Function Modules \[Page 492\]](#)

[Programming Function Modules \[Page 494\]](#)

Creating Function Modules

To create a new function module, proceed as follows:

1. Via the *Function Library: Maintain Function Modules* screen, enter the name of your function module and choose *Create*.

Function module	MY_DIVIDE	Create
-----------------	-----------	--------

2. The *Function Module Create: Administration* screen appears. On this screen, you must enter a function group <fgrp> for your module. You can also enter a short text for your module.

All function modules that belong to one function group are combined in one ABAP program SAPL<fgrp> ([Function Groups \[Page 497\]](#)). Create a function group via the *Function Library: Maintain Function Modules* screen by choosing *Goto → Function groups → Create group* or simply by double clicking on the name of a new function group (see the documentation [BC ABAP Workbench Tools \[Ext.\]](#)).

Classification	
Function group	habo
Application	
Short text	Example for ABAP/4 User Handbook: Division of two Numbers

Then save the function module by choosing *Save* and leave the screen by choosing *Back*.

3. Define the interface via the *Function Library: Maintain Function Modules* screen by selecting the *Import/export parameter interface* radiobutton and choosing *Change*. On the *Function Module Change: Import/Export Parameters* screen, enter your import, export, or changing parameters.

Import parameter	Reference field	Reference type	Proposal	Optional	Reference
N1				<input type="checkbox"/>	<input type="checkbox"/>
N2				<input type="checkbox"/>	<input type="checkbox"/>

Export-Parameter	Reference field	Reference type	Reference
R			<input type="checkbox"/>

To enter your exceptions, choose *Goto → Tab./exc. interface*. Via the *Function Module Change: Table Parameters/ Exceptions* screen, you can enter names for your exceptions.

Exception
DIV_ZERO

Save your interfaces by choosing *Save* and leave the screen by choosing *Back*.

4. Enter your source code via the *Function Library: Maintain Function Modules* screen by selecting the *Source code* radio button and choosing *Change*. The ABAP Editor appears, containing your program. The defined parameters appear in commented mode.

Creating Function Modules

1	function my_divide.
2	***-----
3	***"Local interface:"
4	*** IMPORTING
5	*** N1
6	*** N2
7	*** EXPORTING
8	*** R
9	*** EXCEPTIONS
10	*** DIV_ZERO
11	***-----
12	
13	
14	
15	
16	
17	endfunction.

Now, you can enter and save your program code (see [Programming Function Modules \[Page 494\]](#))

5. Activate your function module via the *Function Library: Maintain Function Modules* screen by choosing *Function Module* → *Activate* and test it as described in [Testing Function Modules \[Page 487\]](#).

Programming Function Modules

To program a function module, you must include your statements between the FUNCTION and ENDFUNCTION statements as follows:

Syntax

```
FUNCTION <module>.
```

```
    <statements>
```

```
ENDFUNCTION.
```

To enter the program code, proceed in the same way as for normal subroutines, but with the following exceptions:

Handling Data in Function Modules

You must not declare the export and import parameters in the source code of function module. The system performs this task for you in an INCLUDE program. It inserts a list of defined parameters as comment lines into the source code (see step 4 in [Creating Function Modules \[Page 492\]](#)).

You can declare local data types and objects in function modules as for subroutines (see [Defining Local Data Types and Objects in Subroutines \[Page 469\]](#)).

You open the ABAP Editor for the INCLUDE program L<fgrp>TOP via the *Function Library: Maintain Function Modules* screen by selecting *Global Data*. This INCLUDE program is included as the first statement in the program SAPL<fgrp>. SAPL<fgrp> is the main program of a function group <fgrp> which comprises all function modules of this group. The single modules again are contained in INCLUDE programs (see [Function Groups \[Page 497\]](#)).

You can write data declarations with the TYPES and DATA statements into L<fgrp>TOP. This data is global to all modules of one function group and local for this group. The system creates them as soon as the first module is called and always keeps the values of the last module call.

Calling Subroutines from Function Modules

You can call different subroutines from function modules.

- You can write the program code of internal subroutines directly behind the ENDFUNCTION statement of a module. This subroutines can be called from all modules of the function group. But to provide more clarity, you should call such subroutines only from the function module behind which it is written.
- To create internal subroutines that can be called from all modules of one function group <fgrp>, use special INCLUDE programs L<fgrp>F<xx>. You can open these INCLUDE programs easily by double clicking their names in the main program SAPL<fgrp> after selecting *Main program* on the *Function Library: Maintain Function Modules* screen. For more information about this topic, see the documentation [BC ABAP Workbench Tools \[Ext.\]](#).
- You can call any external subroutines.

Raising Exceptions

To raise exceptions, ABAP provides two statements that can only be used in function modules:

Syntax

Programming Function Modules

RAISE <exc>.

MESSAGE..... RAISING <exc>.

The effect of these statements depends on whether the calling program will handle the exception itself or leave the exception handling to the system. If the name of the exception <exc> or OTHERS is specified in the EXCEPTION option of the CALL FUNCTION statement (see [Calling Function Modules \[Page 488\]](#)), the calling program itself handles the exception.

If the system handles the exception,

- the RAISE statement terminates the program and switches to debugging mode.
- the MESSAGE..... RAISING statement displays a message. The system continues processing according to the message type. You must specify the MESSAGE-ID in the INCLUDE program L<fgrp>TOP in the first statement. For information about message handling in reports see [Messages in Lists \[Page 1195\]](#), and for information about message handling in transaction programming see [Handling Errors and Messages \[Page 1172\]](#).

If the calling program handles the exception itself, both statements return the program control directly to that program and the system does not pass any data from the function module to the program, except those that are called by reference. The MESSAGE..... RAISING statement does not display a message, but fills the following system fields which you can access in your program:

- SY-MSGID (Message-Id)
- SY-MSGTY (Message type)
- SY-MSGNO (Message number)
- SY-MSGV1 to SY-MSGV4 (contents of fields <f1> to <f4>, included in a message).

(See the keyword documentation of MESSAGE).

To handle messages that are called without the RAISING addition in a function module, use the special exception ERROR_MESSAGE when calling the function module (see [Calling Function Modules \[Page 488\]](#)).

Assume the following function module:

FUNCTION MY_DIVIDE.

```

*|-----
*| Local interface:
*|   IMPORTING
*|     N1
*|     N2
*|   EXPORTING
*|     R
*|   EXCEPTIONS
*|     DIV_ZERO
*|-----
      IF N2 EQ 0.
        RAISE DIV_ZERO.
      ELSE.
```

```
R = N1 / N2.  
ENDIF.
```

```
ENDFUNCTION.
```

In this function module, if N2 is not equal to zero, N1 is divided by N2. Otherwise, the exception DIV_ZERO is raised.

Assume the following program SAPMZTST that calls MY_DIVIDE:

```
REPORT SAPMZTST.  
  
DATA: RES TYPE P DECIMALS 2.  
  
CALL FUNCTION 'MY_DIVIDE'  
  EXPORTING  N1 = 5 N2 = 4  
  IMPORTING  R = RES  
  EXCEPTIONS DIV_ZERO = 11.  
  
IF SY-SUBRC EQ 0.  
  WRITE: / 'Result =', RES.  
ELSE.  
  WRITE 'Division by zero'.  
ENDIF.
```

After starting SAPMZTST, the output appears as follows:

RESULT = 1.25

If you replace N2 = 4 by N2 = 0 in the EXPORTING list, the SAPMZTST handles the exception DIV_ZERO by assigning 11 to SY-SUBRC, and the output is:

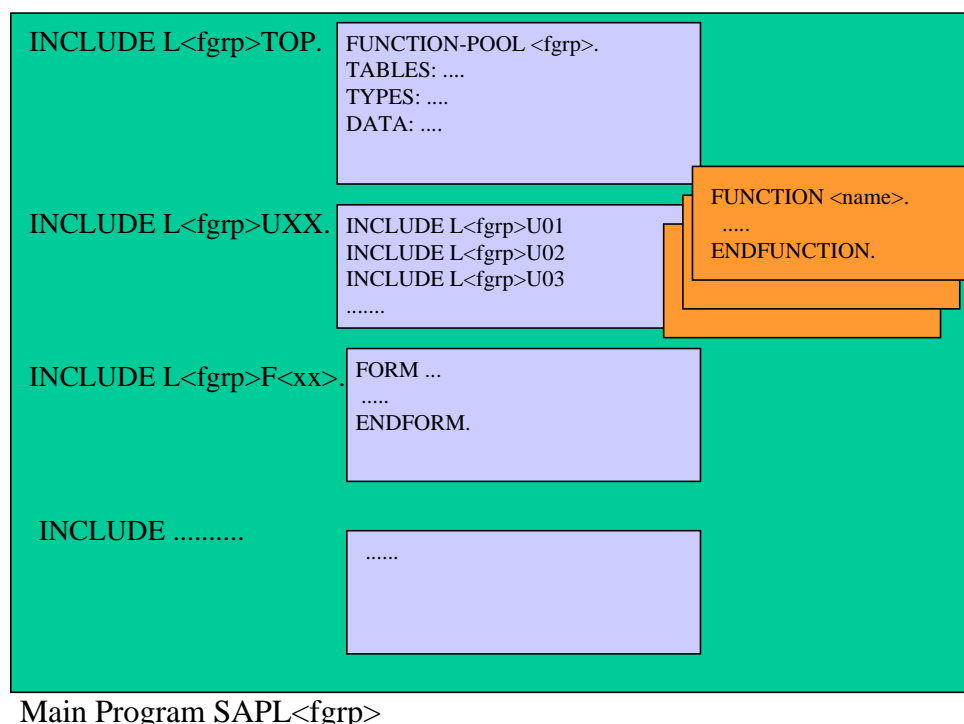
Division by zero

Function Groups

Function Groups

The following figure shows the organization of function modules within a function group (main program). The name <fgrp> of the function group can have a length of up to four characters. The system automatically uses this name to create the names of the single components (main program and include programs).

Function Modules - Organization



The main program SAPL<fgrp> (attribut Typ F) contains nothing but the INCLUDE statements for the following include programs:

- The include program L<fgrp>TOP can be used to declare global data for the function group.
- The include program L<fgrp>UXX contains further INCLUDE statements for the include programs L<fgrp>U01, L<fgrp>U02,.... The latter contain the actual coding of the function modules.
- The include programs L<fgrp>F01, L<fgrp>F02,... can contain the coding of subroutines that can be called with internal subroutine calls from all function modules of the group.

The creation of these include programs is supported from the ABAP Development Workbench by forward navigation (for example creation of a subroutine include by

double clicking on the name of a subroutine in a PERFORM statement within a function module).

Special Techniques

Checking the Runtime of Program Segments

The ABAP Development Workbench provides several tools for the runtime measurement of complete application programs namely 'SQL Trace' and 'Runtime Analysis'. For more information about these tools, please read the documentation [BC - ABAP Workbench: Tools \[Ext.\]](#).

This section shows, how you can measure the runtime of specific program segments during the development of ABAP programs.

To do so, you use the GET RUN TIME FIELD statement.

[GET RUN TIME FIELD \[Page 501\]](#)

The following topic shows an example for the runtime of database accesses.

[Measuring the Runtime of Database Accesses \[Page 503\]](#)

GET RUN TIME FIELD

GET RUN TIME FIELD

You can measure the relative runtime of program segments in units of microseconds with the GET RUN TIME FIELD statement. The syntax is as follows:

Syntax

GET RUN TIME FIELD <f>.

The first execution of this statement sets the value of field <f>, which should be of type I, to zero. Each subsequent execution of the statement sets the value of field <f> to the runtime of the program since the first execution of the statement.

Since runtimes are not fixed in the client/server environment of a R/3 system but can vary due to different loads on the system, you should measure the runtime several times and calculate the minimum or the average from the results.

```
DATA T TYPE I.
GET RUN TIME FIELD T.
WRITE: / 'Runtime', T.

DO 10 TIMES.
  GET RUN TIME FIELD T.
  WRITE: / 'Runtime', T.
ENDDO.
```

The output list of this program segment might look as follows:

```
Runtime      0
Runtime      4.926
Runtime      5.228
Runtime      5.649
Runtime      6.100
Runtime      6.512
Runtime      6.906
Runtime      7.324
Runtime      7.724
Runtime      8.231
Runtime      8.623
```

After initialization, the runtime of the DO-Loop is measured in field T.

For measuring the runtime of program segments, it is preferable to calculate the difference between the relative runtimes before and after the respective segment.

```
DATA: T1 TYPE I,
      T2 TYPE I,
      T TYPE P DECIMALS 2,
      N TYPE I VALUE 1000.

T = 0.
DO N TIMES.
  GET RUN TIME FIELD T1.
```

```
*****
* Code to be tested      *
*****
```

```
GET RUN TIME FIELD T2.
T2 = T2 - T1.
T = T + T2 / N.
ENDDO.
```

```
WRITE: / 'Mean Runtime: ', T, 'microseconds'.
```

This example shows a DO-loop that is constructed around a program segment to be tested. In this example, the program segment is just a comment. The difference of the relative runtimes after (T2) and before (T1) the program segment is measured N times. The mean value (T) is calculated from the results. The output might look as follows:

```
Mean Runtime:      23.40 microseconds
```

The output shows the offset time that results from the runtime measurement itself. This offset must be subtracted from the runtime of a real program segment.

If you replace the comment block in the program with the simple assignment

```
T = T.
```

the result might be

```
Mean Runtime:      28.84 microseconds
```

This shows, that the runtime of a simple assignment (without type conversion) is about 5 microseconds. If you assign T to field of type C, due to the type conversion the runtime is enhanced by a factor of about four.

Runtime Measurement of Database Accesses

Runtime Measurement of Database Accesses

The following example shows how you can use the GET RUN TIME FIELD statement to measure the runtime of data base accesses.

```

DATA: T1 TYPE I,
      T2 TYPE I,
      T  TYPE P DECIMALS 2.

PARAMETERS N TYPE I DEFAULT 10.

DATA: TOFF TYPE P DECIMALS 2,
      TSEL1 TYPE P DECIMALS 2,
      TSEL2 TYPE P DECIMALS 2.

TABLES SBOOK.

T = 0.
DO N TIMES.
  GET RUN TIME FIELD T1.
* Offset
  GET RUN TIME FIELD T2.
  T2 = T2 - T1.
  T = T + T2 / N.
ENDDO.
TOFF = T.
WRITE: / 'Mean Offset Runtime   ', TOFF, 'microseconds'.

SKIP.
T = 0.
DO N TIMES.
  GET RUN TIME FIELD T1.

  SELECT * FROM SBOOK.
  ENDSELECT.

  GET RUN TIME FIELD T2.
  T2 = T2 - T1.
  T = T + T2 / N.
ENDDO.
TSEL1 = T - TOFF.
WRITE: / 'Mean Runtime SELECT * ',
      TSEL1, 'microseconds'.

SKIP.
T = 0.
DO N TIMES.
  GET RUN TIME FIELD T1.

  SELECT CARRID CONNID FROM SBOOK
         INTO (SBOOK-CARRID, SBOOK-CONNID).
  ENDSELECT.

  GET RUN TIME FIELD T2.
  T2 = T2 - T1.
  T = T + T2 / N.

```

Runtime Measurement of Database Accesses

```
ENDDO.  
TSEL2 = T - TOFF.  
WRITE: / 'Mean Runtime SELECT List:',  
        TSEL2, 'microseconds'.
```

The output might look as follows:

```
Mean Offset Runtime      :      25.22 microseconds  
Mean Runtime SELECT *    :    257,496.85 microseconds  
Mean Runtime SELECT List:    167,904.63 microseconds
```

This example determines the runtime of three program segments:

- an 'empty' program segment to get the offset of the runtime measurement
- a program segment that reads all data from database table SBOOK
- a program segment that reads two columns of database table SBOOK.

The result shows that the offset of the runtime measurement is negligible in that case and that reading specific columns of a table is faster than reading the whole line.

Generating and Running Programs Dynamically

Generating and Running Programs Dynamically

This section describes how to create, generate, and start ABAP modules during the runtime of your program. This method gives you much more possibilities to influence the flow of a running ABAP unit than explained, for example, in [Specifying Conditions for Line Selection at Runtime \[Page 563\]](#). In the following topics, this will be explained with the help of some simple examples. But note, that this method is very expensive in regard to performance because a dynamically created program has to be generated for each run. Therefore, it is recommended to use the standard way of creating and generating program code with the ABAP Editor if possible.

The following topics explain about

[Creating a New Program Dynamically \[Page 506\]](#)

[Changing Existing Programs Dynamically \[Page 508\]](#)

[Running Programs Created Dynamically \[Page 509\]](#)

[Creating and Starting Temporary Subroutines \[Page 512\]](#)

Creating a New Program Dynamically

To create new dynamic programs during the runtime of an ABAP program, you must use an internal table. For this purpose, you should create this internal table with one character type column and a line width of 72. You can use any method you like from [Filling Internal Tables \[Page 278\]](#) to write the code of your new program into the internal table. Especially, you can use internal fields in which the contents are dependent on the flow of the program that you use to create a new one, to influence the coding of the new program dynamically. The following example shows how to proceed in principal:

```
DATA CODE(72) OCCURS 10.  
APPEND 'REPORT ZDYN1.'  
      TO CODE.  
APPEND 'WRITE / "Hello, I am dynamically created!".'  
      TO CODE.
```

Two lines of a very simple program are written into the internal table CODE.

In the next step you have to put the new module, in the above example it is an executable program (report), into the library. For this purpose you can use the following statement:

Syntax

```
INSERT REPORT <prog> FROM <itab>.
```

The program <prog> is inserted in your present development class in the R/3 Repository. If a program with this name does not already exist, it is newly created with the following attributes:

- Title: none,
- Type: 1 (Reporting),
- Application: S (Basis).

You can specify the name of the program <prog> explicitly within single quotation marks or you can write the name of a character field which contains the program name. The name of the program must not necessarily be the same as given in the coding, but it is recommended to do so. <itab> is the internal table containing the source code. For the above example you could write:

```
INSERT REPORT 'ZDYN1' FROM CODE.  
or  
DATA REP(8).  
REP = 'ZDYN1'  
INSERT REPORT REP FROM CODE.
```

You can refer to and change all components of this new program via the *ABAP Editor*:

You can call the ABAP Editor for the above example:

Creating a New Program Dynamically

```
1 report zdyn1.  
2 write / 'Hello, I am dynamically created!'.  
3  
4
```

If a program with the name <prog> already exists, its source code is replaced without any further warning by the new one from the internal table <itab>.

Therefore, you have to be very careful when naming the dynamically created programs. This is especially true when the program, which is creating the new programs, is going to be used by more than one users in a parallel manner. Then, you have to consider a user specific naming convention, which ensures that one user does not replace the program of another user.

The statement `GENERATE SUBROUTINE POOL` that creates temporary external subroutines in the main memory offers an alternative way of generating dynamic programs (see [Creating and Starting Temporary Subroutines \[Page 512\]](#)).

Changing Existing Programs Dynamically

Since you cannot use the syntax check during the dynamic creation of a program, it might be useful to provide a set of syntactically correct structures in the system, which you can load into an internal table, modify, and write back to the system. Such structures can be comprised of reports, subroutines, include programs, or modules from the module pool. To do so, you use:

Syntax

READ REPORT <prog> INTO <itab>.

This statement reads a program <prog> that is stored in the library into the internal table <itab>.

Now, you can modify the internal table and create a new program as described in [Creating a New Program Dynamically \[Page 506\]](#).

Assume the following simple program:

```
REPORT ZSTRUC1.
```

```
WRITE / 'Hello, I am a little structure!'
```

and the following lines of a program:

```
DATA CODE(72) OCCURS 10.
```

```
READ REPORT 'ZSTRUC1' INTO CODE.
```

```
APPEND 'SKIP.' TO CODE.
```

```
APPEND 'WRITE / "and I am a dynamic extension!".' TO CODE.
```

```
INSERT REPORT 'ZDYN2' FROM CODE.
```

In this example, the existing executable program ZSTRUC1 is read into the internal table CODE, it is extended by two additional lines and written to the program ZDYN2. After calling the ABAP Editor for ZDYN2, the screen appears as follows:

```
1 report zstruc1 .
2 write / 'Hello, I am a little structure!'.
3 skip.
4 write / 'and I am a dynamic extension!'.
5
```

Running Programs Created Dynamically

Running Programs Created Dynamically

After creating programs dynamically, you can start them from the R/3 System as usual. However, it might be desirable not only to create, but also to start programs at runtime. For this purpose, you can

- use the SUBMIT statement,
- work with include programs or subroutines.

To start the created or modified program directly from your running program use:

Syntax

SUBMIT <prog>.

For further information about the SUBMIT statement, see [Calling Executable Programs \(Reports\) \[Page 1113\]](#)

Assume the following simple program:

```
REPORT ZDYN3.
```

```
WRITE / 'Dynamic Program!'.
```

The following executable program (report) starts, modifies, and restarts ZDYN3:

```
REPORT ZMASTER1.
```

```
DATA CODE(72) OCCURS 10.
```

```
DATA LIN TYPE I.
```

```
READ REPORT 'ZDYN3' INTO CODE.
```

```
SUBMIT ZDYN3 AND RETURN.
```

```
DESCRIBE TABLE CODE LINES LIN.
```

```
MODIFY CODE INDEX LIN FROM  
  'WRITE / "Dynamic Program Changed!"'.
```

```
INSERT REPORT 'ZDYN3' FROM CODE.
```

```
SUBMIT ZDYN3.
```

The output of this program appears on two subsequent output screens. The first screen shows:

Dynamic Program!

The second screen shows

Dynamic Program Changed!

When using the SUBMIT statement, all modifications carried out in a program during runtime are immediately valid before they are submitted. In the above example, ZDYN3 is submitted from ZMASTER1 first in its original and then in its modified form, generating different results.

This is not the case when you change the codes of include programs or subroutine programs dynamically.

Running Programs Created Dynamically

Assume again the include program:

```
*** INCLUDE ZINCLUD1.
```

```
WRITE / 'Original INCLUDE program!'.
```

and an executable program (report) for modifying and including it:

```
REPORT ZMASTER2.
```

```
DATA CODE(72) OCCURS 10.
```

```
DATA LIN TYPE I.
```

```
READ REPORT 'ZINCLUD1' INTO CODE.
```

```
DESCRIBE TABLE CODE LINES LIN.
```

```
MODIFY CODE INDEX LIN FROM
```

```
      'WRITE / "Changed INCLUDE program!"'.
```

```
INSERT REPORT 'ZINCLUD1' FROM CODE.
```

```
INCLUDE ZINCLUD1.
```

If you run ZMASTER2, the source code of the include program ZINCLUD1 is changed and replaced in the system. However, in the last line of ZMASTER2 the previous version is carried out. This stems from the fact, that the run time version of ZMASTER2 is generated before ZINCLUD1 is modified. Only at the second start of ZMASTER2 does the system detect that ZINCLUD1 was changed. Exactly the same is true if you modify the source code of a subroutine program dynamically and call it within the same program.

A possibility to overcome this obstacle is to use the INCLUDE statement within an external subroutine, called by the program. With this method, you can create or modify include programs or subroutines and use the updated versions directly in the same program.

Assume again the include program:

```
*** INCLUDE ZINCLUD1.
```

```
WRITE / 'Original INCLUDE program!'.
```

and an external subroutine:

```
PROGRAM ZFORM1.
```

```
FORM SUB1.
```

```
    INCLUDE ZINCLUD1.
```

```
ENDFORM.
```

The following program reads the include program, modifies it, enters it back into the system, and calls the subroutine.

```
REPORT ZMASTER3.
```

```
DATA CODE(72) OCCURS 10.
```

```
READ REPORT 'ZINCLUD1' INTO CODE.
```

```
APPEND 'WRITE / "Extension of INCLUDE program!"' TO CODE.
```

Running Programs Created Dynamically

INSERT REPORT 'ZINCLUD1' FROM CODE.

PERFORM SUB1(ZFORM1).

The produces the following output:

Original INCLUDE program!

Extension of INCLUDE program!

In this case, the updated version of the include program is used in the subroutine because its time stamp is checked during the calling of the subroutine and not during the generation of the calling program.

Creating and Starting Temporary Subroutines

The dynamic programs discussed in the other topics of the section are created in the R/3 Repository. The present topic describes, how you can create and call dynamic subroutines in the main memory.

You write the source code of the subroutines as explained in [Creating a New Program Dynamically \[Page 506\]](#) into an internal table. For generating the Program, you use the following statement:

Syntax

GENERATE SUBROUTINE POOL <itab> NAME <prog> [<options>].

This statement creates a so-called subroutine pool in the main memory area of the running program. You pass the Source code of the subroutine pool in the internal table <itab>. The statement gives you back the name of the generated subroutine pool in the field <prog> that should have type C of length 8. You use the name in <prog> to call the subroutines defined in <itab> via dynamic subroutine calls as explained in [Specifying the Subroutine Name at Runtime \[Page 449\]](#).

The subroutine pool only exists during the runtime of the generating program and can only be called from within this program. You can create up to 36 subroutine pools for one program.

If an error occurs during generation, SY-SUBRC is set to 8. Otherwise it is set to 0.

You can use the following additions <options> in the statement GENERATE SUBROUTINE POOL:

- MESSAGE <mess>
In the case of a syntax error, field <mess> contains the error message.
- INCLUDE <incl>
In the case of a syntax error, field <incl> contains the name of the include program, where the error eventually occurred.
- LINE <line>
In the case of a syntax error, field <line> contains the number of the wrong line.
- WORD <word>
In the case of a syntax error, field <word> contains the wrong word.
- OFFSET <offs>
In the case of a syntax error, field <offs> contains the offset of the wrong word in the line.
- TRACE-FILE <trac>
If you use this addition, you switch on the trace mode and field <trac> contains the trace output.

Compared to INSERT REPORT, the statement GENERATE SUBROUTINE POOL has a better performance. Furthermore, you do not have to care about

Creating and Starting Temporary Subroutines

naming conventions or about restrictions of the correction and transport system when you use statement GENERATE SUBROUTINE POOL.

```
REPORT SAPMZTST.

DATA: CODE(72) OCCURS 10,
      PROG(8), MSG(120), LIN(3), WRD(10), OFF(3).

APPEND 'PROGRAM SUBPOOL.'
      TO CODE.
APPEND 'FORM DYN1.'
      TO CODE.
APPEND
  'WRITE / "Hello, I am the temporary subroutine DYN1!".'
      TO CODE.
APPEND 'ENDFORM.'
      TO CODE.
APPEND 'FORM DYN2.'
      TO CODE.
APPEND
  'WRIT / "Hello, I am the temporary subroutine DYN2!".'
      TO CODE.
APPEND 'ENDFORM.'
      TO CODE.

GENERATE SUBROUTINE POOL CODE NAME PROG
      MESSAGE MSG
      LINE LIN
      WORD WRD
      OFFSET OFF.

IF SY-SUBRC <> 0.
  WRITE: / 'Error during generation in line', LIN,
        / MSG,
        / 'Word:', WRD, 'at offset', OFF.
ELSE.
  WRITE: / 'The name of the subroutine pool is', PROG.
  SKIP 2.
  PERFORM DYN1 IN PROGRAM (PROG).
  SKIP 2.
  PERFORM DYN2 IN PROGRAM (PROG).
ENDIF.
```

In this program, a subroutine pool containing two subroutines is filled into the table CODE. Note, that the temporary program must contain a REPORT or PROGRAM statement. The statement GENERATE SUBROUTINE POOL generates the temporary program. The output is as follows:

```
Error during generation in line 6
The statement "WRIT" is not expected. A correct similar statement is "WRITE".
Word: WRIT      at offset 0
```

An error occurred during generation because the WRITE statement is wrong in the second subroutine, DYN2. After the following correction of that line:

Creating and Starting Temporary Subroutines

```
APPEND  
'WRITE / "Hello, I am the temporary subroutine DYN2!".'  
  TO CODE.
```

the output looks as follows:

```
The name of the subroutine pool is %_T01E00  
  
Hello, I am the temporary subroutine DYN1!  
  
Hello, I am the temporary subroutine DYN2!
```

Generation was successful. The internal program name is displayed. Then, the subroutines of the subroutine pool are called. Note that the explicit knowledge of the program name is not necessary for calling the subroutines.

ABAP Objects

General

[What are ABAP Objects? \[Page 516\]](#)

[What is Object Orientation? \[Page 517\]](#)

Object Orientation in ABAP

[From Function Groups to Objects \[Page 519\]](#)

[Classes and Class Components \[Page 522\]](#)

[Reference Variables and Object Instances \[Page 526\]](#)

Special Techniques

[Interfaces \[Page 532\]](#)

[Events \[Page 534\]](#)

[Further Reading \[Page 536\]](#)

What are ABAP Objects?

ABAP Objects is a new concept in R/3 Release 4.0. The term has two meanings. On the one hand, it stands for the entire ABAP runtime environment. On the other hand, it represents the object-oriented extension of the ABAP language.

The Runtime Environment

The new name ABAP Objects for the entire ABAP runtime environment is an indication of the way in which SAP has, for some time, been moving towards object orientation, and of its commitment to pursuing this line further. The ABAP Workbench allows you to create R/3 Repository objects such as programs, authorization objects, lock objects, Customizing objects, and so on. Using function modules, you can encapsulate functions in separate programs with a defined interface. The Business Object Repository (BOR) allows you to create SAP Business Objects for internal and external use (DCOM/CORBA). Until now, object-oriented techniques have been used exclusively in system design, and have not been supported by the ABAP language.

The Object-oriented Language Extension

Object orientation has many advantages, such as encapsulation and reusability, which make software simpler and easier to change. To introduce these advantages into ABAP, SAP has developed a set of object-oriented language elements which are part of the language from Release 4.0. This is an extension of the existing language, and fully compatible with it. In particular, you can include ABAP Objects in existing programs, or convert program sections that are already well encapsulated (such as function modules) into objects without having to change too much of their source code. The object-oriented enhancement of ABAP is based on the models of Java and C++. It is compatible with external object interfaces such as DCOM and CORBA. The implementation of object-oriented elements in the kernel of the ABAP language has considerably increased response times when you work with ABAP Objects. SAP Business Objects and GUI objects - already object-oriented themselves - will also profit from being incorporated in ABAP Objects.

What is Object Orientation?

What is Object Orientation?

Object orientation, or to be more precise, object-oriented programming, is a problem-solving method in which the software solution reflects objects in the real world.

A comprehensive introduction to object orientation as a whole would go far beyond the limits of this introduction to ABAP Objects. This documentation introduces a selection of terms that are used universally in object orientation and also occur in ABAP Objects. In subsequent sections, it goes on to discuss in more detail how these terms are used in ABAP Objects.

Objects

An object is a section of source code that contains data and provides services. The data forms the attributes of the object. The services are known as methods (also known as operations or functions). Typically, methods operate on private data (the attributes, or state of the object), which is only visible to the methods of the object. Thus the attributes of an object cannot be changed directly by the user, but only by the methods of the object.

Classes

Classes describe objects. From a technical point of view, objects are runtime instances of a class. In theory, you can create any number of objects based on a single class. Each instance (object) of a class has a unique identity and its own set of values for its attributes.

Object References

In a program, you address objects using unique object references. Object references allow you to create objects and to access the objects, their attributes, and their methods.

In object-oriented programming, objects usually have the following properties:

Encapsulation

Objects restrict the visibility of their resources (attributes and methods) to other users. Every object has an **interface**, which determines how other objects can interact with it. The implementation of the object is encapsulated, that is, invisible outside the object itself.

Polymorphism

Identical (identically-named) methods behave differently in different classes. Object-oriented programming contains constructions called interfaces. They enable you to address methods with the same name in different objects. Although the form of address is always the same, the implementation of the method is object-specific.

Inheritance

You can use an existing class to derive a new class. Derived classes inherit the data and methods of the superclass. However, they can overwrite existing methods, and also add new ones.

You do not necessarily need an object-oriented language to be able to use object-oriented programming. Object-oriented programming languages and tools are an effective support for object-oriented programming techniques. Object-oriented programming requires a large amount of planning, analysis and design, which far exceeds the actual time required to implement the

software. For a guide to other introductions to object-oriented programming, refer to [Further Reading \[Page 536\]](#)

This section of the ABAP User's Guide provides an overview of the object-oriented extension of the ABAP language. We have used simple examples to demonstrate how to use the new features. However, these are not intended to be a model for object-oriented design. More detailed information about each of the ABAP Objects statements is contained in the keyword documentation in the ABAP Editor.

From Function Groups to Objects

From Function Groups to Objects

At the center of any object-oriented model are objects, which contain attributes (data) and methods (functions). Objects should enable programmers to map a real problem and its proposed software solution on a one-to-one basis. Typical objects in a business environment are, for example, 'customer', 'Order', or 'Invoice'. From Release 3.1 onwards, the Business Object Repository (BOR) has contained examples of such objects. The object model of ABAP Objects, the object-oriented extension of ABAP, is compatible with the object model of the BOR.

Before R/3 Release 4.0, the nearest equivalent of objects in ABAP were function modules and function groups. Suppose we have a function group for processing orders. The attributes of an order correspond to the global data of the function group, while the individual function modules represent actions that manipulate that data (methods). This means that the actual order data is encapsulated in the function group, and is never directly addressed, but instead only through the function modules. In this way, the function modules can ensure that the data is consistent.

The following example shows the object-oriented aspect of function groups in the simple case of a counter.

Suppose we have a function group called COUNTER:

```
FUNCTION-POOL COUNTER.
```

```
DATA COUNT TYPE I.
```

```
FUNCTION SET_COUNTER.
```

```
* Local Interface IMPORTING VALUE(SET_VALUE)
```

```
  COUNT = SET_VALUE.
```

```
ENDFUNCTION.
```

```
FUNCTION INCREMENT_COUNTER.
```

```
  ADD 1 TO COUNT.
```

```
ENDFUNCTION.
```

```
FUNCTION GET_COUNTER.
```

```
* Local Interface: EXPORTING VALUE(GET_VALUE)
```

```
  GET_VALUE = COUNT.
```

```
ENDFUNCTION.
```

The function group has a global integer field COUNT, and three function modules, SET_COUNTER, INCREMENT_COUNTER, and GET_COUNTER, that work with the field. Two of the function modules have input and output parameters. These form the data interface of the function group.

Any ABAP program can then work with this function group. For example:

```
DATA NUMBER TYPE I VALUE 5.
```

```
CALL FUNCTION 'SET_COUNTER' EXPORTING SET_VALUE = NUMBER.
```

```
DO 3 TIMES.
```

```
  CALL FUNCTION 'INCREMENT_COUNTER'.
```

```
ENDDO.
```

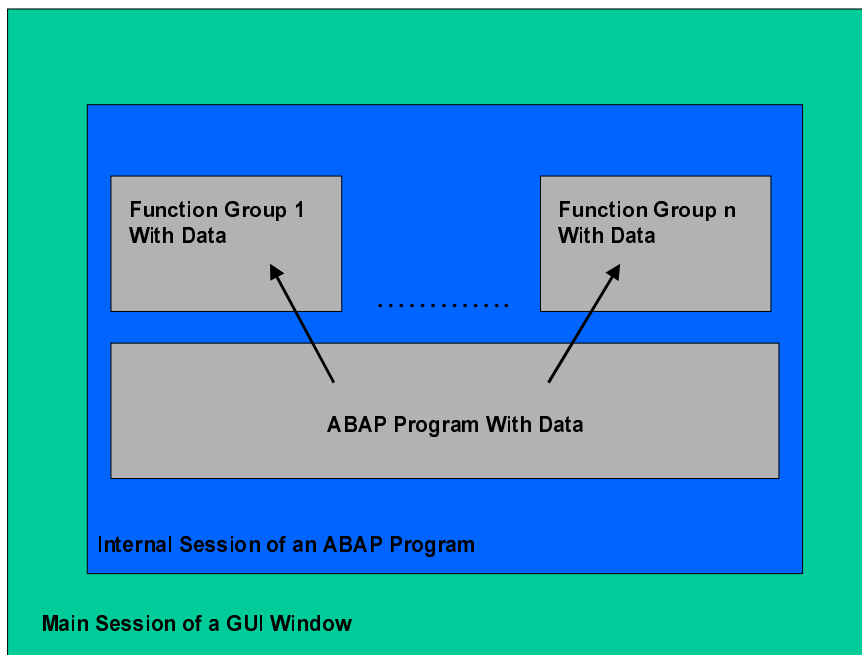
```
CALL FUNCTION 'GET_COUNTER' IMPORTING GET_VALUE = NUMBER.
```

After this section of the program has been processed, the program variable NUMBER will have the value 8.

From Function Groups to Objects

The program itself cannot access the COUNT field in the function group. Operations on this field are fully encapsulated in the function module. The program can only communicate with the function group by calling its function modules.

When you run an ABAP program, the system starts a new internal session. The internal session has a memory area that contains the ABAP program and its associated data. When you call a function module, its main program, that is, the entire function group plus its data, is loaded into the memory area of the internal session.



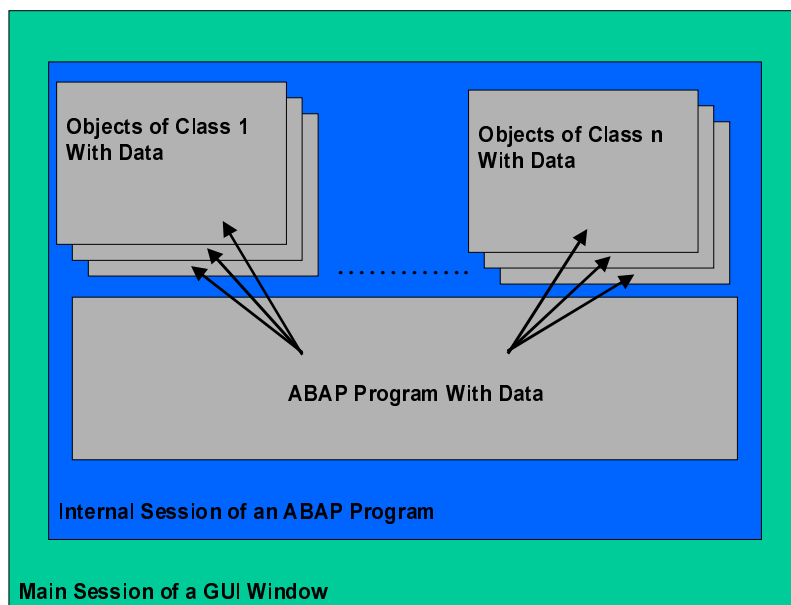
The function group **in the memory area of the internal session** is almost the same as an object in the object-oriented sense (compare the definitions in [What is Object Orientation? \[Page 517\]](#)). When you call a function module, the calling program uses the runtime instance of a function group, based on its description in the Function Builder.

The main difference between object-orientation and function groups is that although a program can work with the instances of several function groups at the same time, it cannot work with several instances of a single function group. Suppose a program wanted to use several independent counters, or process several orders at the same time. In this case, you would have to adapt the function group to include instance administration, for example, by using numbers to differentiate between the instances.

In practice, this is very awkward. Consequently, the data is usually stored in the calling program, and the function modules are called to work with it. The problem with this is that all of the programs that use the function modules must use the same data structures as the function group itself. If you change the internal data structure of a function group, you affect a large number of users, and it is often difficult to predict the implications of the changes. The only way to avoid this is to rely heavily on instances and a technique that guarantees that the internal structures of runtime instances will remain hidden, allowing you to change them later without causing any problems.

From Function Groups to Objects

This requirement is met by object-orientation. ABAP Objects allows you to define data and functions in classes instead of function groups. Using classes, an ABAP program can work with any number of runtime instances based on the same template.



Instead of loading a single runtime instance of a function group into memory implicitly when a function module is called, the ABAP program can now generate the runtime instances of classes explicitly using the new ABAP statement `CREATE OBJECT`. The individual runtime instances represent unique objects. You address these using object references. Object references are a new ABAP data type.

The following sections contain more information about classes, objects, and object references in ABAP Objects.

Classes and Class Components

Classes are templates for objects. Theoretically, you can generate any number of objects for a given class. From the programming point of view, classes are reusable blocks of source code, which you use to generate runtime objects in memory. Thus the relationship between a class and its objects is the same as that between the source code of a main program (and its subroutines) and their load versions in memory. However, there is not a precise parallel between classes and main programs and subroutines.

Local and Global Classes

There is a difference in ABAP Objects between local and global classes. The source code of a local class is part of an ABAP program, generated using the ABAP Editor. Local classes are only visible in the program in which they are defined, and you can only create objects for them within that program. The source code of a global class is written using the Class Builder and stored in the systemwide Class Library. Any ABAP program can generate objects for any global class. When you use a class in an ABAP program, the system first searches for a local class with the specified name. If it does not find one, it then looks for a global class.

At runtime, it makes no difference whether an object has been generated from a local or a global class. ABAP programs use the objects in their memory area in the same way, regardless of whether the class is defined in its own source code or in the R/3 Repository. The only exception to this is that the objects of local classes can access the global data of the user, whereas objects of local classes cannot.

There is, however, a significant difference in the way that local and global classes are administered. If you are defining a local class that is only used in a single program, it is usually sufficient to define the outwardly visible data so that it fits into that program. Global classes, on the other hand, must be able to be used anywhere. This means that certain restrictions apply when you define the interface of a global class, since the system must be able to guarantee that any program using an object of a global class can recognize the data type of each interface parameter.

Defining Classes

Classes consist of ABAP source code, enclosed in the ABAP statements `CLASS... ENDCLASS`. For a local class, you enter this source code directly in the corresponding ABAP program. For a global class, it is generated by the Class Builder in special tables within the R/3 Repository.

A complete class definition consists of a declaration part and, if required, an implementation part. The declaration part of a class `<class>` is a statement block:

```
CLASS <class> DEFINITION.  
...  
ENDCLASS.
```

It contains the declaration for all **class components** (attributes, methods, events). When you define local classes, the declaration part belongs to the global program data. You should therefore place it at the beginning of the program.

If you declare methods in the declaration part of a class, you must also write an implementation part for it. This consists of a further statement block:

```
CLASS <class> IMPLEMENTATION.  
...  
ENDCLASS.
```

Classes and Class Components

The implementation part of a class contains the implementation of all **methods** of the class. The implementation part of a local class is a processing block. Subsequent coding that is not itself part of a processing block is therefore not accessible.

Visibility Sections

You can divide the declaration part of a class into up to three visibility areas:

```
CLASS <class> DEFINITION.  
  PUBLIC SECTION.  
  ...  
  PROTECTED SECTION.  
  ...  
  PRIVATE SECTION.  
  ...  
ENDCLASS.
```

These sections define the visibility of the **class components**.

- **PUBLIC SECTION**
All of the components declared in the public section are accessible to all users of the class, and to the methods of the class and any classes that inherit from it.
- **PROTECTED SECTION**
All of the components declared in the protected section are accessible to all methods of the class and of classes that inherit from it.
- **PRIVATE SECTION**
Any components declared in the private section of the class are visible only to the methods of the class itself.

The use of visibility sections gives ABAP Objects the important characteristic of encapsulation. When you define a class, you should take great care in designing the public components, and try to declare as few public components as possible. The public components of the class form the interface between objects and their users. As such, they may not be changed once you have released the class.

Class Components

The components of a class make up its contents. Class components are the individual parts of the objects that are generated from the class. They are the building blocks that determine the function of the object. All class components must belong to the same namespace, and must all be assigned to one of the three **visibility sections**.

Types

In a class (except in the public section of a global class) you can use the `TYPES` statement to create internal data types. Another restriction that applies to the public section of global classes is that you may not implicitly declare any types in a `DATA` statement or in the method interface definition (for example, by specifying length and offset, or the number of decimal places). Within the public section of a global class, you may only refer to systemwide data types from the ABAP Dictionary.

Attributes

Attributes contain the actual data of an object. You can declare them in any visibility section of the definition part of a class using the DATA statement. From a programming point of view, attributes within a class are the equivalent of global data within a program. All of the attributes of a class are visible within the class, as long as they are not obscured by local attributes in methods or subroutines. The visibility sections define the outward visibility of attributes.

Only public attributes are universally visible from outside the class. They are part of the interface between objects and their users. You should, however, restrict the use of public attributes to exceptions. In object-oriented programming, you should keep the data content of an object encapsulated as far as possible, and only accessible using methods. Consequently, you will mostly use private attributes, which are not visible from outside the class.

Methods

Methods provide the functions associated with an object. They can access all of the attributes of a class. This allows them to change the data content of an object. They also have interface parameters, with which users can supply them with values when calling them, and receive values back from them. The methods of a class can be addressed universally within the class. The visibility section in which they are defined determines whether they can be addressed externally.

In object-oriented programming, public methods and their parameters are the main interface between objects and their users. If you do not use any public attributes, the interface parameters of the public methods form the data interface of an object.

You can declare methods in any visibility section of the definition part of a class using the METHODS statement. You also define the interface parameters in the METHODS statement. You program the actual function of a method in the implementation part of the class, using a processing block that starts with the METHOD statement and ends with the ENDMETHOD statement. You call methods using the CALL METHOD statement.

You can declare local data types and objects in methods in the same way as in other ABAP routines (subroutines and function modules). If local data has the same name as a class attribute, the class attribute is obscured in the method. However, you can still address it using a ME -> reference.

Events

Objects can trigger events and call event handler methods. Like methods, events can have interface parameters, and like all class components, they are subject to the normal visibility rules.

You define events in the definition part of a class using the EVENTS statement. You also use the EVENTS statement to define the interface parameters. The methods of the same class or a subordinate class can trigger an event using the RAISE EVENTS statement. You can register methods of the same or a different class as event handler methods. The interface of an event handler method is determined by the EVENTS statement.

At runtime, you can register the event handler methods of one object as handlers for the events of other objects. When events are triggered, the event handler methods are called in the registered objects. See also [Triggering and Handling Events \[Page 534\]](#)

Static Attributes, Static Methods, and Static Events

The class components introduced so far are merely templates for the components of objects. This means that you cannot use them until you have generated an object as an instance of a given class. For this reason, they are also known as instance attributes, instance methods, and instance events.

However, ABAP Objects also offers static attributes, methods, and events. These class components are independent of instances, and exist once and once only in each class. They are

Classes and Class Components

created in the memory area of the calling program when they are first addressed, and remain there until the program ends. They are global components, and can be used by all instances of the class. Depending on the visibility section in which they are declared, they may also be visible from outside the class.

You declare static components in the definition part of a class using the CLASS-DATA, CLASS-METHODS, and CLASS-EVENTS statements. They are restricted in that they can only work with static attributes, and can only trigger static events.

Constants

Constants are special static attributes. You set their values when you declare them, and they can then no longer be changed. Constants are independent of instances. They exist once only in the memory area of a program. You define constants in the definition part of a class using the CONSTANTS statement.

The following example uses ABAP Objects to program a counter. For comparison, see also the example in [From Function Groups to Objects \[Page 519\]](#)

```
CLASS C_COUNTER DEFINITION.
```

```
  PUBLIC SECTION.
```

```
    METHODS: SET_COUNTER IMPORTING VALUE(SET_VALUE) TYPE I,  
              INCREMENT_COUNTER,  
              GET_COUNTER EXPORTING VALUE(GET_VALUE) TYPE I.
```

```
  PRIVATE SECTION.
```

```
    DATA COUNT TYPE I.
```

```
ENDCLASS.
```

```
CLASS C_COUNTER IMPLEMENTATION.
```

```
  METHOD SET_COUNTER.  
    COUNT = SET_VALUE.  
  ENDMETHOD.
```

```
  METHOD INCREMENT_COUNTER.  
    ADD 1 TO COUNT.  
  ENDMETHOD.
```

```
  METHOD GET_COUNTER.  
    GET_VALUE = COUNT.  
  ENDMETHOD.
```

```
ENDCLASS.
```

The class C_COUNTER contains three public methods - SET_COUNTER, INCREMENT_COUNTER, and GET_COUNTER. Each of these works with the private integer field COUNT. Two of the methods have input and output parameters. These form the data interface of the class. The field COUNT is not outwardly visible.

In the example in the next section, [Reference Variables and Object Instances \[Page 526\]](#), you will see how to create instances of the class C_COUNTER.

Reference Variables and Object Instances

Objects are runtime instances of classes in the memory area of a program. A program accesses individual objects using pointers called reference variables. To create an instance of a class, you need a reference variable with the same type as the class. To create a reference variable, use the new data type object reference.

Reference Variables

You declare reference variables using the statement

```
DATA <obj> TYPE REF TO <class>.
```

The system first looks for the definition of the class <class> locally in the same program. If it does not find it, it looks globally in the Class Library. Reference variables are also known as object variables with the type <class>. An object variable <obj> with type <class> can **only** point to instances of the class <class>.

You can use the addition TYPE REF TO <class> in any ABAP statement in which you can specify types - for example, in the TYPES statement, or when you specify the type of an interface parameter in a routine.

Object Instances

Once you have declared an object variable <obj> for a class <class>, you can create an object using the statement

```
CREATE OBJECT <obj>.
```

The system generates a runtime instance of the class <class> in memory, and the object variable <obj> points to the object. The program can use the object variable to access the **public** components of the object. To access the object components using the object variable <obj>, use the following syntax:

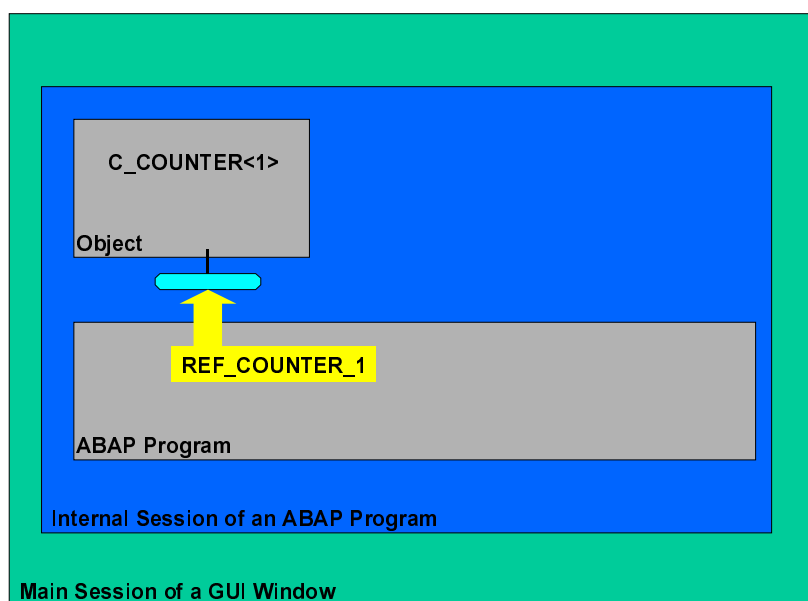
- To access an object attribute <attr>: <obj>-><attr>
- To call a method <meth>: CALL <obj>-><meth>

The following example shows how to create an instance of the class C_COUNTER that we created in the previous section (see [Classes and Class Components \[Page 522\]](#)):

```
DATA REF_COUNTER_1 TYPE REF TO C_COUNTER.  
DATA NUMBER TYPE I VALUE 5.  
CREATE OBJECT REF_COUNTER_1.  
CALL METHOD REF_COUNTER_1->SET_COUNTER  
    EXPORTING SET_VALUE = NUMBER.  
DO 3 TIMES.  
    CALL METHOD REF_COUNTER_1->INCREMENT_COUNTER.  
ENDDO.  
CALL METHOD REF_COUNTER_1->GET_COUNTER  
    IMPORTING GET_VALUE = NUMBER.
```

Reference Variables and Object Instances

The first statement creates an object variable with type C_COUNTER. The CREATE OBJECT statement creates a first object of the class C_COUNTER in the program memory. The object variable REF_COUNTER_1 points to the object and its interface.



This first instance of the class C_COUNTER is called C_COUNTER<1>, because this is how the contents of the object variable REF_COUNTER_1 are displayed in the debugger after the CREATE OBJECT statement has been executed. This name is only used for internal program administration - it does not occur in the ABAP program itself.

The ABAP program only uses the object variable REF_COUNTER_1 to access the public methods of the object. After this section of the program has been executed, both the program variable NUMBER and the private object variable COUNT have the value 8.

You can create any number of objects with the same type. The objects are fully independent of each other. Each one has its own identity within the program and its own attributes. Each CREATE OBJECT statement generates a new object, whose identity is defined by its unique object reference.

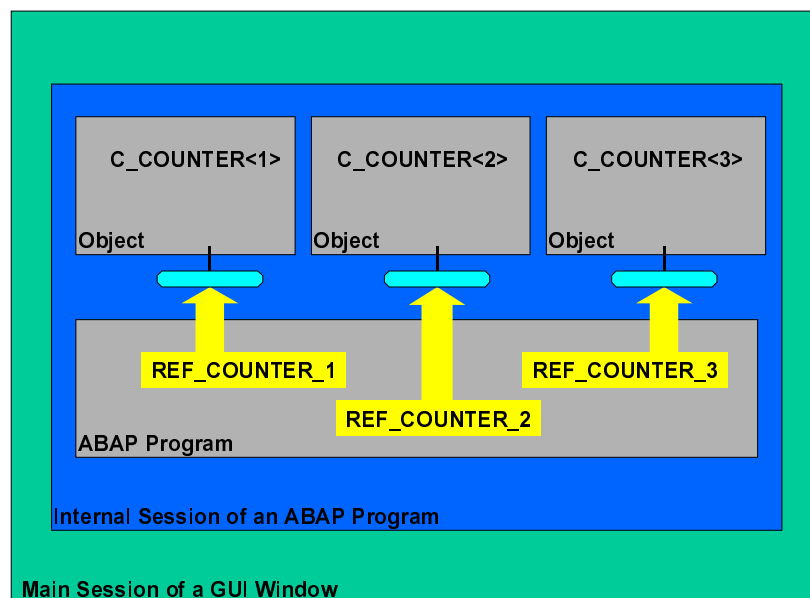
The following example demonstrates how a program generates several instances of the class C_COUNTER (see the example in [Classes and Class Components \[Page 522\]](#)) to function as independent counters.

```
DATA: REF_COUNTER_1 TYPE REF TO C_COUNTER,
      REF_COUNTER_2 TYPE REF TO C_COUNTER,
      REF_COUNTER_3 LIKE REF_COUNTER_1.
```

```
DATA: NUMBER1 TYPE I VALUE 5,  
      NUMBER2 TYPE I VALUE 0,  
      NUMBER3 TYPE I VALUE 2.  
  
CREATE OBJECT: REF_COUNTER_1,  
              REF_COUNTER_2,  
              REF_COUNTER_3.  
  
CALL METHOD: REF_COUNTER_1->SET_COUNTER  
  EXPORTING SET_VALUE = NUMBER1,  
           REF_COUNTER_2->SET_COUNTER  
  EXPORTING SET_VALUE = NUMBER2,  
           REF_COUNTER_3->SET_COUNTER  
  EXPORTING SET_VALUE = NUMBER3.  
  
DO 3 TIMES.  
  CALL METHOD REF_COUNTER_1->INCREMENT_COUNTER.  
ENDDO.  
  
CALL METHOD: REF_COUNTER_2->INCREMENT_COUNTER,  
           REF_COUNTER_3->INCREMENT_COUNTER.  
  
CALL METHOD: REF_COUNTER_1->GET_COUNTER  
  IMPORTING GET_VALUE = NUMBER1,  
           REF_COUNTER_2->GET_COUNTER  
  IMPORTING GET_VALUE = NUMBER2,  
           REF_COUNTER_3->GET_COUNTER  
  IMPORTING GET_VALUE = NUMBER3.
```

The first three statements create object variables with type C_COUNTER. The CREATE OBJECT statement creates three objects of the class C_COUNTER in the program memory. The corresponding object variables point to these objects.

Reference Variables and Object Instances



In the internal program management, the individual instances are called C_COUNTER<1>, C_COUNTER<2>, and C_COUNTER<3>. They are named in the order in which they were created. The ABAP program only uses the object variables to access the instances. Once this part of the program has been processed, the program variables NUMBER1, NUMBER2 and NUMBER3 have the values 8, 1, and 3. Likewise, the internal variables COUNT within the objects have the same different values. Thus, all of the object have the same type, but different attributes - in this case, the count.

More than one reference variable may point to a single object. Apart from in the CREATE OBJECT statement, you can assign object variables to objects using normal assignment statements (ABAP keyword MOVE) between object variables. The only condition for this is that the individual object variables must have the same types. This allows you to reassign references between objects in the same class, so the original object variables do not have to point to the same object all the time.

An object remains in existence for as long as at least one reference variable still points to it. As soon as no more reference variables point to the object, it is deleted by the garbage collection mechanism (automatic memory management for objects).

The following example demonstrates how you can reassign the references to instances of the class C_COUNTER (see the example in [Classes and Class Components \[Page 522\]](#)).

```
DATA: REF_COUNTER_1 TYPE REF TO C_COUNTER,
      REF_COUNTER_2 TYPE REF TO C_COUNTER,
      REF_COUNTER_3 LIKE REF_COUNTER_1.
```

```
CREATE OBJECT: REF_COUNTER_1,
              REF_COUNTER_2.
```

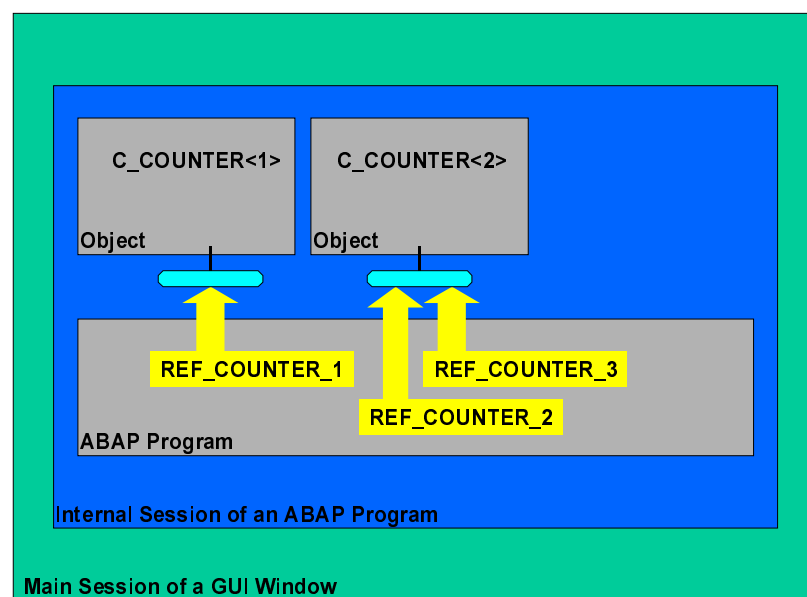
Reference Variables and Object Instances

```
MOVE REF_COUNTER_2 TO REF_COUNTER_3.
```

```
CLEAR REF_COUNTER_2.
```

```
REF_COUNTER_1 = REF_COUNTER_3.
```

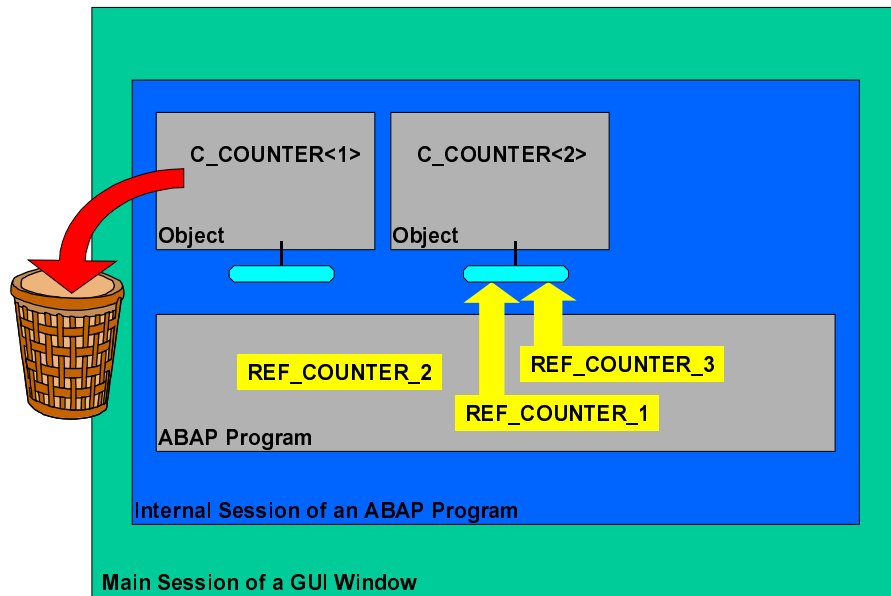
The first three statements create object variables with type C_COUNTER. The CREATE OBJECT statement creates two objects of the class C_COUNTER in the program memory. The object variables REF_COUNTER_1 and REF_COUNTER_2 point to objects with the respective internal names C_COUNTER<1> and C_COUNTER<2>. After the MOVE statement, REF_COUNTER_3 also points to the object C_COUNTER<2>.



Accessing C_COUNTER<2> using either of these object variables will have exactly the same result. The attribute values belong to the object itself, and not to the reference variables that address it.

After the CLEAR statement, the variable REF_COUNTER_2 no longer points to an object, and therefore has the same state as exactly following its declaration. The last assignment statement reassigns the object variable REF_COUNTER_1 from C_COUNTER<1> to C_COUNTER<2>.

Reference Variables and Object Instances



This means that there are no more reference variables pointing to object C_COUNTER<1>. It is therefore deleted automatically from memory.

Interfaces

Classes, their instances (objects), and access to objects using reference variables form the basics of ABAP Objects. These means already allow you to model typical business applications, such as customers, orders, order items, invoices, and so on, using objects, and to implement solutions using ABAP Objects.

However, it is often necessary for similar classes to provide similar functions that are coded differently in each class but which should provide a uniform point of contact for the user. For example, you might have two similar classes, savings account and check account, both of which have a method for calculating end of year charges. The interfaces and names of the methods are the same, but the actual implementation is different. The user of the classes must also be able to run the end of year method for all accounts, without having to worry about the actual type of each individual account.

ABAP Objects makes this possible by using interfaces. Interfaces provide a generic set of functions that you add onto a class. From the point of view of their definition and use in programs, interfaces are very much like classes. An interface definition looks like a class definition. Interfaces contain the same components as classes, and you address them in programs using reference variables. However, from the inside, interfaces are handled completely differently to classes. They are never used to generate objects, and consequently do not contain an implementation part. Instead, they are implemented by classes. The implementation part of a class that uses an interface must also implement the methods of that interface. Thus the methods have exactly the same parameter interface, but can be implemented completely differently. Interface references can point to all objects of a class that has implemented the interface. Using interface references, a program can access the components of an object that are defined in that interface. This enables programs to address components of different objects that have the same name.

Interfaces extend the components of classes by adding their own components.
Interface references provide a new way of accessing the components.

Defining Interfaces

You can define interfaces in a similar way to classes; either locally in your program, or globally using the Class Builder. The definition of an interface <ifac> consists only of a declaration part:

```
INTERFACE <ifac>.  
...  
ENDINTERFACE.
```

Unlike classes, interface definitions have no implementation part. The declaration part contains the declaration for all **interface components** (attributes, methods, events). When you define local interfaces, the declaration part belongs to the global program data. You should therefore place it at the beginning of the program.

You **cannot** divide the declaration part of an interface into the visibility sections that you use in class definitions. All of the components of an interface are automatically public. This characteristic reinforces the nature of interfaces as extensions of classes.

Implementing Interfaces

Using Interfaces

Interfaces

Classes can implement interfaces. Such classes can then be addressed via the interfaces that they have implemented. For example, a class 'Customer' might implement the interface 'Iarchive' and thus become archiveable, or 'Iworkflow' to be able to take part in workflow.

As well as object references, ABAP Objects also contains **interface references**. Classes that implement a particular interface can be addressed using an interface reference. This allows you to access the components defined within the interface. A user can then, for example, address different objects, such as customers, orders, or order items, from the point of view of the interface, that is, in a uniform way.

Interfaces define a specific set of (usually generic) functions. Unlike classes, interfaces **do not just have many users, they are also implemented by many classes**.

This section is still in preparation. Refer also to the keyword documentation for the INTERFACES statement.

Triggering and Handling Events

This section is still in preparation. Instead, refer to the keyword documentation for the RAISE EVENT and SET HANDLER statements.

Inheritance

Inheritance

Inheritance is not yet supported in ABAP Objects.

Further Reading

There are many good books about object orientation, the different object-oriented language, object-oriented analysis and design (OOAD), project management from the object-oriented view, patterns and frameworks, and so on. This list is just a small selection of recommended literature:

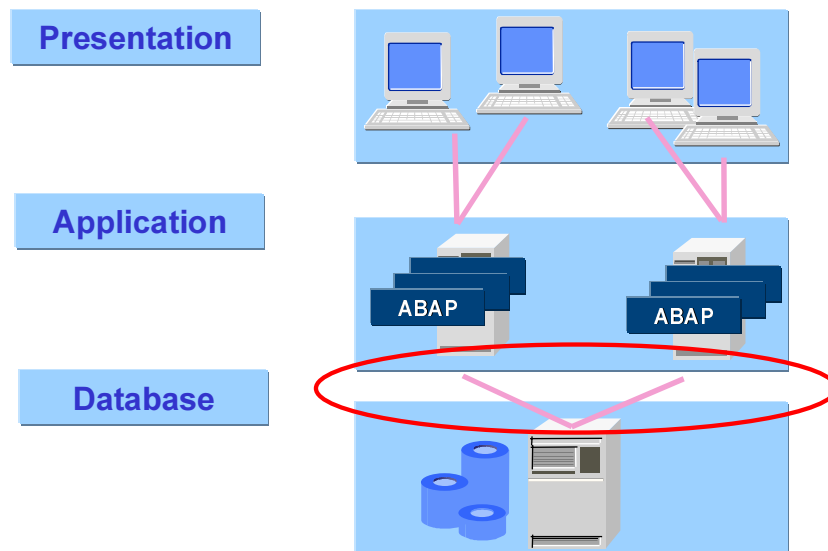
- **Scott Ambler: The Object Primer.** A very good introduction to object orientation for programmers. It introduces the essential concepts of object orientation, and contains a procedure model to enable you to learn object orientation quickly and thoroughly. Contents: The OO paradigm, OO concepts, CRC modelling, Use cases, Project management, and so on.
This is an easy-to-read, practical book, but nevertheless based on theory.
Unfortunately, it uses its own notation instead of UML, but a UML version is planned.
- **Grady Booch: Object Solutions: Managing the Object-Oriented Project.** A book from one of the 'OO gurus' about the non-technical side of object orientation, but covering aspects that are still important to the successful use of object-oriented techniques. Easy to read and full of practical tips.
- **Martin Fowler: UML Distilled.** A good book about UML (Unified Modeling Language, the new standardized object-oriented modeling language and notation) and how to use it. An excellent book from one of the OOA/D gurus. (Assumes OO knowledge/experience).
- **James Rumbaugh. OMT Insights.** A collection of articles addressing many questions and problems in the area of OO analysis and design, implementation, managing dependencies, and so on. Highly recommended.
- **Gamma et al.: Design Patterns. Elements of Reusable Object-Oriented Software.** Provides patterns showing how recurring design problems can be solved using objects. This is the first big 'pattern' book, and is a classic, containing a wealth of examples of good OO design.
- **Booch, Jacobson, Rumbaugh:** Three further books are planned by the authors of UML: a User's Guide, Language Reference, and Objectory Process.

Beginners in OO should definitely read "The Object Primer", and then go on to acquire some practical experience. Start by using the CRC techniques of OO analysis and design; these are described well by Ambler and Fowler. After this, you should move on to UML, since this is the universal notation for OO analysis and design. Finally, a book about patterns is a must.

When you begin a large OO project, the first question is always "what shall we do in what sequence?". Other questions include when a phase is "finished", how to divide and organize the development work, how to minimize the risks, how to assemble a good team, and so on. Many of the issues of good project management had to be readdressed for object-oriented projects, and this has resulted in many new opportunities. To see how OO can be useful in project management, see 'Object Solutions' by Grady Booch, and the chapter 'An outline development process' by Fowler.

There are, of course, many other good titles available about object orientation, and the above list claims neither to be complete nor necessarily to recommend the 'best' books.

ABAP Database Access



Reading and Processing Database Tables

This section covers the following topics:

SQL Concept in ABAP

[Database Tables and SQL Concepts \[Page 539\]](#)

ABAP Open SQL

[Reading Data from Database Tables \[Page 542\]](#)

[Changing the Contents of Database Tables \[Page 570\]](#)

[Reading Lines of Database Tables Using a Cursor \[Page 588\]](#)

[Writing or Undoing Changes to Database Tables \[Page 594\]](#)

[Specifying Clients for Processing Database Tables \[Page 596\]](#)

[Performance Notes \[Page 597\]](#)

ABAP Native SQL

[Using Native SQL Statements in an ABAP Program \[Page 599\]](#)

Locks and Authorizations

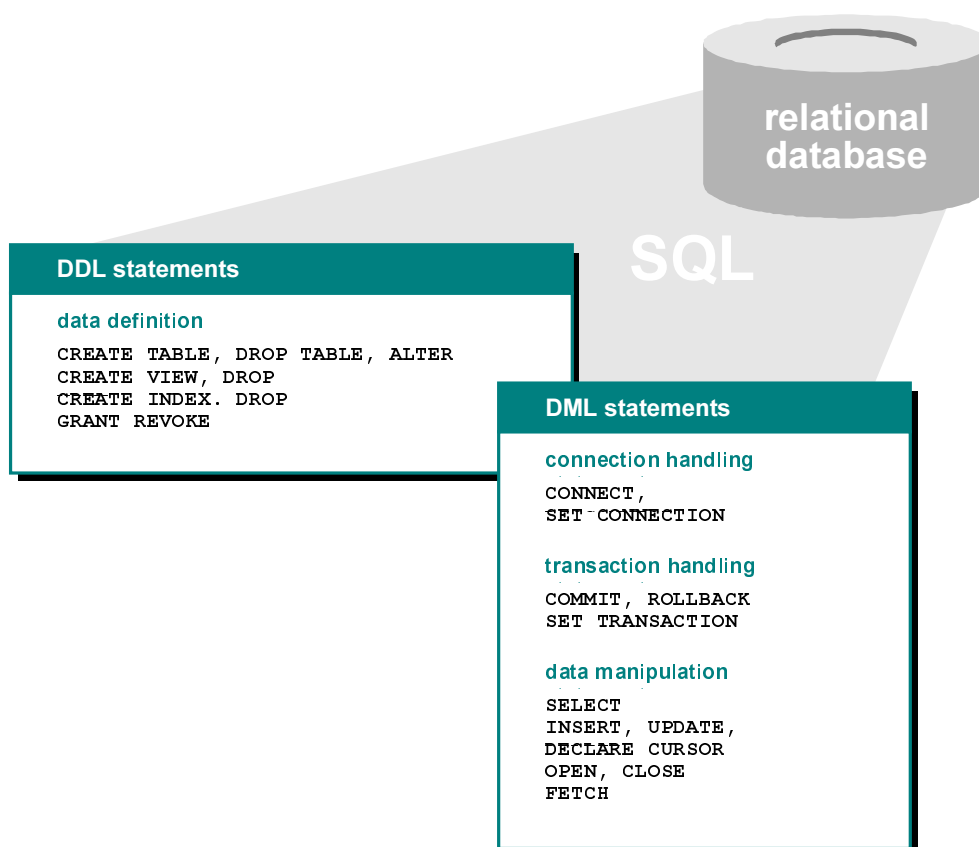
[Locking Database Objects During Program Execution \[Page 628\]](#)

[Checking the Authorization of Program Users \[Page 629\]](#)

Database Tables and SQL Concepts

In the R/3 System, long-life data is stored in relational database tables. For information about the various types of database tables and how they are created and maintained, see the [ABAP Dictionary \[Ext.\]](#) documentation.

Structured Query Language (SQL) was created for accessing relational databases. SQL has two statement types: Data Definition Language (DDL) statements and Data Manipulation Language (DML) statements.



Data Controlling Language (DCL) statements make up a third category of statements.

At present, SQL is still not fully standardized. To access a specific database system, you must refer to the documentation of that system for a list of the SQL statements available and their correct syntax.

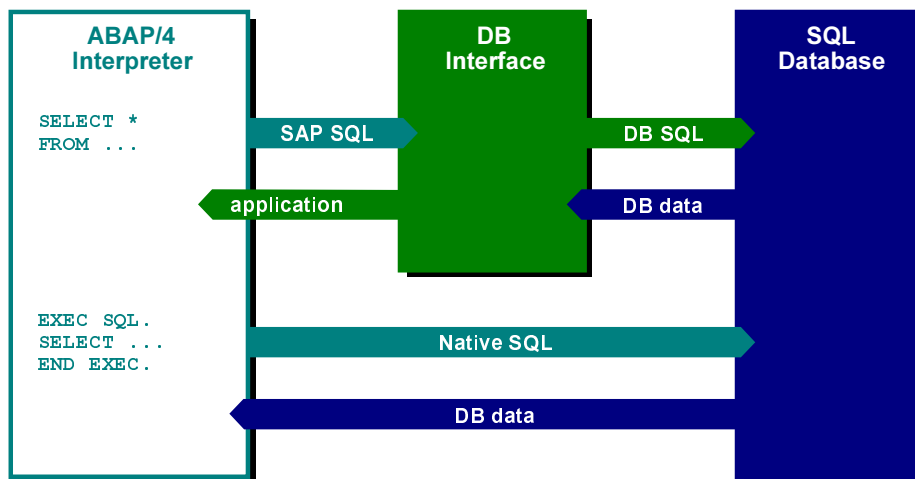
To use this kind of SQL statement in an ABAP program, you must use Native SQL (see [Using Native SQL Statements in an ABAP Program \[Page 599\]](#)).

To avoid incompatibilities between different database tables and also to make ABAP programs independent of the database system in use, SAP has created a set of separate SQL statements called Open SQL. Open SQL contains a subset of standard SQL statements as well as some enhancements which are specific to SAP. Using Open SQL enables you to access any database tables available to the SAP system, regardless of the manufacturer.

Database Tables and SQL Concepts

SAP Open SQL contains no DDL statements. The [ABAP Dictionary \[Ext.\]](#) carries out all tasks related to creating, changing and managing database tables in the R/3 System. Neither does SAP Open SQL contain DCL statements - its corresponding functions (locks, and so on), are managed centrally in the R/3 System.

The difference between Open SQL and Native SQL is shown in the following diagram:



A database interface translates SAP's Open SQL statements into SQL commands specific to the database in use. Native SQL statements access the database directly.

Open SQL keywords

Keyword	Use
SELECT	Reading Data from Database Tables [Page 542]
INSERT	Adding Lines to Database Tables [Page 571]
UPDATE	Changing Lines in Database Tables [Page 576]
MODIFY	Adding or changing lines [Page 581]
DELETE	Deleting Lines from Database Tables [Page 584]
OPEN CURSOR, FETCH, CLOSE CURSOR	Reading Lines of Database Tables Using a Cursor [Page 588]
COMMIT WORK. ROLLBACK WORK	Writing or Undoing Changes to Database Tables [Page 594]

Database Tables and SQL Concepts

When using Open SQL statements in an ABAP program, you must ensure the following:

The database system being addressed must be supported by SAP.

The database tables being addressed must be defined in the ABAP Dictionary.

The following system fields play an important role in Open SQL operations:

- SY-SUBRC

As with other ABAP statements, the return code value in the system field SY-SUBRC indicates after **each** Open SQL operation whether or not the operation was successful. If an operation is successful, SY-SUBRC = 0. If an operation is unsuccessful, SY-SUBRC <> 0.

- SY-DBCNT

The value in the SY-DBCNT field indicates how many lines were affected by the operation or how many lines have already been processed.

Reading Data from Database Tables

To read data from a database table, use the SELECT statement.

Syntax

```
SELECT <result> FROM <source> [INTO <target>] [WHERE <condition>]
      [GROUP BY <fields>] [ORDER BY <sort_order>].
```

This statement has several basic clauses. Each clause is described in the following table.

Clause	Description
SELECT <result>	<p>The SELECT clause defines whether the result of the selection is a single line or a table, which columns are to be selected, and whether identical lines are to be excluded.</p> <p>Defining the Result of a Selection [Page 543]</p>
FROM <source>	<p>The FROM clause specifies the database table or view <source> from which the data is to be selected.</p> <p>Specifying the Database Table to be Read [Page 550]</p>
INTO <target>	<p>The INTO clause determines the target area <target> into which the selected data is to be read. It can also be placed before the FROM clause. If you do not specify an INTO clause, the system uses the table work area. The table work area is a header line which is automatically created by the TABLES statement.</p> <p>Specifying the Target Area for the Selected Data [Page 553]</p>
WHERE <condition>	<p>The WHERE clause specifies which lines are to be read by specifying conditions for the selection.</p> <p>Choosing the Lines to be Read [Page 559]</p>
GROUP BY <fields>	<p>The GROUP-BY clause produces a single line of results from groups of several lines. A group is a set of lines with identical values for each column listed in <fields>.</p> <p>Grouping Lines [Page 567]</p>
ORDER BY <sort_order>	<p>The ORDER-BY clause defines a sequence <sort_order> for the lines resulting from the selection.</p> <p>Specifying the Order of Lines [Page 568]</p>

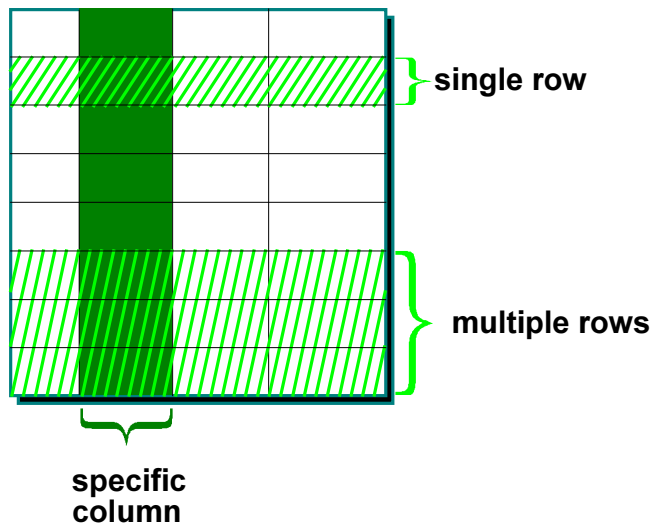
For important information about the performance of the SELECT statement and its clauses, see the keyword documentation.

Defining the Result of a Selection

Defining the Result of a Selection

The SELECT clause defines whether single or multiple lines are to be selected, whether duplicate lines are to be excluded, and which columns are to be selected.

The figure shows the possible selections:



The three variants of the SELECT clause are described in the following topics. There, you learn

[Selecting All Data from Several Lines \[Page 544\]](#)

[Selecting All Data from a Single Line \[Page 545\]](#)

[Selecting and Processing Data from Specific Columns \[Page 546\]](#)

Selecting All Data from Several Lines

To read all columns and multiple lines from database tables, use the SELECT statement in a loop as shown here.

Syntax

```
SELECT [DISTINCT] *.....
....
ENDSELECT.
```

You must terminate the loop with the ENDSELECT statement.

The loop reads the selected lines one by one and executes the ABAP statements inside the loop for each line read. The result of the SELECT loop is a table with exactly the same format as the database table being read.

The DISTINCT option automatically excludes duplicate lines.

If at least one line was read, the system field SY-SUBRC returns 0. If no line was read, the system field SY-SUBRC returns 4. The number of lines read is counted in the system field SY-DBCNT. After each SELECT statement is processed, SY-DBCNT is incremented by one.

You should only read all data of a line if you really need it. Using a list in the SELECT clause to read only the data that you need can be considerable faster (see [Selecting and Processing Data from Specific Columns \[Page 546\]](#)).

```
TABLES SPFLI.
```

```
SELECT * FROM SPFLI WHERE CITYFROM EQ 'FRANKFURT'.
```

```
...
WRITE: / SPFLI-CARRID, SPFLI-CONNID,
       SPFLI-CITYFROM, SPFLI-CITYTO.
```

```
...
ENDSELECT.
```

This SELECT loop reads all lines from table SPFLI which contain "FRANKFURT" in the SPFLI-CITYFROM field (as specified by the WHERE clause). Data from each line read is written to the output list by the WRITE statement in the loop.

The output list appears as follows:

LH	2402	FRANKFURT	BERLIN
LH	2436	FRANKFURT	BERLIN
LH	2462	FRANKFURT	BERLIN
LH	0400	FRANKFURT	NEW YORK
LH	0402	FRANKFURT	NEW YORK
SQ	0026	FRANKFURT	NEW YORK
LH	0454	FRANKFURT	SAN FRANCISCO
UA	0941	FRANKFURT	SAN FRANCISCO

Selecting All Data from a Single Line

Selecting All Data from a Single Line

To read all columns of a single line from a database table, use the SELECT statement as follows:

Syntax

```
SELECT SINGLE [FOR UPDATE] *..... WHERE <condition>.....
```

The result of this statement is a single line. To make sure you specify a line clearly, you must link all the fields which form the primary key of the database table by AND in the condition <condition> of the WHERE clause (for information about the WHERE clause, see [Choosing the Lines to be Read \[Page 559\]](#)).

You should read all data of a line only, if you really need all data. Using a list in the SELECT clause to read only the data that you need can be considerable faster (see [Selecting and Processing Data from Specific Columns \[Page 546\]](#)).

If the system does not find a line with the corresponding key, SY-SUBRC is set to 4. If it finds the specified line, SY-SUBRC is set to 0.

If you do not specify the entire primary key in the WHERE condition, the system reports a warning during the syntax check. Nevertheless, the program is executable and the system reads the first line from the database that matches the specified WHERE condition. This single line may not be unique.

You can use the FOR UPDATE option to lock the selected line in the database table. The program waits until it receives this lock. If the database detects or suspects a deadlock, a runtime error occurs.

The FOR UPDATE option is no substitute for using the SAP locking mechanism with ENQUEUE/DEQUEUE function modules. For example, all lines locked with FOR UPDATE are automatically unlocked when a new screen is displayed. To ensure that a line remains locked when a new screen is displayed, you must use the SAP locking mechanism. This is the only way to be certain that a line remains locked until the end of a transaction. For further information, see [Locking Database Objects During Program Execution \[Page 628\]](#).

TABLES SPFLI.

```
SELECT SINGLE * FROM SPFLI WHERE CARRID EQ 'LH'
      AND CONNID EQ '2407'.
WRITE: / SPFLI-CARRID, SPFLI-CONNID,
      SPFLI-CITYFROM, SPFLI-CITYTO.
```

This SELECT statement reads only the line containing "LH" in the CARRID field and "2407" in the CONNID field from SPFLI. Both fields form the primary key of SPFLI.

The output appears as follows:

LH	2407	BERLIN	FRANKFURT
----	------	--------	-----------

Selecting and Processing Data from Specific Columns

To read lines containing explicitly stated columns, or to get summary information about particular columns of the database table, use the SELECT statement with a list as follows:

Syntax

```
SELECT [SINGLE [FOR UPDATE]] [DISTINCT] <s1> <s2>... INTO...
```

where each <s_i> has either one of the following forms

- <a_i>
<a_i> is either a field of the database table or an aggregate expression of the form:
<aggregate>([DISTINCT] <a>)
For a description of aggregate expressions see below.
- <a_i> AS <b_i>
<b_i> is an alternative name for the i-th component of a structured target area. You can use this alternative name to write the result of reading or processing a specific column to a certain component <b_i> of the target area. To do this, you have to use the CORRESPONDING FIELDS option of the INTO clause (for further information about this option and an example see [Reading Data Component by Component \[Page 557\]](#).)

The DISTINCT option automatically excludes duplicate lines.

If you specify the SINGLE option, the result of the selection consists of the columns <a₁> <a₂>... of one single line. These lines are determined as described in [Reading all Data for a Single Line \[Page 545\]](#).

You can specify the columns at runtime by writing the SELECT statement as follows:

Syntax

```
SELECT [SINGLE [FOR UPDATE]] [DISTINCT] (<itab>)... INTO...
```

This statement works like the one above, if the internal table <itab> contains the list <s₁> <s₂>.... For this purpose, the line type of <itab> must be a type C field with a maximum length of 72. If the internal table is empty, the statement functions as if an asterisk (*) has been specified instead of (<itab>).

Working with a list in the SELECT clause can significantly enhance the performance compared to reading all data from a database table line (see [Runtime Measurement for Database Accesses \[Page 503\]](#)).

When working with a list in the SELECT clause you must use the INTO clause (see below).

Aggregate Expressions

By using aggregate expressions, you can extract characteristic data from a column <a> of the database table. Valid aggregate expressions are:

Selecting and Processing Data from Specific Columns

- MAX: returns the maximum value of the column <a>
- MIN: returns the minimum value of the column <a>
- AVG: returns the average value of the column <a>
- SUM: returns the sum value of the column <a>
- COUNT: counts values or lines as follows:
 - COUNT(DISTINCT <a>) returns the number of distinct values for the column <a>.
 - COUNT(*) returns the total number of lines in the selection.

You must include spaces between the parentheses and the arguments. The arithmetic operators AVG and SUM can only work with numeric fields.

List in the SELECT Clause and INTO Clause

If there is a list in the SELECT clause, you **must** use the INTO clause with the SELECT statement. As described in the relevant topic, you can use either a work area <wa> or an internal table <itab> as an argument. Note that, if the SELECT clause contains a list, the selected data is output left-justified to the target area according to the structure of the work area <wa> or the internal table <itab>. This is an exception from the general rule where the selected data is output left-justified to the target area according to the structure of the table work area, but regardless of the structure of the target area (for further information about the INTO clause, see [Specifying the Target Area for Read Data \[Page 553\]](#)).

With a list in the SELECT clause, you can use a list <f₁>, <f₂>, ... as an argument in the INTO clause:

```
SELECT <a1> <a2>.... INTO (<f1>, <f2>,....).
```

Both the list in the SELECT clause and the list in the INTO clause must contain the same number of elements. If possible, the values are converted to the data types of the target fields during the transport (for information about convertibility between database tables and ABAP data types, see the keyword documentation for the INTO clause).

If you work with a list in the SELECT clause, it is useful to work with the CORRESPONDING FIELDS option of the INTO clause (see [Reading Data Component by Component \[Page 557\]](#)). In that case, the target area does not need to contain the same number of elements as the list in the SELECT clause.

If the list in the SELECT clause contains one or more database fields apart from aggregate expressions, you must list all database fields in the GROUP-BY clause (for further information about this, see [Grouping Lines \[Page 567\]](#)).

```
TABLES SPFLI.
DATA: LIST(72) OCCURS 1,
      LINE(72).
LINE = ' CITYFROM CITYTO '.
APPEND LINE TO LIST.
SELECT DISTINCT (LIST)
      INTO CORRESPONDING FIELDS OF SPFLI FROM SPFLI.
```

Selecting and Processing Data from Specific Columns

```
WRITE: / SPFLI-CITYFROM, SPFLI-CITYTO.
ENDSELECT.
```

In this example, the columns CITYFROM and CITYTO are written to the internal table LIST and only these columns are selected from SPFLI. The result of the selection is written to the table work area SPFLI. For information about the CORRESPONDING FIELDS option of the INTO clause, see [Reading Data Component by Component \[Page 557\]](#). Duplicate lines are excluded because the DISTINCT option is used.

Suppose the following database table TEST consists of 10 lines:

COL_1	COL_2
1	3
2	1
3	5
4	7
5	2
6	3
7	1
8	9
9	4
10	3

To extract and process column data, you have the following program:

```
TABLES TEST.
```

```
DATA RESULT TYPE P DECIMALS 2.
```

```
SELECT <agg> ( [DISTINCT] COL_2) INTO RESULT FROM TEST.
```

```
WRITE RESULT.
```

The following table shows the results of this program according to different combinations of aggregate expressions <agg> and the DISTINCT option.

Aggregate expression used	DISTINCT option included	Result
MAX	no	9.00
MAX	yes	9.00
MIN	no	1.00
MIN	yes	1.00
AVG	no	3.80
AVG	yes	4.43
SUM	no	38.00

Selecting and Processing Data from Specific Columns

SUM	yes	31.00
COUNT	yes	7.00
COUNT(*)	---	10.00

Specifying the Database Table to be Read

To specify the database table or view to be read with a SELECT statement, you use the FROM clause. In the following, views are included in the term database table. For SELECT statements it is the same, whether the Dictionary objects specified in the TABLES statement are tables or views.

You learn

[Specifying the Database Table Name in the Program \[Page 551\]](#)

[Specifying the Database Table Name at Runtime \[Page 552\]](#)

Specifying the Database Table Name in the Program

Specifying the Database Table Name in the Program

To specify the database table or view you want to read in the program, use the following form of the FROM clause:

Syntax

```
..... FROM <dbtab> [CLIENT SPECIFIED] [BYPASSING BUFFER]
          [UP TO <n> ROWS].....
```

The database table or view <dbtab> must be available to the ABAP Dictionary, and you must include a corresponding TABLES statement in the ABAP program.

The CLIENT SPECIFIED option activates the automatic client handling. For more information, see [Specifying a Client for Processing Database Tables \[Page 596\]](#).

The BYPASSING BUFFER option allows you to read the database directly without reading the SAP table buffer.

This option is important if you want to ensure that you are dealing with the most recently updated version of the database table.

When you define a table in the ABAP Dictionary, you can specify that the SAP system should use its own local buffer for that table. This buffer is updated asynchronously. Since the SELECT statement normally uses this buffer, it does not necessarily use the most recently updated version of the database. To ensure that you have the most recent version, use the BYPASSING BUFFER option.

If you want to read only up to <n> lines from the database table <dbtab>, use the optional specification UP TO <n> ROWS. If <n> = 0, the system reads all lines. If <n> is less than 0, a runtime error occurs.

If you combine the UP TO <n> ROWS option with the ORDER BY clause, the system first sorts the lines and then processes the first <n> lines (for further information about the ORDER BY clause, see [Specifying the Sequence of the Lines \[Page 568\]](#)).

```
TABLES SPFLI.
SELECT * FROM SPFLI.
.....
ENDSELECT.
```

In this example, all lines are read from database table SPFLI.

Specifying the Database Table Name at Runtime

Specifying the Database Table Name at Runtime

You can specify the name of the database table at runtime. To do so, you use the following form of the FROM clause:

Syntax

```
.....FROM (<dbtabname>) [CLIENT SPECIFIED] [BYPASSING BUFFER]
          [UP TO <n> ROWS].. INTO <target>.....
```

This form of the FROM clause only works in conjunction with the INTO clause (see [Specifying the Target Area for Read Data \[Page 553\]](#)).

The contents of the field <dbtabname> determines the name of the database table. In this case, the program does not have to contain a TABLES statement. The options CLIENT SPECIFIED, BYPASSING BUFFER, and UP TO <n> ROWS are the same as those used for specifying the database table name in the program (see [Specifying the Database Table Name in the Program \[Page 551\]](#)).

```
DATA: BEGIN OF WA,
      LINE(240),
      END OF WA.

DATA NAME(10).

NAME = 'SPFLI'.

SELECT * FROM (NAME) INTO WA.
  WRITE: / WA-LINE.
ENDSELECT.
```

The database table name SPFLI is assigned to the character field NAME. The SELECT statement reads all the lines from SPFLI into the target area WA. In this example, WA does not have the same structure as SPFLI, and each line is automatically converted into a text field. For more information about this, see [Specifying the Target Area for the Selected Data \[Page 553\]](#) and [Convertibility of Structures \[Page 227\]](#).

You cannot write NAME = 'spfli' instead of NAME = 'SPFLI' because the system would not find the database table SPFLI in the ABAP Dictionary.

The following section of the output list shows that some columns are wrongly interpreted by writing the data to a text field:

000AA 0017NEW YORK	JFKSAN FRANCISCO	SF0060100133000163100###
000AA 0064SAN FRANCISCO	SFONEW YORK	JFK052100090000172100###
000DL 1699NEW YORK	JFKSAN FRANCISCO	SF0062200171500203700###
000DL 1984SAN FRANCISCO	SFONEW YORK	JFK052500100000182500###
000LH 0400FRANKFURT	FRANEW YORK	JFK082400101000113400##\KM
000LH 0402FRANKFURT	FRANEW YORK	EWR083500133000150500##\KM
000LH 0454FRANKFURT	FRASAN FRANCISCO	SF0122000101000123000###
000LH 0455SAN FRANCISCO	SF0FRANKFURT	FRA133000150000103000###

Specifying the Target Area for the Selected Data

Specifying the Target Area for the Selected Data

To specify the target area for the selected data, you use the INTO clause of the SELECT statement. You only need to use this clause if you want to specify a different target area than the table work area. The table work area is generated automatically by the TABLES statement. If you want to read lines directly using the database cursor, enter the INTO clause in the FETCH statement (see [Reading Lines from a Database Table Using a Cursor \[Page 588\]](#)).

The INTO clause has three main variants. Two of these are for reading data into a work area and the other is for reading data into internal tables. These variants are described in the following topics.

You learn how to

[Read Data into a Work Area \[Page 554\]](#)

[Read Data into an Internal Table \[Page 555\]](#)

[Read Data Component by Component \[Page 557\]](#)

The third variant is used for reading data into a list. It only works in conjunction with a list in the SELECT clause, and is described in the section [Reading and Processing Data from Particular Columns \[Page 546\]](#). If you are using the INTO clause in conjunction with a list in the SELECT clause, the selected columns are output left-justified according to the structure of the target area.

In the case of SELECT statements without a list (SELECT *), the selected data is output left-justified to the target area. The system writes the data according to the structure of the table work area, but regardless of the structure of the target area. To ensure that you are able to access the individual columns of the database table, you should use only target areas which have the same structure as the database table. The target area must be at least as large as the line to be read into it.

The data types in the ABAP Dictionary are not the same as the data types in the ABAP programming language. If you specify a list in the SELECT clause or use the CORRESPONDING FIELDS option in the INTO clause, the ABAP Dictionary fields of the database table must be convertible to the target fields of the ABAP programming language. For a list of convertible data types, see the keyword documentation of the INTO clause.

Reading Data into a Work Area

You can read data from a database table into a work area (usually a field string) which is different from the default work area defined in the TABLES statement.

To do this, specify the work area in the INTO clause of the SELECT statement as shown below:

Syntax

```
SELECT... INTO <wa>.....
```

For work area <wa>, you have to declare a data object which is at least as large as the line to be read into it.

```
TABLES SPFLI.  
DATA WA LIKE SPFLI.  
SELECT * FROM SPFLI INTO WA.  
  WRITE: / WA-CITYFROM, WA-CITYTO.  
ENDSELECT.
```

In this example, the work area WA has the same structure as the database table SPFLI because its data type is defined by LIKE SPFLI in the DATA statement. The use of the INTO clause in the SELECT loop causes the work area WA to be filled instead of the standard work area SPFLI specified in the TABLES statement. All fields of SPFLI, i.e. SPFLI-CITYFROM and SPFLI-CITYTO, stay blank.

Reading Data into an Internal Table

Reading Data into an Internal Table

You can write the result set of a line selection from a database table into an internal table in a single operation.

To do this, specify the internal table in the INTO clause of the SELECT statement as shown below:

Syntax

```
SELECT..... INTO TABLE <itab>.
```

In this case, SELECT does not start a loop, and **no** ENDSELECT statement is allowed.

If the internal table <itab> is not empty, its contents are overwritten with the data read by the SELECT statement.

```
TABLES SPFLI.  
DATA ITAB LIKE SPFLI OCCURS 10 WITH HEADER LINE.  
SELECT * FROM SPFLI INTO TABLE ITAB  
      WHERE CARRID = 'LH'.  
  
LOOP AT ITAB.  
  WRITE: / ITAB-CONNID, ITAB-CARRID.  
ENDLOOP.
```

In this example, all lines from the database table SPFLI in which CARRID field contains "LH" are read into the internal table ITAB, where they can be processed further.

When specifying an internal table as the target area for selected lines, you can process the lines in packages or append lines to internal tables instead of overwriting the table contents. These options are described below.

Processing Lines in Packages

If you want to read the selected lines into an internal table in packages of predefined size, use the PACKAGE SIZE option of the INTO clause as follows:

Syntax

```
SELECT *..... INTO TABLE <itab> PACKAGE SIZE <n>.....
```

This statement opens a loop. You must terminate this loop with the ENDSELECT statement. For every package of <n> lines read, the system passes through the loop once. If <n> is less than or equal to 0, a runtime error occurs.

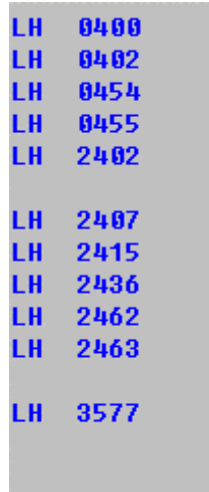
```
Outside the SELECT loop, the contents of the internal table are undetermined. If,  
therefore, you want to process the selected lines further, you must program the  
corresponding ABAP statements within the loop.
```

```
TABLES SPFLI.  
DATA ITAB LIKE SPFLI OCCURS 5 WITH HEADER LINE.
```

```
SELECT * FROM SPFLI INTO TABLE ITAB PACKAGE SIZE 5  
      WHERE CARRID = 'LH'.
```

```
LOOP AT ITAB.  
  WRITE: / ITAB-CARRID, ITAB-CONNID.  
ENDLOOP.  
  
SKIP 1.  
  
ENDSELECT.
```

In this example, all lines from the database table SPFLI in which CARRID field contains "LH" are read into the internal table ITAB in packages of 5 lines. Inside the SELECT loop, there is a second loop which writes these packages to the output list. The output list appears as follows:



LH	0400
LH	0402
LH	0454
LH	0455
LH	2402
LH	2407
LH	2415
LH	2436
LH	2462
LH	2463
LH	3577

Appending Lines to Internal Tables

To prevent the contents of an internal table from being overwritten, you can append the selected lines to the table instead. To do this, use the APPENDING clause instead of the INTO clause, as shown below:

Syntax

```
SELECT..... APPENDING TABLE <itab>.....
```

The only difference between this and the INTO clause described above is that lines are appended to the internal table <itab> instead of their contents. You can also use the PACKAGE SIZE option in this statement.

Reading Data Component by Component

Reading Data Component by Component

To read data component by component into the target area, you use the CORRESPONDING FIELDS option of the INTO clause. The syntax is as follows:

Syntax

For reading data into a work area:

```
SELECT... INTO CORRESPONDING FIELDS OF <wa>.....
```

For reading data into an internal table:

```
SELECT... INTO CORRESPONDING FIELDS OF TABLE <itab>.....
```

For appending data to an internal table:

```
SELECT... APPENDING CORRESPONDING FIELDS OF TABLE <itab>.....
```

These statements do not place all the fields of the selected lines in the target area. The system transfers only the contents of columns for which the target area has components with the same name to the corresponding components in the target area. If possible, the values are converted to the data types of the target fields during the transport (for information about convertibility between database tables and ABAP data types, see the keyword documentation for the INTO clause).

Working with the CORRESPONDING FIELDS option of the INTO clause does **not** limit the amount of data read from the database, but only the amount of data that is read from the resulting set into the ABAP program. To restrict the data selected, you must also include a list in the SELECT clause (see [Reading and Processing Data for Particular Columns \[Page 546\]](#)).

```
TABLES SPFLI.
```

```
DATA: BEGIN OF WA,
      NUMBER TYPE I VALUE 1,
      CITYFROM LIKE SPFLI-CITYFROM,
      CITYTO  LIKE SPFLI-CITYTO,
      END OF WA.
```

```
SELECT * FROM SPFLI INTO CORRESPONDING FIELDS OF WA.
WRITE: / WA-NUMBER, WA-CITYFROM, WA-CITYTO.
ENDSELECT.
```

The output appears as follows:

1	NEW YORK	SAN FRANCISCO
1	SAN FRANCISCO	NEW YORK
1	NEW YORK	SAN FRANCISCO
1	SAN FRANCISCO	NEW YORK
1	FRANKFURT	NEW YORK
1	FRANKFURT	NEW YORK
1	FRANKFURT	SAN FRANCISCO

In this example, the system transfers only the columns CITYFROM and CITYTO from the selected lines of the database table SPFLI to WA. The component NUMBER of WA remains unchanged.

TABLES SBOOK.

DATA: BEGIN OF LUGGAGE,
 AVERAGE TYPE P DECIMALS 2,
 SUM TYPE P DECIMALS 2,
 END OF LUGGAGE.

SELECT AVG(LUGGWEIGHT) AS AVERAGE
 SUM(LUGGWEIGHT) AS SUM
 INTO CORRESPONDING FIELDS OF LUGGAGE FROM SBOOK.

WRITE: / 'Average:', LUGGAGE-AVERAGE, / 'Sum:', LUGGAGE-SUM.

This example calculates the average value and the sum of the fields LUGGWEIGHT for all lines in the database table SBOOK. The results of the aggregate expressions AVG and SUM are written with alternative names to the components AVERAGE and SUM of the structure LUGGAGE. For further information about aggregate expressions and alternative names, see [Reading and Processing Data from Particular Columns \[Page 546\]](#).

Choosing the Lines to be Read

Choosing the Lines to be Read

To access only those lines from a database table which meet certain conditions, use the WHERE clause of the SELECT statement.

You can

[Specifying Conditions for Line Selection in the Program \[Page 560\]](#)

[Specifying Conditions for Line Selection at Runtime \[Page 563\].](#)

The WHERE clause described in this topic is used in Open SQL not only in the SELECT statement, but also in the UPDATE, MODIFY, and DELETE statements.

Specifying Conditions for Line Selection in the Program

Specifying Conditions for Line Selection in the Program

To specify conditions for line selection in the program, use the WHERE clause as shown below:

Syntax

..... WHERE <condition>.....

Basic WHERE Conditions

There are six basic conditions you can specify to limit line selection. They are described below:

1. <f> <operator> <g>

<f> is the name of a database field (column of a database table) without the table name as a prefix and <g> is the name of any field or a literal. The field names and the operator must be separated by blanks.

For <operator>, you can use the following characters or character strings:

<operator>	Meaning
EQ	equal to
=	equal to
NE	not equal to
<>	not equal to
><	not equal to
LT	less than
<	less than
LE	less than or equal to
<=	less than or equal to
GT	greater than
>	greater than
GE	greater than or equal to
>=	greater than or equal to

..... WHERE CARRID = 'UA'.

Selects all lines in which the CARRID field contains "UA".

..... WHERE NUM GE 15.

Selects all lines in which the NUM field contains numbers greater or equal to 15.

..... WHERE CITYFROM NE 'FRANKFURT'.

Selects all lines in which the CITYFROM field contains character strings not equal to "FRANKFURT".

2. <f> [NOT] BETWEEN <g₁> AND <g₂>

The value of the database field <f> must [not] fall between the values of the fields or literals <g₁> and <g₂> to meet this condition.

Specifying Conditions for Line Selection in the Program

..... WHERE NUM BETWEEN 15 AND 45.

Selects all lines in which the NUM field contains numbers between 15 and 45.

..... WHERE NUM NOT BETWEEN 1 AND 99.

Selects all lines in which the NUM field contains numbers not between 1 and 99.

..... WHERE NAME NOT BETWEEN 'A' AND 'H'.

Selects all lines in which the NAME field contains character strings which do not fall alphabetically between "A" and "H".

3. <f> [NOT] LIKE <g> [ESCAPE <h>]

This condition can only be used with character type fields.

The value of the database field <f> must [not] correspond to the pattern in <g> in order to satisfy this condition. You can use the following two wildcards when specifying <g>:

(underscore) represents a single character

(percent sign) represents any character string,

including empty strings.

For example, ABC_EFG% matches the strings ABCxEFGxyz and ABCxEFG, but not ABCEFGxyz.

..... WHERE CITY LIKE '%town%'.

Selects all city names including "town".

..... WHERE NAME NOT LIKE '_n%'.

Selects only names which do not have an "n" as the second letter.

If you want to use the two wildcard characters explicitly in the comparison, use the ESCAPE option. ESCAPE <h> specifies an escape symbol <h>. If preceded by <h>, the wildcards and the escape symbol itself lose their usual function within the pattern <g>.

..... WHERE FUNCNAME LIKE 'EDIT#_%' ESCAPE '#'.

Selects all function names beginning with "EDIT_" are selected.

Using the wildcards _ and % in the LIKE condition corresponds to the SQL standards. However, the ABAP comparison operators CP and NP recognize different wildcards (+ and *) (for further information about wildcards, see [Comparing Strings and Numeric Text \[Page 238\]](#)).

4. <f> [NOT] IN (<g_{1n}

The value of the database field <f> must [not] be equal to one of the values in the list in parentheses to satisfy this condition.

You **cannot** use spaces between the brackets and the comparison fields <g_i> in this variant. However, you can use spaces in the list between the fields themselves.

Specifying Conditions for Line Selection in the Program

..... WHERE CITY IN ('Berlin', 'New York', 'London').
Selects the cities "Berlin", "New York", and "London".

..... WHERE CITY NOT IN ('Frankfurt', 'Rome').
Selects the cities except "Frankfurt" and "Rome".

5. <f> IS [NOT] NULL

The value of the database field <f> must [not] be equal to the NULL value.

6. <f> [NOT] IN <seltab>

The value of the database field <f> must [not] correspond to the condition specified in the selection table <seltab> to satisfy this condition. The selection table is a special internal table which the report user can fill on the selection screen. Usually, you create a selection table with SELECT-OPTIONS or RANGES, but you can also define it as described in [Creating and Processing Internal Tables \[Page 260\]](#). For information about the structure of selection tables and examples, see [SELECT-OPTIONS - Defining Selection Criteria \[Page 815\]](#).

Combining Conditions with Logical Link Operators

You can combine the six basic WHERE conditions in any order using the logical link operators AND, OR, and NOT.

If you want to specify several conditions which must all be satisfied, combine them using AND as shown below:

....WHERE <condition₁> AND <condition₂> AND <condition₃> AND...

If you want to specify several conditions, at least one of which has to be satisfied, combine them using OR as shown below:

....WHERE <condition₁> OR <condition₂> OR <condition₃> OR...

If you want to select only those table entries which do **not** satisfy a specified condition, use NOT to invert conditions, as shown below:

....WHERE NOT <condition>

NOT takes priority over AND, and AND takes priority over OR. However, you can use parentheses to define the processing sequence explicitly. These parentheses must be enclosed by blanks.

.....WHERE (NUMBER = '0001' OR NUMBER = '0002') AND
NOT (COUNTRY = 'F' OR COUNTRY = 'USA').

In this example, only the lines in which NUMBER field contains "0001" or "0002" and COUNTRY field does not contain "F" or "USA", are selected.

Specifying Conditions for Line Selection at Runtime

Specifying Conditions for Line Selection at Runtime

You can specify either the full condition or part of the condition for line selection at runtime.

Specifying conditions at runtime requires more CPU-time than specifying them in the program since the system cannot perform the syntax check or generate the internal control blocks until runtime. Also, dynamic WHERE conditions work only with the SELECT statement.

Specifying the Full Condition at Runtime

Syntax

SELECT.....WHERE (<itab>).....

You must specify your conditions in an internal table <itab> containing only one field of type C with a maximum length of 72. The table name must be specified in parentheses, but without any blanks between the parentheses and the name.

You can use any conditions with the same syntax as described in the table <itab> in [Specifying Conditions for Line Selection in the Program \[Page 560\]](#). However, the following exceptions apply:

- Use literals. Variables are not allowed.
- Do not use the operator IN with a selection table.

The internal table can also be left empty.

```
TABLES SPFLI.
DATA ITAB(72) OCCURS 10 WITH HEADER LINE.
PARAMETERS: CITY1(10) TYPE C, CITY2(10) TYPE C.
CONCATENATE 'CITYFROM = ' CITY1 ' ' INTO ITAB.
APPEND ITAB.
CONCATENATE 'OR CITYFROM = ' CITY2 ' ' INTO ITAB.
APPEND ITAB.
CONCATENATE 'OR CITYFROM = ' 'BERLIN' ' ' INTO ITAB.
APPEND ITAB.
LOOP AT ITAB.
  WRITE ITAB.
ENDLOOP.
SKIP.
SELECT * FROM SPFLI WHERE (ITAB).
  WRITE / SPFLI-CITYFROM.
ENDSELECT.
```

When you start the program, the PARAMETERS statement causes a selection screen to appear. This is described under PARAMETERS - [Creating Input Fields for Variables \[Page 803\]](#).

Specifying Conditions for Line Selection at Runtime

In this example, the user is asked to enter the parameters CITY1 and CITY2 on the selection screen. Suppose the user of your program fills the input fields of the selection screen as follows:

CITY1	FRANKFURT
CITY2	NEW YORK

This would produce the following output on the screen:

```

CITYFROM = 'FRANKFURT'
OR CITYFROM = 'NEW YORK'
OR CITYFROM = 'BERLIN'

NEW YORK
NEW YORK
FRANKFURT
FRANKFURT
FRANKFURT
FRANKFURT
BERLIN
BERLIN
FRANKFURT
FRANKFURT
BERLIN
FRANKFURT
NEW YORK
FRANKFURT

```

The first three lines show the contents of the internal table ITAB. The output list shows that the only lines selected are those in which the CITYFROM field contains "FRANKFURT", "NEW YORK", or "BERLIN".

Specifying only Part of the Condition at Runtime

To specify part of a condition in a SELECT statement at runtime, use the WHERE clause as follows:

Syntax

```
SELECT.....WHERE <condition> AND (<itab>).....
```

Specify the first part of the condition <condition> as described in [Specifying Conditions for Line Selection in the Program \[Page 560\]](#).

You must append the part of the condition you want to specify in <itab> at runtime after <condition> using AND.

You specify the internal table <itab> as described above.

Specifying a List of Conditions

To specify a list of conditions at runtime to select a number of particular lines, use the following special variant of the WHERE clause in the SELECT statement:

Syntax

```
SELECT.....FOR ALL ENTRIES IN <itab> WHERE <condition>.....
```

Specifying Conditions for Line Selection at Runtime

In the condition <condition>, you can specify internal fields and literals as comparison values as before. You can also use columns of the internal table <itab> as comparison values. In the WHERE condition, these columns are used as placeholders.

The result set of this SELECT statement is the union of all result sets of the SELECT statements which result from replacing the placeholders on each line by the appropriate values from <itab>. If <itab> is empty, the addition FOR ALL ENTRIES is disregarded, and all entries are read.

Duplicate lines are eliminated from the result set.

Database fields and the associated comparison fields of the internal table must have the same type and length.

Do not use the operators LIKE, BETWEEN, and IN in comparisons between database fields and table fields.

If you are using this variant of the WHERE clause, do not use the ORDER BY clause.

TABLES SPFLI.

```
DATA: BEGIN OF ITAB OCCURS 10,
      CITYFROM LIKE SPFLI-CITYFROM,
      CITYTO   LIKE SPFLI-CITYTO,
      END OF ITAB.
```

```
ITAB-CITYFROM = 'FRANKFURT'.
ITAB-CITYTO   = 'BERLIN'.
APPEND ITAB.
```

```
ITAB-CITYFROM = 'NEW YORK'.
ITAB-CITYTO   = 'SAN FRANCISCO'.
APPEND ITAB.
```

```
SELECT * FROM SPFLI FOR ALL ENTRIES IN ITAB WHERE
      CITYFROM = ITAB-CITYFROM AND
      CITYTO   = ITAB-CITYTO.
WRITE: / SPFLI-CITYFROM, SPFLI-CITYTO.
ENDSELECT.
```

This example selects from SPFLI the union of lines where

- the CITYFROM field contains "FRANKFURT" and the CITYTO field contains "BERLIN"
- the CITYFROM field contains "NEW YORK" and the CITYTO field contains "SAN FRANCISCO"

You can use the option FOR ALL ENTRIES to replace nested select loops by operations on internal tables. This can significantly improve the performance for large sets of selected data.

Assume the following nested SELECT loops:

```
SELECT * FROM SPFLI WHERE CARRID = 'LH'.
  SELECT * FROM SBOOK WHERE CARRID = SPFLI-CARRID
    AND FLDATE = '19950130'.
```

Specifying Conditions for Line Selection at Runtime

```
WRITE: / SBOOK-CARRID, SBOOK-CONNID, SBOOK-FLDATE.  
ENDSELECT.  
ENDSELECT.
```

The following program segment, using two internal tables, has the same functionality:

```
DATA: TAB_SPFLI LIKE SPFLI OCCURS 100 WITH HEADER LINE,  
      TAB_SBOOK LIKE SBOOK OCCURS 100 WITH HEADER LINE.  
  
SELECT * FROM SPFLI INTO TABLE TAB_SPFLI  
      WHERE CARRID = 'LH'.  
  
SELECT * FROM SBOOK INTO TABLE TAB_SBOOK  
      FOR ALL ENTRIES IN TAB_SPFLI  
      WHERE CARRID = TAB_SPFLI-CARRID  
      AND FLDATE = '19950130'.  
  
LOOP AT TAB_SPFLI.  
  LOOP AT TAB_SBOOK.  
    WRITE: / TAB_SBOOK-CARRID, TAB_SBOOK-CONNID,  
           TAB_SBOOK-FLDATE.  
  ENDLOOP.  
ENDLOOP.
```

The second SELECT statement links the two internal tables via the FOR ALL ENTRIES option. For a table which is not empty, it is a short form of the following LOOP/ENDLOOP block:

```
LOOP AT TAB_SPFLI.  
  SELECT * FROM SBOOK APPENDING TABLE TAB_SBOOK  
        WHERE CARRID = TAB_SPFLI-CARRID  
        AND FLDATE = '19950130'.  
ENDLOOP.
```

Note the use of APPENDING in the SELECT statement.

If TAB_SPFLI is empty, the addition FOR ALL ENTRIES is disregarded.

Grouping Lines

Grouping Lines

To summarize the contents of a group of lines from a database table in to a single line, use the GROUP BY clause in the SELECT statement as follows:

Syntax

```
SELECT [DISTINCT] <a1> <a2>..  
      FROM clause INTO clause GROUP BY <f1> <f2>....
```

A group consists of the lines with the same values in the columns listed in <f₁> <f₂>.....

You can specify the columns at runtime by writing the SELECT statement as follows:

Syntax

```
SELECT [DISTINCT] <a1> <a2>..  
      FROM clause INTO clause GROUP BY (<itab>)
```

This statement works like the one above if the internal table <itab> contains the list <f₁> <f₂>.... The lines of the internal table must consist of a type C field with maximum length 72.

The GROUP BY <f₁> <f₂>... clause only works if you use a list in the SELECT clause (see [Reading and Processing Data from Particular Columns \[Page 546\]](#)).

If you use aggregate functions with the GROUP BY clause, the GROUP BY clause must include all of the database fields that appear outside of aggregate functions in the list in the SELECT clause.

```
TABLES SFLIGHT.
```

```
DATA CARRID LIKE SFLIGHT-CARRID.
```

```
DATA: MINIMUM TYPE P DECIMALS 2, MAXIMUM TYPE P DECIMALS 2.
```

```
SELECT CARRID MIN( PRICE) MAX( PRICE)  
      INTO (CARRID, MINIMUM, MAXIMUM) FROM SFLIGHT  
      GROUP BY CARRID.
```

```
WRITE: / CARRID, MINIMUM, MAXIMUM.
```

```
ENDSELECT.
```

In the example, the lines of database table SFLIGHT are grouped according to the contents of field CARRID. The smallest and greatest values of the PRICE field are displayed as follows for each group:

AA	555,56	555,56
DL	555,56	555,56
LH	555,56	8.000,00
SQ	1.500,00	1.500,00
UA	555,56	1.500,00

Specifying the Order of Lines

When you read sets of lines, you can define the order in which the lines are presented to the ABAP program by using the ORDER BY clause of the SELECT statement.

If you do not use the ORDER BY clause, the order of the selected lines is undefined.

You can either sort the lines by primary key or by explicitly specified fields.

Sorting by Primary Key

To sort the selected lines by the primary key, use the following variant of the ORDER BY clause:

Syntax

```
SELECT *..... ORDER BY PRIMARY KEY.
```

If you use the ORDER BY PRIMARY KEY option, the system sorts the lines by the primary key in ascending order.

Sorting by Specified Fields

You can sort the selected lines by fields other than the primary key. To do this, use the following syntax:

Syntax

```
.. ORDER BY <f1> [ASCENDING|DESCENDING] <f2> [ASCENDING|DESCENDING]...
```

This sorts the lines by the specified table fields <f₁>, <f₂>,.... You can specify the sort sequence explicitly for each table field by specifying the options ASCENDING or DESCENDING after each field name. The standard sort sequence is in ascending order.

If you specify more than one field, the system first sorts the lines by <f₁>, then by <f₂> and so on.

You can also specify the fields at runtime by writing the ORDER BY clause as follows:

Syntax

```
.. ORDER BY (<itab>)
```

This statement works like the one above if the internal table <itab> contains the list <f₁> [ASCENDING|DESCENDING] <f₂>.... Here, the lines of <itab> must be type C with a maximum length of 72.

```
TABLES SPFLI.
```

```
SELECT * FROM SPFLI ORDER BY CITYFROM ASCENDING  
                           CITYTO  DESCENDING.  
WRITE: / SPFLI-CITYFROM, SPFLI-CITYTO.  
ENDSELECT.
```

In this example, the lines which have been selected are sorted in the first instance according to the contents of the CITYFROM field in ascending order, and then by the contents of the CITYTO field in descending order:

Specifying the Order of Lines

BERLIN	FRANKFURT
BERLIN	FRANKFURT
BERLIN	FRANKFURT
FRANKFURT	SAN FRANCISCO
FRANKFURT	SAN FRANCISCO
FRANKFURT	NEW YORK
FRANKFURT	NEW YORK
FRANKFURT	NEW YORK
FRANKFURT	BERLIN
FRANKFURT	BERLIN
FRANKFURT	BERLIN
NEW YORK	SAN FRANCISCO
NEW YORK	SAN FRANCISCO
NEW YORK	SAN FRANCISCO
ROM	FRANKFURT
SAN FRANCISCO	NEW YORK
SAN FRANCISCO	NEW YORK
SAN FRANCISCO	FRANKFURT
SAN FRANCISCO	FRANKFURT

Changing the Contents of Database Tables

You can change the contents of a database table with the following operations:

Operation	ABAP keywords
Adding lines	<p>INSERT</p> <p>You use the INSERT statement to add new lines, i.e. lines with a primary key which does not already exist in the database table.</p> <p>Adding Lines to Database Tables [Page 571]</p>
Changing lines	<p>UPDATE</p> <p>You use the UPDATE statement to change existing lines in a database table i.e. lines with a primary key which already exists in the database table.</p> <p>Changing Lines in Database Tables [Page 576]</p>
Adding or changing lines	<p>MODIFY</p> <p>You use the MODIFY statement to add new lines if no line with the primary key of the line to be added exists. Otherwise, you change the existing line</p> <p>Adding or changing lines [Page 581]</p>
Deleting lines	<p>DELETE</p> <p>You use the DELETE statement to delete lines from a database table.</p> <p>Deleting Lines from Database Tables [Page 584]</p>

Adding Lines to Database Tables

Adding Lines to Database Tables

To add new lines to database tables, use the INSERT statement. The INSERT statement enables you to insert either a single line or several lines from a particular internal table.

If you are not sure whether you can use the INSERT statement because you do not know whether the primary key of the line you want to insert already exists or not, use the MODIFY statement.

You can only insert lines into a database table using a view if the view refers to a single table. The view's maintenance status must be defined in the ABAP Dictionary as having "no restriction".

You can

[Add Single Lines \[Page 572\]](#)

[Add Several Lines from an Internal Table \[Page 574\]](#)

Adding a Single Line

To add a single line to a database table, you can use one of the following variants of the INSERT statement.

Basic Form

The basic form of the INSERT statement is as follows:

Syntax

```
INSERT INTO <dbtab> [CLIENT SPECIFIED] VALUES <wa>.
```

The contents of the work area <wa> are written to the database table <dbtab>. You must declare the database table with the TABLES statement in the program.

The work area <wa> must be at least as long as the table work area of <dbtab>. To ensure that the work area has the same structure as the database table, you should define it with the DATA or TYPES statement using the LIKE <dbtab> option (see [The Basic Form of the DATA Statement \[Page 120\]](#)).

If the database table does not already contain a line with the same primary key as specified in the work area, the operation is completed successfully and SY-SUBRC is set to 0. Otherwise, it is set to 4.

The CLIENT SPECIFIED option activates the automatic client handling. For more information, see [Specifying a Client for Processing Database Tables \[Page 596\]](#).

If you want to specify the name of the database table at runtime, use the following syntax instead:

```
INSERT INTO (<dbtabname>) [CLIENT SPECIFIED] VALUES <wa>.
```

You specify the database table (<dbtabname>) as described for the SELECT statement in [Specifying the Database Table to be Read \[Page 550\]](#).

Short Forms

You can replace the above syntax with the following short forms:

Syntax

```
INSERT <dbtab> [CLIENT SPECIFIED] FROM <wa>.
```

or if you want to specify the database table name at runtime:

```
INSERT (<dbtabname>) [CLIENT SPECIFIED] FROM <wa>.
```

The statements produce the same results as the basic form.

An even shorter form is:

Syntax

```
INSERT <dbtab> [CLIENT SPECIFIED].
```

In this case, you do not specify a work area <wa>. Instead, the contents of the table work area <dbtab> are written to the database table.

If you specify the database table name at runtime, you must use the option FROM <wa>. It is not possible to use the table work area.

```
TABLES SPFLI.
```

Adding a Single Line

```
DATA WA LIKE SPFLI.  
WA-CARRID = 'LH'.  
WA-CITYFROM = 'WASHINGTON'.  
.....  
INSERT INTO SPFLI VALUES WA.  
WA-CARRID = 'UA'.  
WA-CITYFROM = 'LONDON'.  
.....  
INSERT SPFLI FROM WA.  
SPFLI-CARRID = 'LH'.  
SPFLI-CITYFROM = 'BERLIN'.  
.....  
INSERT SPFLI.
```

In this example, the work area WA is defined with the same structure as the database table SPFLI. Values are assigned to WA and the work area is inserted into the database table SPFLI using the three possible variants of the INSERT statement.

Instead of INSERT SPFLI in the last line, you could also use the longer forms INSERT SPFLI FROM SPFLI or INSERT INTO SPFLI VALUES SPFLI.

Adding Several Lines from an Internal Table

To add several lines to a database table from an internal table using the INSERT statement, use the following syntax:

Syntax

```
INSERT <dbtab> [CLIENT SPECIFIED] FROM TABLE <itab>
[ACCEPTING DUPLICATE KEYS].
```

If you want to specify the name of the database table at runtime, use the following syntax instead:

```
INSERT (<dbtabname>) [CLIENT SPECIFIED] FROM TABLE <itab>
[ACCEPTING DUPLICATE KEYS].
```

This writes **all** lines of the internal table <itab> to the database table in one single operation.

The lines of the internal table must have at least the same length as the database table. To ensure that the internal table has the same structure as the database table, you should define it with the DATA or TYPES statement using the LIKE <dbtab> option (see [The Basic Form of the DATA Statement \[Page 120\]](#)).

If the operation is successfully concluded, SY-SUBRC is set to 0. If some lines cannot be inserted because at least one line with the same primary key already exists in <dbtab>, the system reacts with a runtime error. You can prevent this by using the ACCEPTING DUPLICATE KEYS option. This throws away the corresponding lines and sets SY-SUBRC to 4. If you want to change the existing lines instead of skipping them, use the MODIFY statement (see [Adding or Changing Lines \[Page 581\]](#)).

The system field SY-DBCNT always contains the number of inserted lines, regardless of the value in SY-SUBRC.

The procedure for naming the database table and the CLIENT SPECIFIED option are the same as in [Adding Individual Lines \[Page 572\]](#).

Working on sets of lines is always preferable to working on single lines because it is considerably more efficient.

```
TABLES SPFLI.
```

```
DATA ITAB LIKE SPFLI OCCURS 10 WITH HEADER LINE.
```

```
ITAB-CARRID = 'UA'. ITAB-CONNID = '0011'. ITAB-CITYFROM =..
APPEND ITAB.
```

```
ITAB-CARRID = 'LH'. ITAB-CONNID = '1245'. ITAB-CITYFROM =..
APPEND ITAB.
```

```
ITAB-CARRID = 'AA'. ITAB-CONNID = '4574'. ITAB-CITYFROM =..
APPEND ITAB.
```

```
.....
```

```
INSERT SPFLI FROM TABLE ITAB ACCEPTING DUPLICATE KEYS.
```

```
IF SY-SUBRC = 0.
```

```
.....
```

```
ELSEIF-SUBRC = 4.
```

Adding Several Lines from an Internal Table

```
.....  
ENDIF.
```

In this example, an internal table ITAB, which has the same structure as the database table SPFLI, is declared. ITAB is filled and then inserted straight into SPFLI. The success of the operation is checked by using IF statements and the program continues according to the results of the check.

Changing Lines in Database Tables

To change lines in a database table, use the UPDATE statement. This allows you to change either a single line or several lines.

If you are not sure whether you can use the UPDATE statement because you do not know whether the primary key of the line you want to insert already exists or not, you can use the MODIFY statement. The MODIFY statement changes existing lines and inserts lines which do not yet exist (see [Adding or Changing Lines \[Page 581\]](#)).

You can only change lines of a database table using a view if the view refers to a single table. The view's maintenance status must be defined in the ABAP Dictionary as having "no restriction".

You can

[Change Single Lines \[Page 577\]](#)

[Change Several Lines \[Page 579\]](#)

[Changing Several Lines Using Internal Tables \[Page 580\]](#)

Changing a Single Line

Changing a Single Line

To change a single line with the UPDATE statement, you can use

- the short forms of the UPDATE statement
- the UPDATE statement with the SET clause and a fully specified WHERE condition (see [Changing Several Lines \[Page 579\]](#)).

The short forms of the UPDATE statement are as follows:

Syntax

UPDATE <dbtab> [CLIENT SPECIFIED] FROM <wa>.

and

UPDATE <dbtab> [CLIENT SPECIFIED].

In the first statement, the contents of the work area <wa> overwrite the line of the database table <dbtab> which has the same primary key as <wa>. You must declare the database table in the program using a TABLES statement.

In the second statement, you do not specify a work area <wa>. Instead, the contents of the table work area <dbtab> overwrite the line of the database table which has the same primary key.

The work area <wa> must be at least as long as the table work area of <dbtab>. To ensure that the work area has the same structure as the database table, you should define it with the DATA or TYPES statement using the LIKE <dbtab> option (see [The Basic Form of the DATA Statement \[Page 120\]](#)).

The CLIENT SPECIFIED option activates the automatic client handling. For more information, see [Specifying a Client for Processing Database Tables \[Page 596\]](#).

If the operation is concluded successfully, SY-SUBRC is set to 0 and SY-DBCNT is set to 1. Otherwise, SY-SUBRC returns 4 and SY-DBCNT returns 0.

If you want to specify the name of the database table at runtime, use the following syntax instead:

UPDATE (<dbtabname>) [CLIENT SPECIFIED] FROM <wa>.

You specify the database table (<dbtabname>) as described for the SELECT statement in [Specifying the Database Table to be Read \[Page 550\]](#).

If you specify the database table name at runtime, you must use the option FROM <wa>. It is not possible to use the table work area.

```
TABLES SPFLI.
DATA WA LIKE SPFLI.
MOVE 'AA'      TO WA-CARRID.
MOVE '0064'    TO WA-CONNID.
MOVE 'WASHINGTON' TO WA-CITYFROM.
.....
UPDATE SPFLI FROM WA.
MOVE 'LH'      TO SPFLI-CARRID.
MOVE '0017'    TO SPFLI-CONNID.
```

MOVE 'BERLIN' TO SPFLI-CITFROM.

.....

UPDATE SPFLI.

CARRID and CONNID are the primary key fields of table SPFLI. All fields of those lines where the primary key fields are "AA" and "0064", or "LH" and "0017", are replaced by the values in the corresponding fields of the work area WA or the table work area SPFLI.

Changing Several Lines

Changing Several Lines

To change multiple lines in the database table <dbtab> with the UPDATE statement, use the following syntax:

Syntax

```
UPDATE <dbtab> [CLIENT SPECIFIED] SET <S1>.. <Sn> [WHERE <condition>].
```

You use the WHERE clause (see [Choosing Lines for Selection \[Page 559\]](#)) to select the lines you want to change. If you do not specify a WHERE clause, **all** lines are changed. You must declare the database table with a TABLES statement in the program. The CLIENT SPECIFIED option activates the automatic client handling. For more information, see [Specifying a Client for Processing Database Tables \[Page 596\]](#).

You can identify the columns of the lines to be changed and assign values to each column using the list <S₁>.. <S_n> of the SET clause. The elements of this list can be different SET commands. There are three kinds of SET commands:

- <f> = <g>
The value in column <f> is set to the value <g> for all lines selected.
- <f> = <f> + <g>
The value in column <f> is incremented by the value of <g> for all lines selected.
- <f> = <f> - <g>
The value in column <f> is decremented by the value of <g> for all lines selected.

<g> can be either a variable or a literal. For information about convertibility between database fields and ABAP fields, see the keyword documentation for the INTO clause of the SELECT statement.

In SY-DBCNT you find the number of lines changed. SY-SUBRC returns 0 if at least one line was changed, and 4 if no line was changed.

If you change lines of database tables using the SET clause, you **cannot** specify the name of the database table at runtime - you must specify it in the program.

```
TABLES SFLIGHT.
```

```
UPDATE SFLIGHT SET PLANETYPE = 'A310'  
              FLPRICE = FLPRICE - '100.00'  
              WHERE CARRID = 'LH'.
```

For all lines of SFLIGHT where the CARRID field contains "LH", the PLANETYPE field is changed to "A310" and the FLPRICE field is decremented by 100.00. For more examples, refer to the keyword documentation for the UPDATE statement.

Changing Several Lines Using an Internal Table

To change several lines in a database table with the UPDATE statement using an internal table, use the following syntax:

Syntax

```
UPDATE <dbtab> [CLIENT SPECIFIED] FROM TABLE <itab>.
```

If you want to specify the name of the database table at runtime, use the following syntax instead:

```
UPDATE (<dbtabname>) [CLIENT SPECIFIED] FROM TABLE <itab>.
```

The lines of the internal table <itab> overwrite the lines of the database table with the same primary key.

The lines of the internal table must have at least the same length as the database table. To ensure that the internal table has the same structure as the database table, you should define it with the DATA or TYPES statement using the LIKE <dbtab> option (see [The Basic Form of the DATA Statement \[Page 120\]](#)).

If the system cannot change a line because no line with the specified key exists, it does not terminate the entire operation, but continues processing the next line of the internal table.

If all lines from the internal table have been processed, SY-SUBRC is set to 0. Otherwise, it is set to 4. In the latter case, you can calculate the number of lines the system did not process by subtracting the number of actually processed lines given in SY-DBCNT from the number of lines of the internal table. If the internal table is empty, SY-SUBRC and SY-DBCNT are set to 0.

Working on sets of lines is always preferable to working on single lines because it is considerably more efficient.

```
TABLES SPFLI.
```

```
DATA ITAB LIKE SPFLI OCCURS 10 WITH HEADER LINE.
```

```
ITAB-CARRID = 'UA'. ITAB-CONNID = '0011'. ITAB-CITYFROM =..
APPEND ITAB.
```

```
ITAB-CARRID = 'LH'. ITAB-CONNID = '1245'. ITAB-CITYFROM =..
APPEND ITAB.
```

```
ITAB-CARRID = 'AA'. ITAB-CONNID = '4574'. ITAB-CITYFROM =..
APPEND ITAB.
```

```
.....
```

```
UPDATE SPFLI FROM TABLE ITAB.
```

In this example, an internal table ITAB is given the same structure as the database table SPFLI. After filling ITAB, those lines of the database table are overwritten, which have the same contents in the primary key fields (CARRID and CONNID) as a line in the internal table.

Adding or changing lines

To insert a line into a database table, regardless of whether the primary key of this line already exists, you use the MODIFY statement.

There are two possibilities:

- If the database table contains no line with the same primary key as the line to be inserted, MODIFY works like INSERT, i.e. the line is added.
- If the database already contains a line with the same primary key as the line to be inserted, MODIFY works like UPDATE, i.e. the line is changed.

You can

[Insert a Single Line \[Page 582\]](#)

[Insert Several Lines \[Page 583\]](#)

For performance reasons, you should use MODIFY only if you cannot distinguish between these two options in your ABAP program.

Inserting a Single Line

To insert a single line, use the following syntax:

Syntax

MODIFY <dbtab> [CLIENT SPECIFIED] [FROM <wa>].

If you want to specify the name of the database table at runtime, use the following syntax instead:

MODIFY (<dbtabname>) [CLIENT SPECIFIED] [FROM <wa>].

This writes the contents of the work area <wa> to the database table. SY-SUBRC is set to 0 and SY-DBCNT is set to 1.

If the primary key specified in the work area <wa> does not exist in the database table, the above statements add <wa> to the database table, as described in [Adding Single Lines \[Page 572\]](#).

If the primary key of the work area <wa> already exists in the database table, the above statements change the appropriate line of the database table, as described in [Changing Single Lines \[Page 577\]](#).

Inserting Several Lines

To insert several lines, use the following syntax:

Syntax

MODIFY <dbtab> [CLIENT SPECIFIED] FROM TABLE <itab>.

If you want to specify the name of the database table at runtime, use the following syntax instead:

MODIFY (<dbtabname>) [CLIENT SPECIFIED] FROM TABLE <itab>.

The lines of the internal table <itab> overwrite the lines of the database table with the same primary key. The remaining lines are added to the database table. SY-SUBRC is set to 0 and SY-DBCNT is set to the number of lines in the internal table.

When they insert lines, the above statements work as described in [Adding Several Lines from an Internal Table \[Page 574\]](#)

When they overwrite lines, the above statements work as described in [Changing Several Lines from an Internal Table \[Page 580\]](#)

Working on sets of lines is always preferable to working on single lines because it is considerably more efficient.

Deleting Lines from Database Tables

To delete lines from a database table, you use the DELETE statement. The DELETE statement allows you to delete either several lines or a single line.

You can only delete lines of a database table using a view if the view refers to a single table. The view's maintenance status must be defined in the ABAP Dictionary as having "no restriction".

You can

[Delete Single Lines \[Page 585\]](#)

[Delete Several Lines \[Page 586\]](#)

[Delete Several Lines Using an Internal Table \[Page 587\]](#)

Deleting a Single Line

Deleting a Single Line

To delete a single line with the DELETE statement, you can either use

- the DELETE statement with a fully specified WHERE condition (see [Deleting Several Lines \[Page 586\]](#)).
- the short forms of the DELETE statement.

The short forms of the DELETE statement are as follows:

Syntax

DELETE <dbtab> [CLIENT SPECIFIED] FROM <wa>.

and

DELETE <dbtab> [CLIENT SPECIFIED].

In the first statement, the line of the database table <dbtab> with the same primary key as specified in <wa> is deleted. You must declare the database table in the program using a TABLES statement.

In the second statement, you do not specify a work area <wa>. Instead, the line of the database table with the same primary key as specified in the table work area <dbtab> is deleted.

The work area <wa> must have at least the same length as the primary key of the database table. The CLIENT SPECIFIED option activates the automatic client handling. For more information, see [Specifying a Client for Processing Database Tables \[Page 596\]](#).

SY-SUBRC is set to 0 if the line was successfully deleted and to 4 if there is no line with a primary key which matches the one specified.

If you want to specify the name of the database table at runtime, use the following syntax instead:

DELETE (<dbtabname>) [CLIENT SPECIFIED] FROM <wa>.

You specify the database table (<dbtabname>) as described for the SELECT statement in [Specifying the Database Table to be Read \[Page 550\]](#).

If you specify the database table name at runtime, you must use the option FROM <wa>. It is not possible to use the table work area.

```
TABLES SPFLI.
```

```
DATA WA LIKE SPFLI.
```

```
MOVE 'AA'      TO WA-CARRID.
```

```
MOVE '0064'    TO WA-CONNID.
```

```
DELETE SPFLI FROM WA.
```

```
MOVE 'LH'      TO SPFLI-CARRID.
```

```
MOVE '0017'    TO SPFLI-CONNID.
```

```
DELETE SPFLI.
```

CARRID and CONNID are the primary key fields of table SPFLI. All lines with the primary key fields "AA" and "0064" or "LH" and "0017" are deleted.

Deleting Several Lines

To delete multiple lines in the database table <dbtab> with the DELETE statement, use the following syntax:

Syntax

```
DELETE FROM <dbtab> [CLIENT SPECIFIED] WHERE <conditions>.
```

If you want to specify the name of the database table at runtime, use the following syntax instead:

```
DELETE FROM (<dbtabname>) [CLIENT SPECIFIED] WHERE <conditions>.
```

You must select the lines you want to delete using a WHERE condition (see [Selecting the Lines to be Read \[Page 559\]](#)).

All of the lines in the database that meet the condition are deleted.

Check the WHERE condition carefully, to ensure that you do not accidentally delete all of the entries in a table with the DELETE statement.

The CLIENT SPECIFIED option activates the automatic client handling. For more information, see [Specifying a Client for Processing Database Tables \[Page 596\]](#). For information about how to specify the name of the database table, see [Specifying the Database Table to be Read \[Page 550\]](#).

The number of lines deleted is counted in the system field SY-DBCNT. SY-SUBRC returns 0 if at least one line was deleted, and 4 if no line was changed.

```
TABLAS SFLIGHT.
```

```
DELETE FROM SFLIGHT WHERE PLANETYPE = 'A310'  
AND CARRID = 'LH'.
```

This example deletes all lines with plane type "A310" and carrier "LH" from SFLIGHT.

Deleting Several Lines Using an Internal Table

Deleting Several Lines Using an Internal Table

To delete several lines in a database table with the DELETE statement using an internal table, use the following syntax:

Syntax

```
DELETE <dbtab> [CLIENT SPECIFIED] FROM TABLE <itab>.
```

If you want to specify the name of the database table at runtime, use the following syntax instead:

```
DELETE (<dbtabname>) [CLIENT SPECIFIED] FROM TABLE <itab>.
```

These statements delete the lines from the database table whose primary key is specified in one of the lines in the internal table <itab>.

The lines of the internal table <itab> must have at least the same length as the work area of the database table.

If the system cannot delete a line because no line with the specified key exists, it does not terminate the entire operation, but continues processing the next line of the internal table.

If all lines from the internal table have been processed, SY-SUBRC is set to 0. Otherwise, it is set to 4. In the latter case, you can calculate the number of lines the system did not process by subtracting the number of actually processed lines given in SY-DBCNT from the number of lines of the internal table. If the internal table is empty, SY-SUBRC and SY-DBCNT are set to 0.

Working on sets of lines is always preferable to working on single lines because it is considerably more efficient.

```
TABES SPFLI.
```

```
DATA ITAB LIKE SPFLI OCCURS 10 WITH HEADER LINE.
```

```
ITAB-CARRID = 'UA'. ITAB-CONNID = '0011'.
```

```
APPEND ITAB.
```

```
ITAB-CARRID = 'LH'. ITAB-CONNID = '1245'.
```

```
APPEND ITAB.
```

```
ITAB-CARRID = 'AA'. ITAB-CONNID = '4574'.
```

```
APPEND ITAB.
```

```
.....
```

```
UPDATE SPFLI FROM TABLE ITAB.
```

In this example, an internal table ITAB is given the same structure as the database table SPFLI. After filling ITAB, those lines of the database table are overwritten, which have the same contents in the primary key fields (CARRID and CONNID) as a line in the internal table.

Reading Lines of Database Tables Using a Cursor

You can read the lines of a database table using a database cursor. This involves the following steps:

[Opening a Cursor \[Page 589\]](#)

[Using the Cursor to Read Data \[Page 590\]](#)

[Closing the Cursor \[Page 591\]](#)

For a comprehensive example of how to read data from a database table using a cursor, see

[Example of Reading Data Using a Cursor \[Page 592\]](#)

Opening a Cursor

Opening a Cursor

You can use a cursor to read the next line or set of lines from the result set of almost any SELECT statement. You do this by linking the cursor to the corresponding SELECT statement with the OPEN CURSOR statement as follows:

Syntax

```
OPEN CURSOR [WITH HOLD] <c> FOR SELECT.....  
           [WHERE <conditions>].
```

You must already have declared the cursor <c> with type CURSOR.

If you use the WITH HOLD option, the cursor remains open even if a Native SQL database commit occurs (see [Using Native SQL Statements in an ABAP Program \[Page 599\]](#)).

You **cannot** use SELECT statements with the following clauses:

- SELECT SINGLE.....
- SELECT.... <a₁> <a₂>..... if the list contains only aggregate expressions and no individual database fields.

You can use any other clause listed under [Defining the Result of a Selection \[Page 543\]](#).

Using a Cursor to Read Data

Once you have opened a cursor, you can use the FETCH statement to read the next line or set of lines from the set of results formed in the OPEN CURSOR statement.

Syntax

```
FETCH NEXT CURSOR <c> INTO <target>.
```

The selected lines are read into the target area specified in the INTO clause (see [Specifying a Target Area for Read Lines \[Page 553\]](#)).

If the FETCH statement cannot read a line, SY-SUBRC has the value 4, otherwise, its value is 0.

Closing the Cursor

Closing the Cursor

You should close all cursors that you no longer require using the CLOSE CURSOR statement as follows:

Syntax

CLOSE CURSOR <c>.

Cursors are automatically closed in the following situations:

- When a COMMIT WORK or ROLLBACK WORK statement occurs (see [Writing or Undoing Changes to Database Tables \[Page 594\]](#)).
- When a Native SQL database commit or rollback occurs (see [Using Native SQL Statements in an ABAP Program \[Page 599\]](#)).
- When a new screen is sent.
- When a Remote Function Call occurs.

If you use the WITH HOLD option in the OPEN CURSOR statement, the cursor is not closed if a Native SQL database commit occurs.

Example of Reading Data Using a Cursor

The following program shows how you can read data using a cursor:

```

TABLES SPFLI.
DATA: C1 TYPE CURSOR, C2 TYPE CURSOR.
DATA: WA1 LIKE SPFLI,
      WA2 LIKE SPFLI.
DATA: FLAG1, FLAG2.
OPEN CURSOR: C1 FOR SELECT * FROM SPFLI WHERE CARRID = 'LH',
             C2 FOR SELECT * FROM SPFLI WHERE CARRID = 'AA'.
DO.
  IF FLAG1 NE 'X'.
    FETCH NEXT CURSOR C1 INTO WA1.
    IF SY-SUBRC <> 0.
      CLOSE CURSOR C1. FLAG1 = 'X'.
    ELSE.
      WRITE: / WA1-CARRID, WA1-CITYFROM, WA1-CITYTO.
    ENDIF.
  ENDIF.
  IF FLAG2 NE 'X'.
    FETCH NEXT CURSOR C2 INTO WA2.
    IF SY-SUBRC <> 0.
      CLOSE CURSOR C2. FLAG2 = 'X'.
    ELSE.
      WRITE: / WA2-CARRID, WA2-CITYFROM, WA2-CITYTO.
    ENDIF.
  ENDIF.
  IF FLAG1 = 'X' AND FLAG2 = 'X'. EXIT. ENDIF.
ENDDO.

```

This produces the following output:

LH	FRANKFURT	NEW YORK
AA	NEW YORK	SAN FRANCISCO
LH	FRANKFURT	NEW YORK
AA	SAN FRANCISCO	NEW YORK
LH	FRANKFURT	SAN FRANCISCO
LH	SAN FRANCISCO	FRANKFURT
LH	FRANKFURT	BERLIN
LH	BERLIN	FRANKFURT
LH	BERLIN	FRANKFURT
LH	FRANKFURT	BERLIN
LH	FRANKFURT	BERLIN
LH	BERLIN	FRANKFURT
LH	ROM	FRANKFURT

The database table SPFLI is read using two cursors, each with different conditions.. Lines with CARRID "LH" or "AA" are processed alternately within the

Example of Reading Data Using a Cursor

DO loop. If no line is found that corresponds to one of the conditions, cursor C1 or C2 is closed, and FLAG1 or FLAG2 respectively is set to "X".

Writing or Undoing Changes to Database Tables

Sometimes, it is necessary to make sure that changes to database tables have been written permanently to the database before a program continues processing. On the other hand, you may need to undo changes to database tables before they are written permanently.

To write changes permanently to the database, use the COMMIT WORK statement. To undo changes before they are written to the database, use the ROLLBACK WORK statement.

These statements play an important role in transaction programming. For an overview of SAP transactions, database transactions, opening and closing screens, and so on, see [Programming Database Changes \[Page 667\]](#). This section explains how to use COMMIT WORK and ROLLBACK WORK in an ABAP program consisting only of a single dialog step.

Your ABAP program can contain various actions that make up a Logical Unit of Work (LUW). Normally, the actions in an LUW will either all be carried out, or none of them will be carried out. Let us assume that an LUW inserts five lines into a database table. For this transaction to be successful, all five lines must be saved in the database (update request, database transaction). Database transactions end automatically before a new screen is displayed (end of the ABAP program). You have no control over this in your program.

However, if you want to make sure that all of the changes you have made to date are written to the database immediately, you can conclude the LUW with the COMMIT WORK statement. COMMIT WORK marks the end of the LUW in your program and starts the update process. After a COMMIT WORK statement, you can no longer undo the changes in the database.

However, if an error occurs in an LUW, the part of it that has already been processed must be undone. In other words, none of the entries can be written to the database. The ROLLBACK WORK statement "cancels" all database changes in the current LUW.

To write changes permanently to the database, use the COMMIT WORK statement as follows:

Syntax

COMMIT WORK [AND WAIT]

If you use the AND WAIT option, the program waits until the system has finished updating the database. If the update is successful, SY-SUBRC is set to 0. If the update is not successful, SY-SUBRC is unequal to zero.

To undo changes before they are written to the database, use the ROLLBACK WORK statement as follows:

Syntax

ROLLBACK WORK.

If the changes are canceled successfully, SY-SUBRC is set to 0. If the changes cannot be canceled successfully, SY-SUBRC is unequal to zero.

All database cursors are lost when you use the COMMIT WORK and ROLLBACK WORK statements. For this reason, you should not use them within SELECT loops, or before you have executed all required SQL statements.

For further information about the COMMIT WORK and ROLLBACK WORK statements, see the keyword documentation in the ABAP Editor.

Writing or Undoing Changes to Database Tables

```
TABLES SPFLI.

DATA FLAG.

SPFLI-CARRID = 'UA'. SPFLI-CONNID = '0011'.
SPFLI-CITYFROM =.....
INSERT SPFLI.
IF SY-SUBRC <> 0.
    FLAG = 'X'.
ENDIF.

SPFLI-CARRID = 'LH'. SPFLI-CONNID = '1245'.
SPFLI-CITYFROM =.....
INSERT SPFLI.
IF SY-SUBRC <> 0.
    FLAG = 'X'.
ENDIF.

SPFLI-CARRID = 'AA'. SPFLI-CONNID = '4574'.
SPFLI-CITYFROM =.....
INSERT SPFLI.
IF SY-SUBRC <> 0.
    FLAG = 'X'.
ENDIF.

.....

.....

IF FLAG = 'X'.
    ROLLBACK WORK.
ELSE.
    COMMIT WORK.
ENDIF.
```

In this example, a set of lines is inserted into table SPFLI. This forms an LUW. After each INSERT statement, the program checks whether the operation was successful, or whether there is already a line in SPFLI with the same primary key (CARRID and CONNID). In the latter case, FLAG is set to "X". If SY-SUBRC is not 0 for each INSERT statement, all database changes following the last IF statement are canceled using ROLLBACK WORK. Otherwise, they are written to the database by the COMMIT WORK statement.

Specifying Clients for Processing Database Tables

Automatic client handling is the default in Open SQL. Statements that access **client-dependent tables** only use the data from the current client.

If this is the case, you cannot:

- Specify a condition for the client in the WHERE clause - the system returns a syntax error if you do.
- Fill the client field in the table work area for INSERT, UPDATE, or DELETE statements. The ABAP runtime system automatically fills the field with the current client before executing the Open SQL statements.

If you want to specify a client when you process a table, use the appropriate Open SQL statement (SELECT, INSERT, UPDATE, MODIFY, or DELETE) with the following addition:

Syntax

... CLIENT SPECIFIED....

The addition must always come directly after the name of the database table.

The CLIENT SPECIFIED option switches off automatic client handling. You can then specify the client in a WHERE condition, and fill the client field in table work areas.

```
TABLES SPFLI.  
SELECT * FROM SPFLI CLIENT SPECIFIED  
      WHERE MANDT BETWEEN '001' AND '003'.  
...  
ENDSELECT.
```

In this example, all lines of the database table SPFLI for clients "001" to "003" are read, regardless of the client in which the user is logged on.

Performance Notes

This section contains a few hints for improving the performance of Open SQL statements:

- **Keep the data selection small**

Select as little data as possible to avoid transporting unnecessary data across the network. Always use the WHERE clause in the corresponding Open SQL statement. Avoid selecting useless data that you filter out later (using CHECK, for example).

Use the indexes of the relevant database tables to make your WHERE clause more efficient, by checking all index fields for equality (EQ, =) and using the AND operator. The primary key of a database table is automatically its primary index. You can also create secondary indexes for a database table in the ABAP Dictionary. For further information, see the online documentation for the [ABAP Dictionary \[Ext.\]](#).

Avoid using complex WHERE clauses, since the system has to break them down into several individual statements for the database system.

If possible, avoid using the NOT operator in the WHERE clause, because it is not supported by database indexes; invert the logical expression instead.
- **Transport as little data as possible**

Transport only the fields of the database table that you really need. If you do not need all of the fields in a table, use a field list in the SELECT clause instead of SELECT *.

Use the aggregate functions in the SELECT clause for calculations, instead of transporting large amounts of data and then performing the equivalent calculation.

Use the UPDATE statement sparingly: Only update the columns that have actually changed, and do not overwrite the entire line.

Note here that the addition INTO CORRESPONDING FIELDS in the INTO clause of the SELECT statement is only effective for large amounts of data, because the time required to compare the field names is otherwise too great.

Consider using the DISTINCT option if you are expecting a lot of duplicate table entries.
- **Use fewer database accesses**

Use multiple, not single operations if you want to evaluate selected database entries more than once. Transfer all of the data at once from the database into internal tables.

Where possible, avoid accessing the same data more than once (for example, by using SELECT before an UPDATE or DELETE statement).

Avoid nested SELECT loops. Instead, use an internal table and a second SELECT statement with the FOR ALL ENTRIES addition.

In exceptional cases, you can also select data using a separate cursor.
- **Using Database Buffering**

Saving database tables in local buffers can save a considerable amount of time. Wherever possible, use buffered data, and only use the BYPASSING BUFFER addition where absolutely necessary.

Note that the following additions automatically bypass the buffer: DISTINCT, SINGLE FOR UPDATE, and aggregate functions in the SELECT clause.

You can check the performance of your SQL or ABAP functions by using the 'SQL Trace' and 'Runtime analysis' utilities. Choose *System → Utilities → SQL Trace* or *System → Utilities → Runtime Analysis*. For more information about these utilities, refer to the [ABAP Workbench Tools \[Ext.\]](#) documentation.

For an example of how to measure the runtime of a SELECT statement using the GET RUN TIME FIELD statement, see [Measuring the Runtime of Database Accesses \[Page 503\]](#).

For tips on how to improve the performance of ABAP tasks, choose transaction SE30 or *Test → Runtime analysis* from the *ABAP Development Workbench* and click on *Tips & Tricks* there. The relevant examples are under *SQL Interface*.

Using Native SQL Statements in an ABAP Program

Using Native SQL Statements in an ABAP Program

Open SQL allows you to access database tables declared in the ABAP Dictionary regardless of the database platform that your R/3 System is using. However, you may occasionally need to use database-specific SQL statements (Native SQL) in your programs.

As a rule, an ABAP program containing database-specific SQL statements will **not** run under different database systems. If your program will be used on more than one database platform, only use Open SQL statements.

To use a Native SQL statement, you must precede it with the EXEC SQL statement, and follow it with the ENDEXEC statement as follows:

Syntax

```
EXEC SQL [PERFORMING <form>].
```

```
    <Native SQL statement> [;]
```

```
ENDEXEC.
```

The semicolon (;) after the Native SQL statement is optional. However, you may not use a period at this point. Furthermore, using inverted commas (") in a native SQL statement does not introduce a comment as it would in normal ABAP syntax.

When you use Native SQL, the tables that you address do not have to be declared in the ABAP Dictionary. Accordingly, the ABAP program does not have to have a TABLES statement for them. Note also that you do not need an explicit CONNECT for the database - this is done automatically when the R/3 System is started. You need to know whether table and field names are case-sensitive in your chosen database.

Native SQL has no automatic client handling (see [Specifying Clients for Processing Database Tables \[Page 596\]](#)). You must always specify the relevant client.

Data is transported between the database table and the ABAP program using host variables. These are declared in the ABAP program, and preceded in the Native SQL statement by a colon (:). You can use both elementary and structured fields as host variables.

If the result of a SELECT statement is a table, use the PERFORMING <form> option to read it line by line in a closed loop. The subroutine <form> is called once for each line read. You can use it to process the data that has just been read. For example, you can append it to an internal table.

For further information about Native SQL, see the keyword documentation for the EXEC SQL statement.

```
DATA: BEGIN OF WA,  
      CLIENT(3),  
      ARG1(3),  
      ARG2(3),  
      END OF WA.  
DATA F3 VALUE ' 1 '.
```

Using Native SQL Statements in an ABAP Program

```
EXEC SQL PERFORMING LOOP_OUTPUT.  
SELECT CLIENT, ARG1 INTO :WA FROM TABLE_001 WHERE ARG2 = :F3  
ENDEXEC.
```

```
FORM LOOP_OUTPUT.  
  WRITE: / WA-CLIENT, WA-ARG2.  
ENDFORM.
```

This example uses the work area WA and the text field F3 as host variables in the Native SQL SELECT statement. WA is the target area into which the selected data is written. F3 is used in the WHERE condition. The subroutine LOOP_OUTPUT writes the data from WA to the screen.

Native SQL for Informix

The principal new features in the Native SQL interface in Release 4.0 are:

- Ability to connect to several databases in parallel (including non-SAP databases)
- Use of the cursor for stored procedures
- Access to non-SAP tables
- Ability to use (almost) all Native SQL statements for Informix The Native SQL statements that are not supported are listed below.

Cursor Processing

You cannot use the normal PREPARE, DECLARE, OPEN, FETCH, CLOSE logic to work with cursors. Instead, use the syntax described in OSS note number 44977. The following cursor types are not supported at present:

- INSERT CURSOR
- SELECT FOR UPDATE or WHERE CURRENT OF

Transactions

Native SQL COMMIT or ROLLBACK statements are not passed to the database by the Native SQL interface. Instead, they are converted into Open SQL COMMIT statements. This is to ensure that transactions behave consistently in conjunction with the DBSL. Since a COMMIT also triggers a BEGIN WORK in the DBSL, the Native SQL interface cannot process the COMMIT statement.

Isolation Levels

All SET ISOLATION LEVEL statements last not only for the Native SQL context, but also in subsequent statements, for example, in Open SQL for the same database connection. This can adversely affect the lock mechanism. To prevent this, you should reset the isolation level to DIRTY READ.

Additionally, do not use the above statements in stored procedures, since they cannot be compared with the DBSL.

Multi Connect

You can only connect to the database using TCP/IP, not using shared memory.

Data Types

When you use Native SQL and host variables to access SAP tables, you should use work areas or the LIKE statement.

When accessing non-SAP tables, you must ensure that the ABAP variable type and the database field type are compatible, since the Native SQL interface works without information from the ABAP Dictionary. The Native SQL interface opens the type and memory area reserved in ABAP to the DBMS. This allows you to read and write directly to and from it. This means that the interface behaves, with a few exceptions, exactly as though you were using ESQL/C. In other words, some conversions are not allowed and trigger error messages, and rounding errors and

truncation are also possible. The principal data type descriptions and conversions for Informix databases are described in the **Programmers Manual**.

For example, if you use Native SQL to attempt to convert a DATE value into an ABAP variable with type I, the system will be unable to perform the conversion.

Below is a description of the various type compatibilities between ABAP variables and Informix database field types.

It lists all of the type conversions that are permitted and supported by the Native SQL interface. Preferred type combinations are shown in bold type.

For illegal type combinations, the error procedure of the ABAP Workbench is given, along with the SQL error code of any resulting ABAP short dump.

There is no formal description of how other type combinations not listed here behave.

The first section deals with using Native SQL (INSERT and UPDATE) to save ABAP variables in non-SAP database tables. The second section deals with how to read from external database tables into ABAP variables using Native SQL (SELECT).

The following abbreviations are used:

- OK
OK means that the interface supports the conversion. However, data can still be lost without the database returning an error. For example, the right-hand end of a string may be truncated, if the target variable in the ABAP program is not long enough. This is indicated in the tables by T.
- NO
'NO' in the tables below indicates that the conversion is either refused by the system, or that it fails. The SQL code that appears in an ABAP short dump is also given. However, truncations, rounding, and undefined values are also possible.
- K
No error message
- T
Truncation
- U
Undefined value
- Rounding

The description of the SQL data type DECIMAL applies also to the types MONEY and NUMERIC.

INSERT and UPDATE

Saving values from ABAP variables using Native SQL.

Each of the following tables represents a single **ABAP data type**. This is the type of the ABAP variable whose value you want to save.

In the left-hand column is the **SQL data type** of the database field of the external database table.

On the right is the reaction of the ABAP Workbench.

ABAP Data Type C

Native SQL for Informix

The ABAP data type C can be reproduced in most databases. However, where there are non-CHAR fields, you must take care with length, format, and permitted value ranges, since rounding, truncation, or format errors can easily occur. Truncation and rounding errors are not returned as SQL errors.

SQL data type	Test	Result	Code
char	ABAP data value width > SQL field width	KT	
	ABAP data value width <= SQL field width	OK	
varchar	ABAP data value width > SQL field width	KT	
	ABAP data value width <= SQL field width	OK	
nchar	ABAP data value width > SQL field width	KT	
	ABAP data value width <= SQL field width	OK	
text	Occurs in a DELETE statement, or text field occurs in the WHERE clause	NO	-615
	otherwise	OK	
byte	NO	-608	
date	ABAP data value does not have a valid SQL date format	NO	KU
	ABAP date value has a valid SQL date format, but wrong value range	NO	-608
	ABAP data value has a valid SQL date format	OK	
datetime	ABAP data value does not have a valid SQL datetime format	NO	-1262
	ABAP date value has a valid SQL date format, but wrong value range	NO	-1218
	ABAP data value has a valid SQL time format	OK	
interval	ABAP data value does not have a valid SQL interval format	NO	-1264
	ABAP data value does not have a valid SQL interval format	NO	-1263
	ABAP data value has a valid SQL interval format	OK	
decimal	ABAP data value is non-numeric	NO	-1213
	Loss of significant figures in conversion	NO	-1226
	Loss of non-significant figures in conversion	NO	-1226
	No loss in conversion	OK	
integer	ABAP data value is non-numeric	NO	-1213
	Loss of significant figures in conversion	NO	-1215
	No loss in conversion	OK	
smallint	ABAP data value is non-numeric	NO	-1213
	Loss of significant figures in conversion	NO	-1215

	No loss in conversion	OK	
float	ABAP data value is non-numeric	NO	-1213
	Loss of significant figures in conversion	NO	KO
	Loss of non-significant figures in conversion	NO	KO
	No loss in conversion	OK	
smallfloat	ABAP data value is non-numeric	NO	-1213
	Loss of significant figures in conversion	NO	KO
	Loss of non-significant figures in conversion	NO	KO
	No loss in conversion	OK	
double precision	ABAP data value is non-numeric	NO	-1213
	Loss of significant figures in conversion	NO	KO
	Loss of non-significant figures in conversion	NO	KO
	No loss in conversion	OK	

Informix 7.3 does not support the DESCRIBE command for DELETE statements or input variables in WHERE clauses. For this reason, the ABAP EXEC SQL interface does not map to the field type SQLTEXT.

ABAP Data Type N

SQL data type	Test	Result	Code
char	ABAP data value width > SQL field width	KT	
	ABAP data value width <= SQL field width		OK
varchar	ABAP data value width > SQL field width	KT	
	ABAP data value width <= SQL field width		OK
nchar	ABAP data value width > SQL field width	KT	
	ABAP data value width <= SQL field width		OK
text		NO	-608
byte		NO	-608
date		NO	-1218
datetime	SQL field type consists of a single element (for example, datetime hour to hour)		
interval	and ABAP value range is valid	OK	
	otherwise	NO	-1218,-1261
decimal	Loss of figures in conversion	NO	-1226
	No loss in conversion	OK	
integer	Loss of figures in conversion	NO	-1215

Native SQL for Informix

	No loss in conversion	OK	
smallint	Loss of figures in conversion	NO	-1215
	No loss in conversion	OK	
float	Loss of figures in conversion	KO	
	No loss in conversion	OK	
smallfloat	Loss of figures in conversion	KO	
	No loss in conversion	OK	
double	Loss of figures in conversion	KO	
precision	No loss in conversion	OK	

ABAP Data Type P

SQL data type	Test	Result	Code
char	ABAP data value width > N	KT	
	ABAP data value width <= N	OK	
varchar	ABAP data value width > N	KT	
	ABAP data value width <= N	OK	
nchar	ABAP data value width > N	KT	
	ABAP data value width <= N	OK	
text		NO	-608
byte		NO	-608
date		NO	-1218
datetime interval	SQL field type consists of a single element (for example, datetime hour to hour) and ABAP value range is valid		
	otherwise	NO	.-1218,-1261
decimal	Loss of figures in conversion	NO	-1226
	No loss in conversion	OK	
integer	Loss of figures in conversion	NO	-1215
	No loss in conversion	OK	
smallint	Loss of figures in conversion	NO	-1215
	No loss in conversion	OK	
float	Loss of figures in conversion	KO	
	No loss in conversion	OK	
smallfloat	Loss of figures in conversion	KO	
	No loss in conversion	OK	

double	Loss of figures in conversion	KO
precision	No loss in conversion	OK

The memory format of ABAP data type P is different to the Informix type DECIMAL (see the Informix Guide to SQL Reference and the ABAP User's Guide). For INSERT with CHAR fields, you need to set the following field width N: $N = (\text{packed length} * 2) + 1$ (for the decimal point)

ABAP Data Type I

SQL data type	Test	Result	Code
char	Loss of figures in conversion	NO	-1207
	No loss in conversion	OK	
varchar	Loss of figures in conversion	NO	-1207
	No loss in conversion	OK	
nchar	Loss of figures in conversion	NO	-1207
	No loss in conversion	OK	
text	NO	-608	
byte	NO	-608	
date	NO	-1218	
datetime interval	NO	-1260	
decimal	Loss of figures in conversion	NO	-1226
	No loss in conversion	OK	
integer	OK		
smallint	Loss of figures in conversion	NO	-1214
	No loss in conversion	OK	
float	Loss of figures in conversion	KO	
	No loss in conversion	OK	
smallfloat	Loss of figures in conversion	KO	
	No loss in conversion	OK	
double precision	Loss of figures in conversion	KO	
	No loss in conversion	OK	

ABAP Data Type F

SQL data type	Test	Result	Code
char	Loss of figures in conversion	NO	-1207
	No loss in conversion	OK	

Native SQL for Informix

varchar	Loss of figures in conversion	NO	-1207
	No loss in conversion	OK	
nchar	Loss of figures in conversion	NO	-1207
	No loss in conversion	OK	
text		NO	-608
byte		NO	-608
date		NO	-1218
datetime interval		NO	-1260
decimal	Loss of figures in conversion	NO	-1226
	No loss in conversion	OK	
integer	Loss of figures in conversion	NO	-1215
	No loss in conversion	OK	
smallint	Loss of figures in conversion	NO	-1214
	No loss in conversion	OK	
float		OK	
smallfloat	Loss of figures in conversion	KO	
	No loss in conversion	OK	
double precision		OK	

ABAP Data Type D

SQL data type	Test	Result	Code
char (8)	ABAP data value width = 8 > SQL field width		KT
	ABAP data value width = 8 <= SQL field width		OK
varchar	ABAP data value width = 8 > SQL field width		KT
	ABAP data value width = 8 <= SQL field width		OK
nchar	ABAP data value width = 8 > SQL field width		KT
	ABAP data value width = 8 <= SQL field width		OK
date		NO	-1205
other types	See ABAP data type C		

You cannot map the ABAP data type D to the Informix data type date because they have different formats. Format conversion is not possible because of the different display variants.

ABAP Data Type T

SQL data type	Test	Result	Code
char (6)	ABAP data value width = 6 > SQL field width		KT
	ABAP data value width = 6 <= SQL field width		OK
varchar	ABAP data value width = 6 > SQL field width		KT
	ABAP data value width = 6 <= SQL field width		OK
nchar	ABAP data value width = 6 > SQL field width		KT
	ABAP data value width = 6 <= SQL field width		OK
datetime hour to second	NO	-1261	
other types	See ABAP data type C		

You cannot map the ABAP data type T to the Informix data type datetime because they have different formats. There is no reformatting.

ABAP Data Type X

SQL data type	Test	Result	Code
char	ABAP data value width >= 256		-609
	ABAP data value width < 256		OK
varchar	ABAP data value width >= 256		-609
	ABAP data value width < 256		OK
nchar	ABAP data value width >= 256		-609
	ABAP data value width < 256		OK
text	ABAP data value width >= 256		OK
byte	ABAP data value width > 256		OK
date	Not defined		
datetime	Not defined		
interval	Not defined		
decimal	Not defined		
smallint	Not defined		
integer	Not defined		
float	Not defined		
smallfloat	Not defined		
double precision	Not defined		

Native SQL for Informix

Since the DBSL for R/3 tables in Informix allows you to store hexadecimal fields shorter than 256 characters in CHAR format, the EXEC SQL interface allows you to access these fields. If the ABAP variable is shorter than 256 characters, the database field is interpreted as CHAR. The first two characters in the database field contain the length information.

This also means that the ABAP variable must be larger than 256 characters when you access a byte or text field.

SELECT

Reading database fields using Native SQL.

Each of the following tables refers to a **database field type**, which is also the type of the value you want to read from the database table. In some cases, several SQL data types behave in the same way. Where this occurs, the data types have been included in a single table to save space.

In the left-hand column is the **ABAP data type** of the target variable in the ABAP program.

On the right is the reaction of the ABAP Workbench.

Database Column Type char, varchar, or nchar

ABAP data type	Test	Result	Notes	Code
Character C	SQL column width > ABAP data value width	KT		
	SQL column width < ABAP data field width with spaces at the right-hand end)	OK	(left-justified, filled	
	SQL field width = ABAP field width	OK		
Numeric N	SQL column width > ABAP data value width	KT		
	SQL field width < ABAP data value width	OK	(right-justified, leading	
	SQL field width = ABAP field width	OK	zeros on the left)	
Packed P	SQL field width > N	KT		
	SQL field width <= N	OK		
Integer I	SQL field value is non-numeric	NO	-1213	
	Loss of significant figures in conversion	NO	-1215	
	No loss in conversion	OK		
Float F	SQL field value is non-numeric	NO	-1213	
	Loss of significant figures in conversion	KO		
	Loss of non-significant figures in conversion	KO		
	No loss in conversion	OK		
Date D	SQL field width > 8 = ABAP data value width	KT		
	SQL field width < 8 = ABAP data value width with spaces at the right-hand end)	OK	(left-justified, filled	
	SQL field width = 8 = ABAP data value width	OK		
Time T	SQL field width > 6 = ABAP data value width	KT		
	SQL field width < 6 = ABAP data value width	OK	(left-justified, filled	

	with spaces at the right-hand end) SQL field width = 6 = ABAP data value width OK
Hexadecimal X	ABAP data value width < 256 < SQL field width KT ABAP data value width >= 256 NO . -1269 ABAP data value width < 256 and SQL field width <= 256 OK left-justified. First and second characters are lost

For SELECT with CHAR fields of length N for ABAP variables with type P, you calculate the required packed length as follows:

Packed length = ceil (N / 2)

Database Field Type text

ABAP data type	Test	Result	Notes	Code
Character C	SQL column width > ABAP data value width	OK	(String truncated at right-hand end)	
	SQL column width < ABAP data field width	OK	(left-justified, filled with spaces at the right-hand end)	
	SQL field width = ABAP field width	OK		
Numeric N	NO	. -1269		
Packed P	NO	. -1269		
Integer I	NO	. -1269		
Float F	NO	. -1269		
Date D	NO	. -1269		
Time T	NO	. -1269		
Hexadecimal X	ABAP data value width >= 256	NO	-1269	
	ABAP data value width >= 256	OK		

Database Field Type byte

ABAP data type	Test	Result	Notes	Code
Character C	ABAP data value width < 256	NO	-1269	
	ABAP data value width >= 256			
	SQL column width > ABAP data value width	KT		
	SQL column width < ABAP data field width	OK	(left-justified, filled with spaces at the right-hand end)	
	SQL field width = ABAP field width	OK		
Numeric N	NO	. -1269		

Native SQL for Informix

Packed P	NO	.-1269
Integer I	NO	.-1269
Float F	NO	.-1269
Date D	NO	.-1269
Time T	NO	.-1269
Hexadecimal X	ABAP data value width >= 256	NO -1269
	ABAP data value width >= 256	OK

Database Field Type date

ABAP data type	Test	Result	Notes	Code
Character C	10 > ABAP data value width			KT
	10 < ABAP data value width the right-hand end)	OK	(left-justified, filled with spaces at	
	10 = ABAP data value width	OK		
Numeric N	Not defined			
Packed P	Not defined			
Integer I	Not defined			
Float F	Not defined			
Date D	Not defined			
Time T	Not defined			
Hexadecimal X		NO	1269	

Database Field Type datetime or interval

ABAP data type	Test	Result	Notes	Code
Character C	SQL field width > ABAP data value width			KT
	SQL field width <= ABAP data value width	OK	Not always left-justified	
Numeric N	Not defined			
Packed P	Not defined			
Integer I	NO	-1260		
Float F	NO	-1260		
Date D	Not defined			
Time T	Not defined			
Hexadecimal X	NO	-1269		

Database Field Type decimal, numeric, or money

ABAP data type	Test	Result	Notes	Code
Character C	Loss of figures in conversion			
	No loss in conversion OK			
Numeric N	Loss of figures in conversion KT			
	No loss in conversion OK			
Packed P	Loss of figures in conversion KT			
	No loss in conversion OK			
Integer I	Loss of figures in conversion NO -1215			
	No loss in conversion OK			
Float F	Loss of figures in conversion KO			
	No loss in conversion OK			
Date D	Not defined			
Time T	Not defined			
Hexadecimal X	Not defined			

When you read the data type DECIMAL into an ABAP field with type C, you must allow space for thousand separators as well as for the decimal point.

Database Field Type integer or smallint

ABAP data type	Test	Result	Notes	Code
Character C	Loss of significant figures in conversion NO . KU			
	No loss in conversion OK			
Numeric N	Loss of significant figures in conversion KT			
	No loss in conversion OK			
Packed P	Loss of significant figures in conversion KT			
	No loss in conversion OK			
Integer I	OK			
Float F	OK			
Date D	Not defined			
Time T	Not defined			
Hexadecimal X	Not defined			

Database Field Type smallfloat, float, or double precision

ABAP data type	Test	Result	Notes	Code
----------------	------	--------	-------	------

Native SQL for Informix

Character C	Loss of significant figures in conversion	K
	No loss in conversion	OK
Numeric N	Loss of significant figures in conversion	KT
	No loss in conversion	OK
Packed P	Loss of significant figures in conversion	KT
	No loss in conversion	OK
Integer I	Loss of significant figures in conversion	NO -1215
	No loss in conversion	OK
Float F	OK	
Date D	Not defined	
Time T	Not defined	
Hexadecimal X	Not defined	

List of non-executable Commands

You cannot use the following Native SQL statements in ABAP. They come under the following categories:

- NA
not applicable. These statements are non-executable and generate an error.
- SP
Special syntax. These statements require a special Native SQL syntax; see the keyword documentation for EXEC SQL
- NR
Not recommended. These statements should not be used with an R/3 database that is managed using the ABAP Dictionary.

This list is not intended to be exhaustive. It is merely intended as a programming guideline. Any Informix SQL statements that are not listed here should be executable without any problem.

Informix SQL statement	Class
ALLOCATE DESCRIPTOR	NA
ALTER FRAGMENT	NR
ALTER INDEX	NR
ALTER TABLE ... MODIFY NEXT SIZE	NR
ALTER TABLE ... LOCK MODE	NR
ALTER TABLE ... ADD ROWIDS	NR
ALTER TABLE ... DROP ROWIDS	NR

BEGIN WORK	NA
CHECK TABLE	NR
CLOSE DATABASE	NR
CONNECT	SP
CREATE AUDIT	NR
CREATE DATABASE	NR
DATABASE	NR
DEALLOCATE DESCRIPTOR	NR
DECLARE CURSOR	SP
DESCRIBE	NA
DISCONNECT	SP
DROP AUDIT	NR
DROP DATABASE	NR
EXECUTE	NA
EXECUTE IMMEDIATE	NA
EXECUTE PROCEDURE	SP
FETCH	SP
FLUSH	NA
FREE	NA
GET DESCRIPTOR	NA
GET DIAGNOSTICS	NA
GRANT FRAGMENT	NR
INFO	NA
LOAD	NA
OPEN CURSOR	SP
OUTPUT	NA
PREPARE	NA
PUT	NA
RECOVER TABLE	NR
RENAME DATABASE	NR
REVOKE FRAGMENT	NR
ROLLFORWARD DATABASE	NR
SELECT INTO TEMP <tab>	NA
SET CONNECTION	SP

Native SQL for Informix

SET DATASKIP	NR
SET DEBUG FILE	NR
SET DESCRIPTOR	NR
SET TRANSACTION	NR
START DATABASE	NR
UNLOAD	NA
UPDATE STATISTICS	NR
WHENEVER	NA

Native SQL for DB2 Common Server

The ABAP Native SQL interface allows you to use tables in ABAP that are not administered in the ABAP Dictionary. In a program, you may use Native SQL statements between the EXEC SQL and ENDEXEC statements. From Release 4.0, you can use any Native SQL statements permitted by the database itself between the EXEC SQL and ENDEXEC statements.

This allows you to integrate data created outside the R/3 System into the ABAP Workbench. The Native SQL interface allows you to use ABAP variables within the Native SQL statements, either to store values of ABAP variables in the non-SAP database or to process data from a non-SAP database in an ABAP program.

Since the Native SQL interface works without reference to the ABAP Dictionary, it must ensure itself that the data types from both sides (the internal ABAP data type and the field type from the database) are compatible. It therefore attempts to convert the data types. If it cannot convert them appropriately, an error is triggered, and a message is sent to the user.

For example, if you use Native SQL to attempt to convert a database field with type TIME into an ABAP variable with type I, the system will be unable to perform the conversion.

If there were no type check, the date field would be partially transferred into the ABAP variable, and subsequently interpreted as an integer. The worst-case scenario is that the system would complete a calculation using these incorrect values. It would then be a question of good fortune as to whether you could still identify from the result that something was wrong.

Most of the necessary conversion routines are already included in the CLI interface of the relevant database manufacturer. The DBDS interface for DB2-CS porting also incorporates a set of compatibility checks. There are also conversion routines for cases where conversion is theoretically possible but not (or not yet) supported by the database interface. These conversion routines greatly reduce the probability of an error occurring in the ABAP application.

The permitted and supported type conversions between ABAP variables and field types for the DB2 Common Server are listed below.

Preferred type combinations are shown in bold type.

For illegal type combinations, the error procedure of the ABAP Workbench is given, along with the SQL error code of any resulting ABAP short dump.

There is no formal description of how other type combinations not listed here behave.

The first section deals with using Native SQL (INSERT and UPDATE) to save ABAP variables in non-SAP database tables. The second section deals with how to read from external database tables into ABAP variables using Native SQL (SELECT).

Insert and Update

Saving values from ABAP variables using Native SQL.

Each of the tables listed below refers to a single ABAP data type.

In the left-hand column is the SQL data type of the database field of the external database table.

On the right is the reaction of the ABAP runtime system.

This reaction depends both on the compatibility of the ABAP data type and the variable value and on the field type in the database. OK means that the interface supports the conversion. NO indicates that the conversion is not possible. In the latter case, the SQL code that appears in an ABAP short dump is also given.

Native SQL for DB2 Common Server

ABAP Data Type C

SQL data type	Test	Result	Code
char	ABAP data value width > SQL field width	NO	22001
	ABAP data value width <= SQL field width	OK	
varchar	ABAP data value width > SQL field width	NO	22001
	ABAP field width <= SQL field width	OK	
decimal	ABAP data value is non-numeric	NO	22005
	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion	NO	22001
	No loss in conversion	OK	
numeric	ABAP data value is non-numeric	NO	22005
	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion	NO	22001
	No loss in conversion	OK	
smallint	ABAP data value is non-numeric	NO	22005
	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion	NO	22001
	No loss in conversion	OK	
integer	ABAP data value is non-numeric	NO	22005
	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion	NO	22001
	No loss in conversion	OK	
float	ABAP data value is non-numeric	NO	22005
	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion	NO	22001
	No loss in conversion	OK	
double	ABAP data value is non-numeric	NO	22005
	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion	NO	22001
	No loss in conversion	OK	
date	ABAP data value does not have a valid SQL date format	NO	22007
	ABAP data value has a valid SQL date format	OK	
time	ABAP data value does not have a valid SQL time format	NO	22007
	ABAP data value has a valid SQL time format	OK	
timestamp	ABAP data value does not have a valid SQL timestamp format	NO	

Native SQL for DB2 Common Server

	22007	
	ABAP data value has a valid SQL timestamp format	OK

ABAP Data Type N

SQL data type	Test	Result	Code
char	ABAP data value width > SQL field width	NO	22001
	ABAP data value width <= SQL field width		OK
varchar	ABAP data value width > SQL field width	NO	22001
	ABAP data value width <= SQL field width		OK
decimal	Loss of figures in conversion	NO	22003
	No loss in conversion		OK
numeric	Loss of figures in conversion	NO	22003
	No loss in conversion		OK
smallint	Loss of figures in conversion	NO	22003
	No loss in conversion		OK
integer	Loss of figures in conversion	NO	22003
	No loss in conversion		OK
float	Loss of figures in conversion	NO	22003
	No loss in conversion		OK
double	Loss of figures in conversion	NO	22003
	No loss in conversion		OK
date		NO	22007
time		NO	22007
timestamp		NO	22007

ABAP Data Type P

SQL data type	Test	Result	Code
char	Not defined		
varchar	Not defined		
decimal	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion	NO	22001
	No loss in conversion		OK
numeric	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion	NO	22001

Native SQL for DB2 Common Server

	No loss in conversion OK
smallint	Loss of significant figures in conversion NO 22003
	Loss of non-significant figures in conversion NO 22001
	No loss in conversion OK
integer	Loss of significant figures in conversion NO 22003
	Loss of non-significant figures in conversion NO 22001
	No loss in conversion OK
float	Loss of significant figures in conversion NO 22003
	Loss of non-significant figures in conversion NO 22001
	No loss in conversion OK
double	Loss of significant figures in conversion NO 22003
	Loss of non-significant figures in conversion NO 22001
	No loss in conversion OK
date	Not defined
time	Not defined
timestamp	Not defined

ABAP Data Type I

SQL data type	Test	Result	Code
char	Loss of figures in conversion	NO	22003
	No loss in conversion	OK	
varchar	Loss of figures in conversion	NO	22003
	No loss in conversion	OK	
decimal	Loss of figures in conversion	NO	22003
	No loss in conversion	OK	
numeric	Loss of figures in conversion	NO	22003
	No loss in conversion	OK	
smallint	Loss of figures in conversion	NO	22003
	No loss in conversion	OK	
integer	Loss of figures in conversion	NO	22003
	No loss in conversion	OK	
float	Loss of figures in conversion	NO	22003
	No loss in conversion	OK	
double	Loss of figures in conversion	NO	22003

	No loss in conversion	OK
date	NO	22007
time	NO	22007
timestamp	NO	22007

ABAP Data Type F

SQL data type	Test	Result	Code
char	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion	NO	22001(?)
	No loss in conversion	OK	
varchar	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion	NO	22001(?)
	No loss in conversion	OK	
decimal	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion	NO	22001
	No loss in conversion	OK	
numeric	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion	NO	22001
	No loss in conversion	OK	
smallint	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion	NO	22001
	No loss in conversion	OK	
integer	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion	NO	22001
	No loss in conversion	OK	
float	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion	NO	22001
	No loss in conversion	OK	
double	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion	NO	22001
	No loss in conversion	OK	
date	NO	22007	
time	NO	22007	
timestamp	NO	22007	

Native SQL for DB2 Common Server

ABAP Data Type D

SQL data type	Test	Result	Code
char (8)	ABAP data value width = 8 > SQL field width	NO	22001
	ABAP data value width = 8 <= SQL field width	OK	
varchar	ABAP data value width = 8 > SQL field width	NO	22001
	ABAP data value width = 8 <= SQL field width	OK	
smallint	NO	22003	
integer	OK		
float	OK		
double	OK		
decimal	Loss of figures in conversion	NO	22003
	No loss in conversion	OK	
numeric	Loss of figures in conversion	NO	22003
	No loss in conversion	OK	
date	NO	22007	
time	NO	22007	
timestamp	NO	22007	

ABAP Data Type T

SQL data type	Test	Result	Code
char (6)	ABAP data value width = 6 > SQL field width	NO	22001
	ABAP data value width = 6 <= SQL field width	OK	
varchar	ABAP data value width = 6 > SQL field width	NO	22001
	ABAP data value width = 6 <= SQL field width	OK	
smallint	NO	22003	
integer	OK		
float	OK		
double	OK		
decimal	Loss of figures in conversion	NO	22003
	No loss in conversion	OK	
numeric	Loss of figures in conversion	NO	22003
	No loss in conversion	OK	
date	NO	22007	

Native SQL for DB2 Common Server

time	NO	22007
timestamp	NO	22007

ABAP Data Type X

SQL data type	Test Result Code
char	ABAP data value width > SQL field width NO 22001
	ABAP data value width = SQL field width OK
	ABAP data value width < SQL field width Not defined
varchar	ABAP data value width > SQL field width NO 22001
	ABAP data value width = SQL field width OK
	ABAP data value width < SQL field width Not defined
decimal	Not defined
numeric	Not defined
smallint	Not defined
integer	Not defined
float	Not defined
double	Not defined
date	Not defined
time	Not defined
timestamp	Not defined

Select

Reading database fields using Native SQL.

Each of the following tables refers to a database field type, which is also the type of the value you want to read from the database table. In some cases, several SQL data types behave in the same way. Where this occurs, the data types have been included in a single table to save space.

In the left-hand column is the ABAP data type of the target variable in the ABAP program.

On the right is the reaction of the ABAP Workbench.

This reaction depends on the compatibility of the database field type and value on the one hand, and on the ABAP data type on the other hand. OK means that the interface supports the conversion. However, data can still be lost without the database returning an error. For example, the right-hand end of a string may be truncated, if the target variable in the ABAP program is not long enough. The reason for this is that, even though the data is truncated when it is read, the original data remains undamaged in the database. Reactions of this nature are noted in the table.

Truncation is not tolerated when you insert values into the database. The original data is kept in a temporary program variable, and when you try to insert the

Native SQL for DB2 Common Server

values in the database, an SQL error is generated if the column width is too small.
The operation then terminates in a short dump (see above tables).

'NO' in the tables below indicates that the conversion is either refused by the system, or that it fails. The SQL code that appears in an ABAP short dump is also given.

22sim stands for simulated SQL errors.

Database Column Type char or varchar

ABAP data type	Test	Result	Code
Character C	SQL column width > ABAP data value width (String truncated at right-hand end)	OK	
	SQL column width < ABAP data value width (left-justified, filled with spaces at the right-hand end)	OK	
	SQL field width = ABAP data value width	OK	
Numeric N	SQL column width > ABAP data value width	NO	22sim
	SQL field width < ABAP data value width (right-justified, leading zeros on the left)	OK	
	SQL field width = ABAP field width	OK	
Packed P		NO	22sim
Integer I	SQL field value is non-numeric	NO	22005
	Loss of significant figures in conversion	NO	22003
	No loss in conversion	OK	
Float F	SQL field value is non-numeric	NO	22005
	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion (non-significant figures are lost)	OK	
	No loss in conversion	OK	
Date D	SQL field width > 8 = ABAP data value width (String truncated at right-hand end)	OK	
	SQL field width < 8 = ABAP data value width (left-justified, filled with spaces at the right-hand end)	OK	
	SQL field width = 8 = ABAP data value width	OK	
Time T	SQL field width > 6 = ABAP data value width (String truncated at right-hand end)	OK	
	SQL field width < 6 = ABAP data value width (left-justified, filled with spaces at the right-hand end)	OK	
	SQL field width = 6 = ABAP data value width	OK	
Hexadecimal X	SQL column width > ABAP data value width (String truncated at right-hand end)	OK	
	SQL field width < ABAP data value width (left-justified, remaining characters are undefined)	OK	

	SQL field width = ABAP data value width OK
--	--

Database Field Type decimal or numeric

ABAP data type	Test	Result	Code
Character C	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion (non-significant figures are lost)		OK
	No loss in conversion	OK	
Numeric N	NO	22sim	
Packed P	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion (non-significant figures are lost)		OK
	No loss in conversion	OK	
Integer I	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion (non-significant figures are lost)		OK
	No loss in conversion	OK	
Float F	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion (non-significant figures are lost)		OK
	No loss in conversion	OK	
Date D	NO	22sim	
Time T	NO	22sim	
Hexadecimal X	Not defined		

Database Field Type float or double

ABAP data type	Test	Result	Code
Character C	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion (non-significant figures are lost)		OK
	No loss in conversion	OK	
Numeric N	NO	22sim	
Packed P	NO	22sim	
Integer I	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion (non-significant figures are lost)		OK

Native SQL for DB2 Common Server

	No loss in conversion	OK	
Float F	Loss of significant figures in conversion	NO	22003
	Loss of non-significant figures in conversion (non-significant figures are lost)	OK	
	No loss in conversion	OK	
Date D	NO	22sim	
Time T	NO	22sim	
Hexadecimal X	Not defined		

Database Column Type integer or smallint

ABAP data type	Test	Result	Code
Character C	Loss of significant figures in conversion	NO	22003
	No loss in conversion	OK	
Numeric N	NO	22sim	
Packed P	NO	22sim	
Integer I	Loss of significant figures in conversion	NO	22003
	No loss in conversion	OK	
Float F	Loss of significant figures in conversion	NO	22003
	No loss in conversion	OK	
Date D	NO	22sim	
Time T	NO	22sim	
Hexadecimal X	Not defined		

Database Field Type date

ABAP data type	Test	Result	Code
Character C	10 > ABAP data value width	NO	22003
	10 < ABAP data value width (left-justified, filled with spaces at the right-hand end)	OK	
	10 = ABAP data value width	OK	
Numeric N	NO	22sim	
Packed P	NO	22sim	
Integer I	NO	07006 (restricted data type attribute violation)	
Float F	NO	07006 (restricted data type attribute violation)	
Date D	NO	22sim	
Time T	NO	22sim	
Hexadecimal X	NO	07006 (restricted data type attribute violation)	

Database Field Type: time

ABAP data type	Test	Result	Code
Character C	8 > ABAP data value width	NO	22003

	8 < ABAP data value width OK (left-justified, filled with spaces at the right-hand end)
	8 = ABAP data value width OK
Numeric N	NO 22sim
Packed P	NO 22sim
Integer I	NO 07006 (restricted data type attribute violation)
Float F	NO 07006 (restricted data type attribute violation)
Date D	NO 22sim
Time T	NO 22sim
Hexadecimal X	NO 07006 (restricted data type attribute violation)

Database Field Type: timestamp

ABAP data type	Test Result Code
Character C	18 > ABAP data value width NO 22003 18 < ABAP data value width OK (left-justified, filled with spaces at the right-hand end) 18 = ABAP data value width OK
Numeric N	NO 22sim
Packed P	NO 22sim
Integer I	NO 07006 (restricted data type attribute violation)
Float F	NO 07006 (restricted data type attribute violation)
Date D	NO 22sim
Time T	NO 22sim
Hexadecimal X	NO 07006 (restricted data type attribute violation)

Locking Database Objects During Program Execution

In the R/3 System, application programs must lock the objects that they use. This is because each program in the system actually consists of two tasks:

- Dialog task, in which users enter and change data
- Update task, in which changes are made in the database.

The dialog task passes the data entered by the user to the update task, which reads it and changes the database accordingly.

Normally, the dialog task and update task work asynchronously. This means that the dialog task does not wait for the update task to finish updating the database before it returns to the user dialog.

This asynchronous process means that you need to lock the corresponding objects beyond the end of the user dialog, until the database has been successfully updated.

Your ABAP program must lock all of the objects with which it needs to work.

For example, you would need to lock a customer's data to change parts of it. No other user would then be able to access that data until the lock is released. Other users must wait until your program, including the update task, is finished.

To lock data objects, use the SAP locking mechanism (ENQUEUE/DEQUEUE).

Each application contains a number of function modules that you can use to lock objects.

For an overview of the function modules used for locking and unlocking objects choose *Development* → *Function Builder* from the initial screen of the ABAP Workbench. In the *Function Module* field, enter *queue*. This starts a generic search for all locking (ENQUEUE) and unlocking (DEQUEUE) function modules.

Alternatively, you can search directly for the object that you want to lock. Enter the name in the *Function module* field.

For further information, see the [System Services \[Ext.\]](#) documentation.

Checking the Authorization of Program Users

Checking the Authorization of Program Users

Unlike logical databases, SQL statements do not trigger any authorization checks in ABAP programs when you use them to read data (see [Advantages of Logical Databases \[Page 1206\]](#)). This can cause problems, since Open SQL and Native SQL statements allow unrestricted access to all database tables.

Not all users should have authorization to access all data that is available by using the SQL statements in the program. However, once you have released a program, any user with authorization for it can run it. This means that the programmer is responsible for checking that the user is authorized to access the data that the program processes.

To check the authorization of the user of an ABAP program, use the `AUTHORITY-CHECK` statement:

Syntax

```
AUTHORITY-CHECK OBJECT '<object>'
      ID '<name1>' FIELD <f1>
      ID '<name2>' FIELD <f2>
      .....
      ID '<name10>' FIELD <f10>.
```

<object> is the name of the object that you want to check. You must list the names (<name1>, <name2>...) of **all** authorization fields that occur in <object>. You can enter the values <f₁>, <f₂>.... for which the authorization is to be checked either as variables or as literals. The `AUTHORITY-CHECK` statement checks the user's profile for the listed object, to see whether the user has authorization for **all** values of <f>. Then, and only then, is `SY-SURC` set to 0. You can avoid checking a field by replacing `FIELD <f>` with `DUMMY`. You can only evaluate the result of the authorization check by checking the contents of `SY-SUBRC`. For a list of the possible return values and further information, see the keyword documentation for the `AUTHORITY-CHECK` statement. For further general information about the SAP authorization concept, refer to [Users and Authorizations \[Ext.\]](#).

There is an authorization object called `F_SPFLI`. It contains the fields `ACTVT`, `NAME`, and `CITY`.

```
SELECT * FROM SPFLI.
  AUTHORITY-CHECK OBJECT 'F_SPFLI'
    ID 'ACTVT' FIELD '02'
    ID 'NAME' FIELD SPFLI-CARRID
    ID 'CITY' DUMMY.
  IF SY-SUBRC NE 0. EXIT. ENDIF.
ENDSELECT.
```

If the user has the following authorizations for `F_SPFLI`:

`ACTVT 01-03`, `NAME AA-LH`, `CITY none`,

and the value of `SPFLI-CARRID` is not between "AA" and "LH", the authorization check terminates the `SELECT` loop.

Using Contexts

Contexts are objects within the ABAP Workbench that enable you to store details about the relationships between data. You use them in your ABAP programs to derive data which is dependent on a small number of key fields.

Contexts allow you to

- store processing logic from application programs in context programs, reducing the complexity of the application.
- make better use of recurring logic.
- use buffering to improve system performance.

[What are Contexts? \[Page 631\]](#)

[The Context Builder in the ABAP Workbench \[Page 632\]](#)

[Using Contexts in ABAP Programs \[Page 654\]](#)

[Working With Contexts - Hints \[Page 666\]](#)

What are Contexts?

What are Contexts?

Within the large quantities of data in the R/3 System database, there are always smaller sets of basic data that you can use to derive further information. In a relational database model, for example, these are the key fields of database tables.

When an application program requires further information in order to continue, this is often found in these small amounts of basic data. The relational links in the database are often used to read further data on the basis of this basic data, or further values are calculated from it using ABAP statements.

It is often the case that certain relationships between data are always used in the same form to get further data, either within a single program or in a whole range of programs. This means that a particular set of database accesses or calculations is repeatedly executed, despite the fact that the result already exists in the system. Contexts provide a way of avoiding this unnecessary system load.

Contexts are objects within the ABAP Workbench, consisting of key input fields, the relationships between these fields, and other fields which can be derived from them. Contexts can link these derived fields by foreign key relationships between tables, by function modules, or by other contexts. You define contexts abstractly in the ABAP Workbench (Transaction SE33), and use instances of them as runtime objects in your ABAP programs.

In the ABAP Workbench, contexts consist of fields and modules. Their fields are divided into key fields and derived fields. The modules describe the relationship between the fields.

Technically, contexts are special ABAP programs. You can save them

- in a non-executable program, with the name `CONTEXT_S_<context name>`. For the example context `DEMO_TRAVEL`, this would be `CONTEXT_S_DEMO_TRAVEL`.
- in a generated executable program, with the name `CONTEXT_X_<context name>`. For the example context `DEMO_TRAVEL`, this would be `CONTEXT_X_DEMO_TRAVEL`.

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

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

Whenever, in a program, new key fields are entered into an instance that has already been previously supplied with them, the system invalidates all affected values that have already been derived. These values are only re-supplied the next time they are queried.

The Context Builder in the ABAP Workbench

Normally you will use existing contexts within your R/3 System, and only use the Context Builder to familiarize yourself with them and test them (see [Finding and Displaying a Context \[Page 655\]](#)). However, if no suitable context already exists in the system, you can use the Context Builder to create your own.

This section describes how to use the Context Builder (Transaction SE33) in the ABAP Workbench.

[Creating and Editing a Context \[Page 633\]](#)

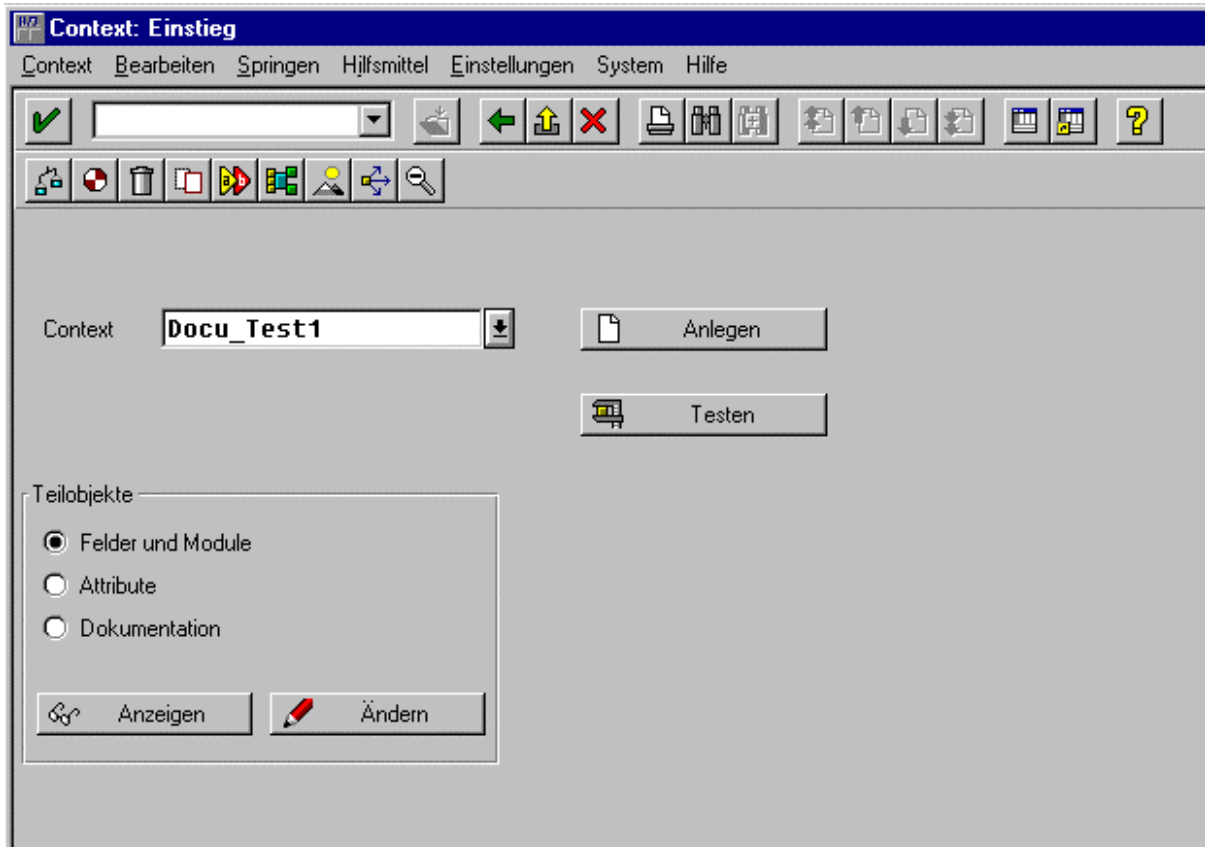
[Testing a Context \[Page 644\]](#)

[Buffering Contexts \[Page 646\]](#)

Creating and Editing a Context

Creating and Editing a Context

To create a context, call the Context Builder (Transaction SE33) and enter a name for your new context in the *Context* field. In the *Sub-objects* group box, select *Fields and modules*, then choose *Create*.


















On the next screen, you enter the attributes for your context:

Attribute von Context DOCU_TEST1 anlegen

Context Bearbeiten Springen System Hilfe

☒

Erstellt

Letzte Änderung

Titel

Originalsprache

Attribute

Status


Anwendung Basis (System)

Entwicklungsklasse

When you have saved the attributes, choose *Fields and modules*. The context maintenance screen is then displayed.

The screenshot shows the 'Modulverwaltung' (Module Management) window in SAP. The window is divided into three main sections: 'Felder' (Fields) on the left, 'Module' in the top right, and 'Parameter' in the bottom right. Each section contains a table with columns for Name, Typ, and Text. The 'Felder' table is empty. The 'Module' table has one row with 'Name' and 'Typ' columns. The 'Parameter' table has one row with 'Parameter', 'VC', 'Feldname', and 'Text' columns. The window has a standard SAP menu bar and toolbar at the top.

Creating and Editing a Context

There are three tables on this screen, in which you enter the [Fields \[Page 649\]](#), [Modules \[Page 651\]](#) and [Interfaces \[Page 653\]](#) in your context. You can enlarge each of these tables by choosing *Enlarge* .

The contents of the individual tables are interdependent, and are filled automatically to a certain extent.

The next three sections use simple examples to demonstrate how you can create contexts using different modules.

[Using Tables as Modules \[Page 636\]](#)

[Using Function Modules as Modules \[Page 639\]](#)

[Using Contexts as Modules \[Page 642\]](#)

Using Tables as Modules

The following example demonstrates how to create a context called DOCU_TEST1, which has two modules with type table. The tables used are SPFLI and SFLIGHT from the data model BC_TRAVEL.

1. Enter SPFLI in the *Name* or *Table/Module* column of the [Modules \[Page 651\]](#) table and choose *Enter*. The system then fills all of the tables of the context maintenance screen with unique entries as follows:

Felder	
Name	Typ
CARRID	SPFLI-CARRID
CONNID	SPFLI-CONNID
CITYFROM	SPFLI-CITYFROM
AIRPFROM	SPFLI-AIRPFROM
CITYTO	SPFLI-CITYTO
AIRPTO	SPFLI-AIRPTO
FLTIME	SPFLI-FLTIME
DEPTIME	SPFLI-DEPTIME
ARTIME	SPFLI-ARTIME
DISTANCE	SPFLI-DISTANCE
DISTID	SPFLI-DISTID
FLTYPE	SPFLI-FLTYPE

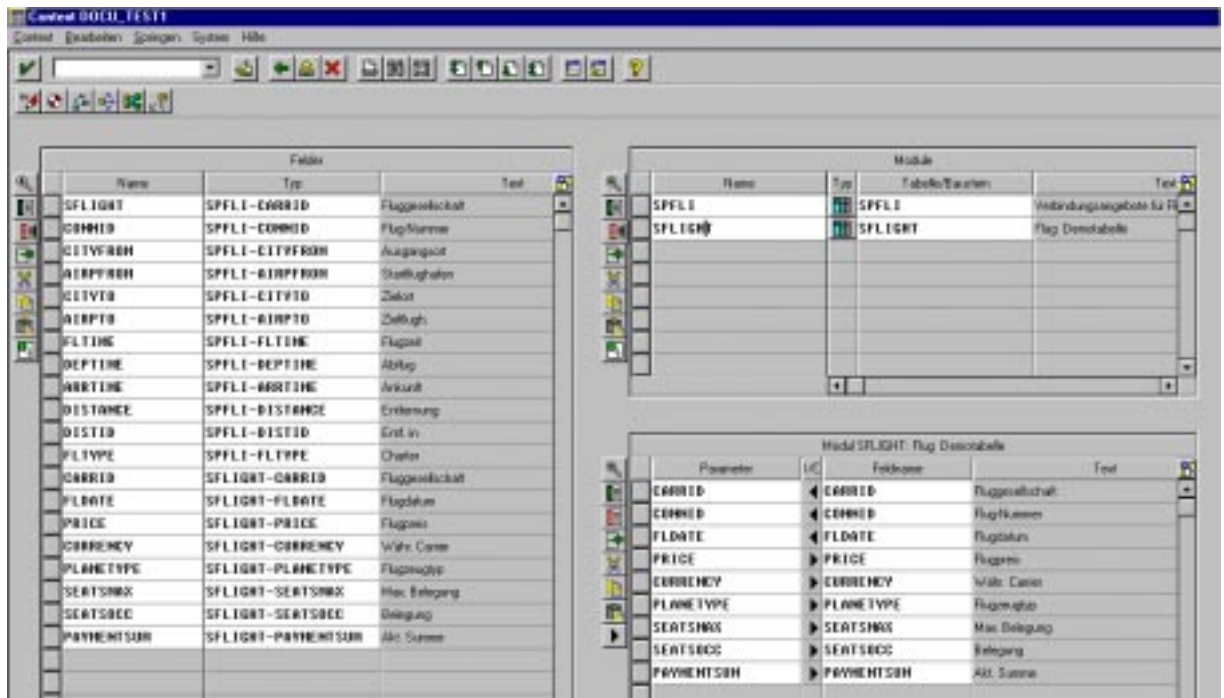
Module			
Name	Typ	Tabelle/Struktur	Test
SPFLI		SPFLI	Verbindungsangebote für Flüge

Modul SPFLI: Verbindungsangebote für Flüge			
Parameter	I/C	Feldname	Test
CARRID	◀	CARRID	Fluggesellschaft
CONNID	◀	CONNID	Flug-Nummer
CITYFROM	▶	CITYFROM	Ausgangsart
AIRPFROM	▶	AIRPFROM	Startflughafen
CITYTO	▶	CITYTO	Zielort
AIRPTO	▶	AIRPTO	Zielflugh.
FLTIME	▶	FLTIME	Flugzeit
DEPTIME	▶	DEPTIME	Abflug
ARTIME	▶	ARTIME	Ankunft
DISTANCE	▶	DISTANCE	Entfernung
DISTID	▶	DISTID	Entf. in
FLTYPE	▶	FLTYPE	Charakter

All of the columns of table SPFLI are used as fields in the context. The interface of module SPFLI shows that the key fields CARRID and CONNID are the input parameters.

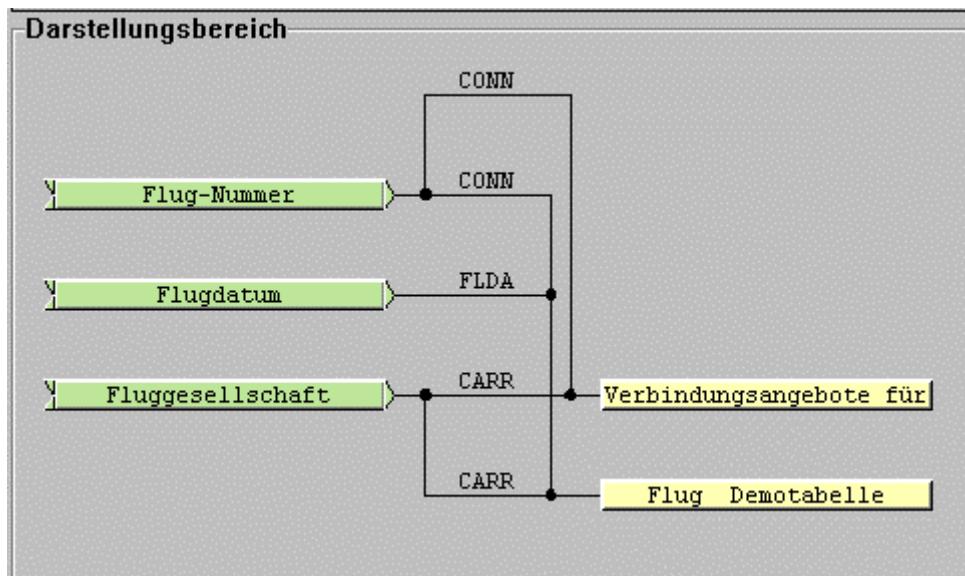
2. Now enter SFLIGHT in the *Name* or *Table/Module* column of the [Modules \[Page 651\]](#) table and choose *Enter*. The system now makes the following additional entries on the screen:

Using Tables as Modules



All of the columns in SFLIGHT which are not already contained in SPFLI are now also used in the context. The interface for module SFLIGHT shows that the key fields CARRID, CONNID and FLDATE are the input parameters of the new module.

3. Choose **Save**. Save the context.
4. Choose **Check**. The system checks the context for errors.
5. Choose **Network graphic**. The system displays the following graphic showing how values are derived within the context.



6. Generate the context. The system checks and saves your context before generating it.

The context is finished and can be tested (see [Testing a Context \[Page 644\]](#)) and used in programs (see [Using Contexts in ABAP Programs \[Page 654\]](#)).

Using Function Modules as Modules

Using Function Modules as Modules

The following example demonstrates how to create a context called DOCU_TEST2 which uses a function module as a module.

Function module

The function module is called SUBTRACTION, and is part of the function group TESTCONTEXT. It has the following interface:

Import-Parameter	Bezugsfeld/-struktur	Bezugstyp
V1	SFLIGHT-SEATSMAX	
V2	SFLIGHT-SEATSOCC	
◀		
Export-Parameter	Bezugsfeld/-struktur	
RESULT	SFLIGHT-SEATSOCC	

and the following source code:

```

1  function subtraction.
2  *"--
3  *"--Lokale Schnittstelle:
4  *"--      IMPORTING
5  *"--          VALUE(V1) LIKE  SFLIGHT-SEATSMAX
6  *"--          VALUE(V2) LIKE  SFLIGHT-SEATSOCC
7  *"--      EXPORTING
8  *"--          VALUE(RESULT) LIKE  SFLIGHT-SEATSOCC
9  *"--
10
11
12  result = v1 - v2.
13
14
15  endfunction.

```

The function module subtracts import parameter V2 from the other import parameter V1, and returns the value in RESULT.

To create the context:

Using Function Modules as Modules

1. Enter SUBTRACTION in the *Name* or *Table/Module* column of the [Modules \[Page 651\]](#) table and choose *Enter*. The system fills the tables on the context maintenance screen as follows:

The screenshot shows the SAP Context Maintenance screen (DD03L_TEST2) for the context 'SEATSOCC'. The screen is divided into three main sections: 'Fields', 'Module', and 'Parameters'.

Fields Table:

Name	Type	Text
SEATSMAX	SFLCNT-SEATSMAX	Max. Belegung
SEATSOCC	SFLCNT-SEATSOCC	Belegung
SEATSOCC_1	SFLCNT-SEATSOCC	Belegung 1

Module Table:

Name	Type	Table/Module	Text
SUBTRACTION	SUBTRACTION	Test Function Module by C	

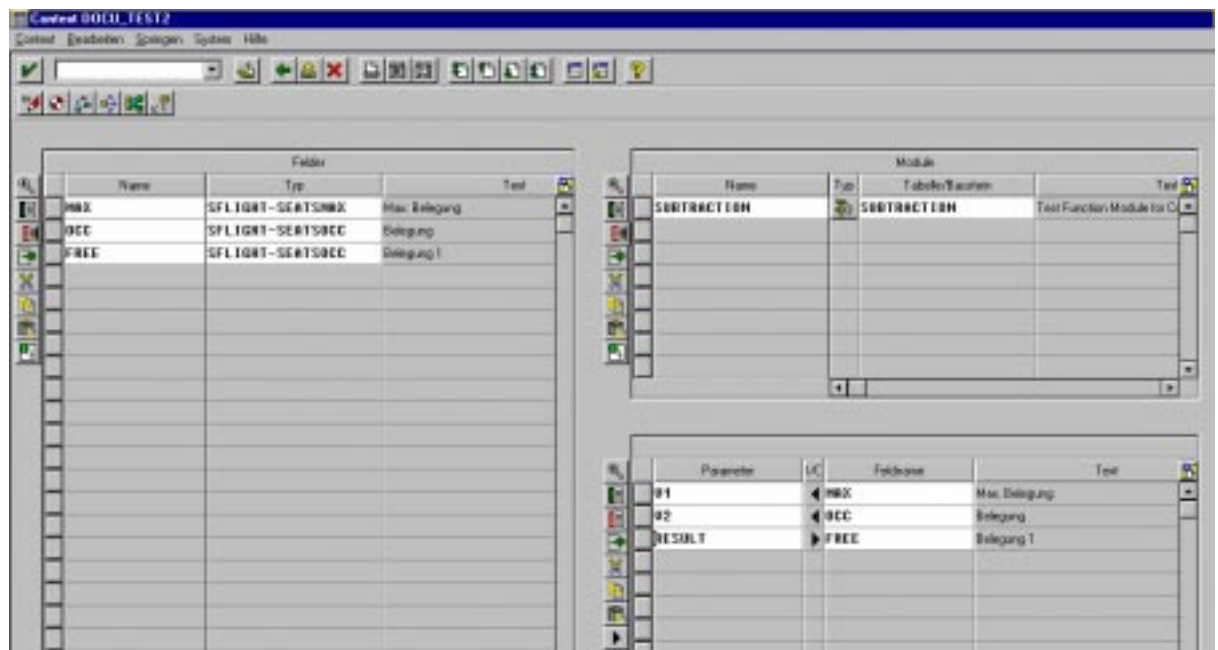
Parameters Table:

Parameter	UT	Feldname	Text
V1	←	SEATSMAX	Max. Belegung
V2	←	SEATSOCC	Belegung
RESULT	→	SEATSOCC_1	Belegung 1

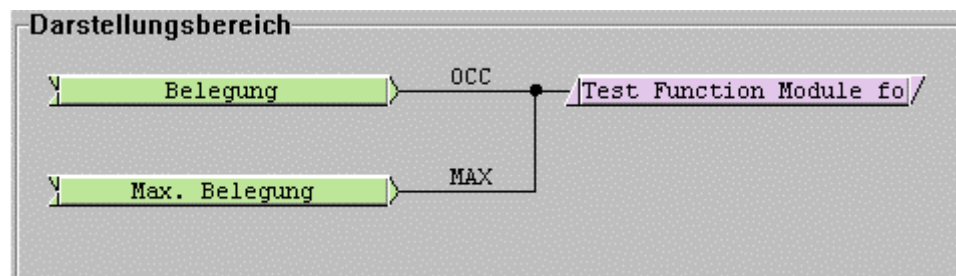
All of the interface parameters for the function module are used as fields in the context. The system generates the names of the context fields. The interface of the SUBTRACTION module shows that the import parameters V1 and V2 are assigned to the key fields SEATSMAX and SEATSOCC, and that the export parameter RESULT is assigned to the dependent field SEATSOCC_1.

2. You do not have to adopt these generated names. You can, instead, overwrite the context field names in the [Fields \[Page 649\]](#) and [Modules \[Page 651\]](#) table. For example, as follows:

Using Function Modules as Modules



3. Choose **Save**. Save the context.
4. Choose **Check**. The system checks the context for errors.
5. Choose **Network graphic**. The system displays the following graphic showing how values are derived within the context.



6. Generate the context. The system checks and saves your context before generating it.

The context is finished and can be tested (see [Testing a Context \[Page 644\]](#)) and used in programs (see [Using Contexts in ABAP Programs \[Page 654\]](#)).

Using Contexts as Modules

The following example demonstrates how to create a context called DOCU_TEST3, which uses two other contexts, DOCU_TEST1 and DOCU_TEST2, as modules. DOCU_TEST1 and DOCU_TEST2 have already been used in the preceding examples [Using Tables as Modules \[Page 636\]](#) and [Using Function Modules as Modules \[Page 639\]](#).

1. Enter DOCU_TEST1 and DOCU_TEST2 in the *Name* or *Table/Module* column of the [Modules \[Page 651\]](#) table and choose *Enter*. The system fills the tables on the context maintenance screen as follows:

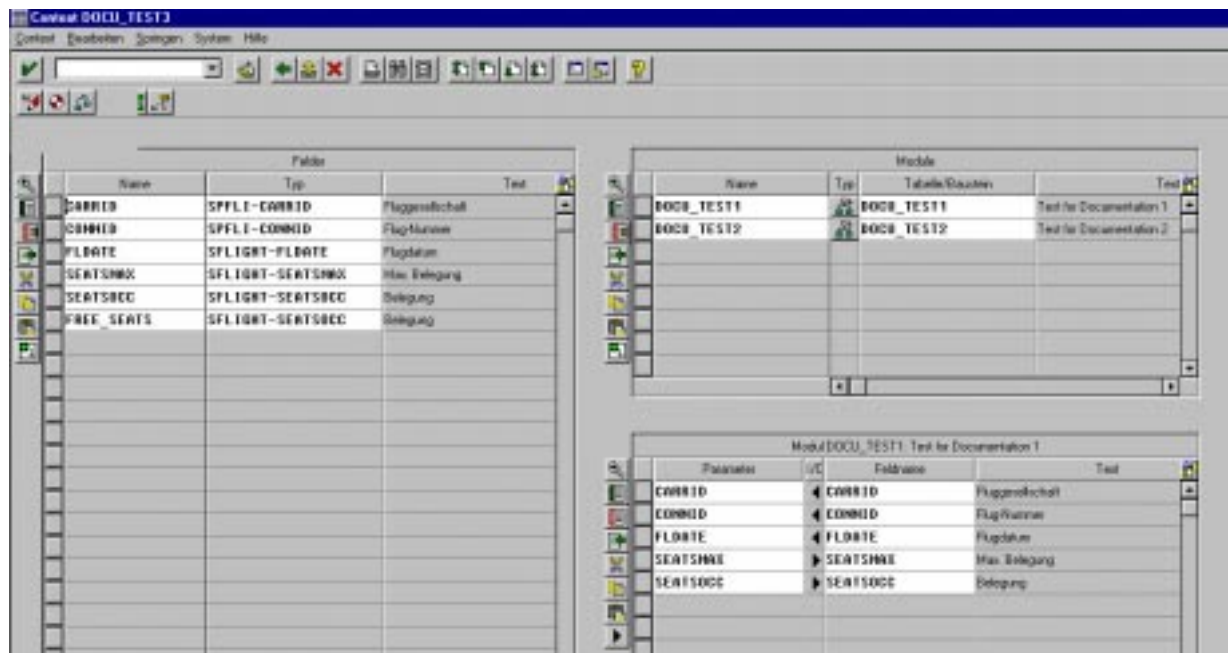
The screenshot displays the SAP Context Maintenance interface for Context DOCU_TEST3. It features three main tables:

- Fields Table:** Lists fields inherited from parent contexts. Columns include Name, Typ, and Test. Fields include CARRID, CONNID, CITYFROM, AIRPORT, CITYTO, AIRPTO, FLTIME, DEPTIME, WRTIME, DISTANCE, DESTID, FLTYPE, FLDATE, PRICE, CURRENCY, PLANE TYPE, SEATSMAX, SEATSECC, PAYMENTSUM, SEATSMAX_1, SEATSECC_1, and SEATSECC_2.
- Modules Table:** Lists modules used in the context. Columns include Name, Typ, Tabelle/Modulname, and Test. Modules listed are DOCU_TEST1 and DOCU_TEST2.
- Model DOCU_TEST2: Test for Documentation 2 Table:** Lists parameters and their field names. Columns include Parameter, UC, Feldname, and Test. Parameters listed are MAX, MCC, and FREE, with corresponding field names like SEATSMAX_1, SEATSECC_1, and SEATSECC_2.

All fields from the two contexts are used as fields in this context. The system generates the names of the context fields.

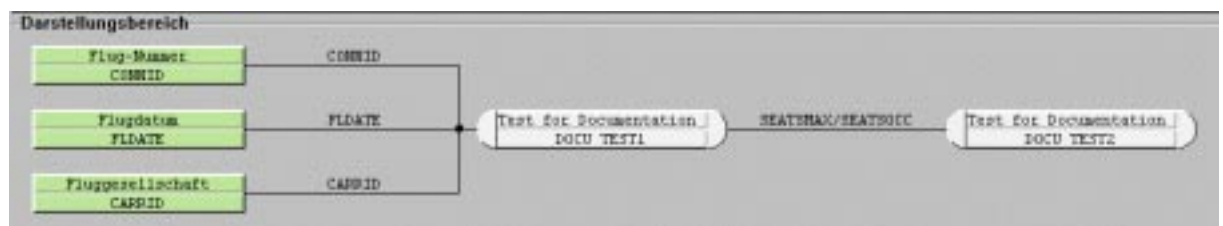
2. You do not have to use all of the context fields. You can also change the context field names in the [Fields \[Page 649\]](#) and [Modules \[Page 651\]](#) tables:

Using Contexts as Modules



In this example, we have kept the three key fields CARRID, CONNID and FLDATE, and deleted all dependent fields apart from SEATSMAX, SEATSOCC and FREE_SEATS. The output fields from module DOCU_TEST1, SEATSMAX and SEATSOCC are used as input fields for module DOCU_TEST2. The dependent field FREE_SEATS is filled from the output field FREE in DOCU_TEST2.

3. Save the context.
4. Choose *Check*. The system checks the context for errors.
5. Choose *Network graphic*. The system displays the following graphic showing how values are derived within the context.



6. Generate the context. The system checks and saves your context before generating it.

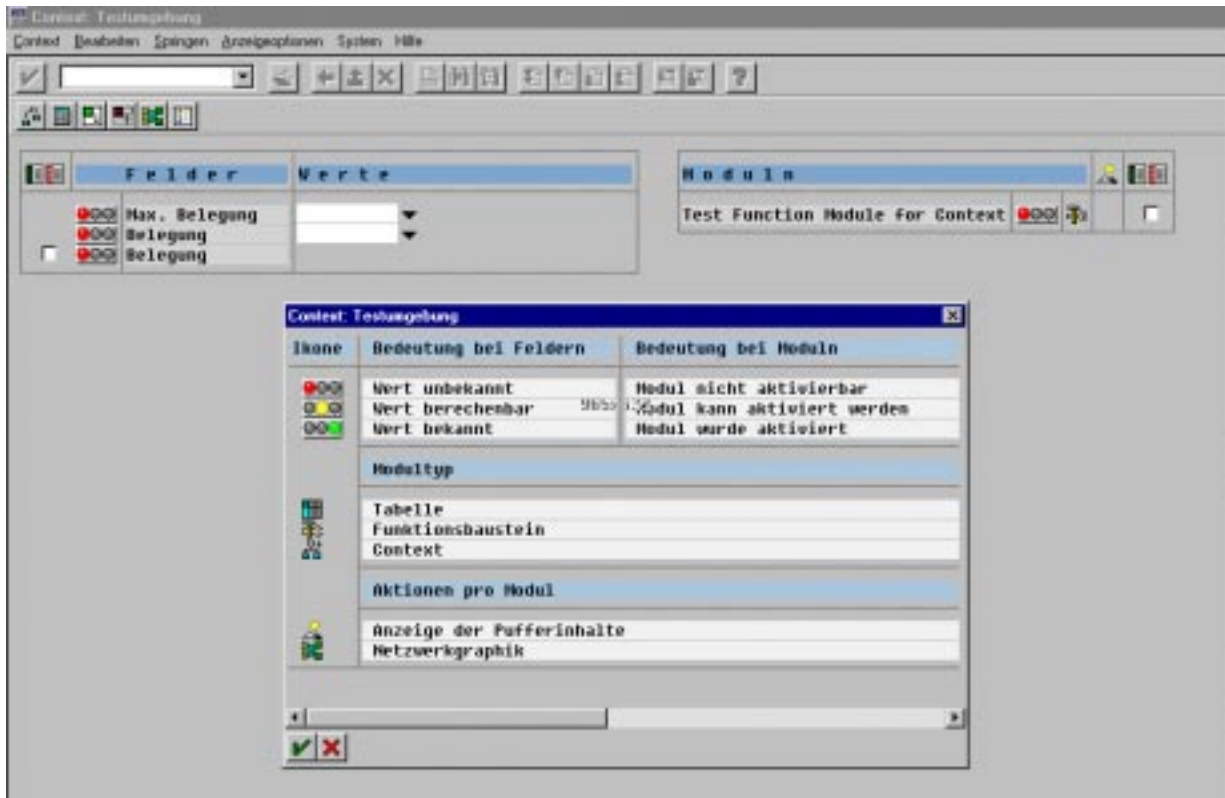
The context is finished and can be tested (see [Testing a Context \[Page 644\]](#)) and used in programs (see [Using Contexts in ABAP Programs \[Page 654\]](#)).

Testing a Context

This section describes how to test a context. It uses the context DOCU_TEST2 from the previous example [Using Function Modules as Modules \[Page 639\]](#).

To test a context, call the Context Builder (Transaction SE33) and enter the name of the context you wish to test in the *Context* field. Choose *Test*.

The *Context Builder: Testing* screen is then displayed. To display the meanings of the various icons, choose *Legend*.



To test the context, enter values in the key fields of the context and choose *Enter*.



If you enter values in all of the input fields, the system will be able to calculate all output fields and will be able to activate the context. Select the output fields required and choose *Calculate values*.

Testing a Context

The screenshot shows the 'Context Builder: Testing' screen. It features two main panels: 'Felder' (Fields) and 'Moduln' (Modules). The 'Felder' panel contains a table with the following data:

Felder	Werte
Max. Belegung	500
Belegung	345
Belegung	155

The 'Moduln' panel contains a list of modules, with 'Test Function Module for Context' selected.

By choosing individual fields, you can use the *Context Builder: Testing* screen to display the relationships between fields and modules. For example, if you choose the *Test Function Module for Context* module, the following is displayed:

The screenshot shows the 'Context Builder: Testing' screen after selecting the 'Test Function Module for Context' module. The 'Felder' panel now displays a diagram showing the relationships between the fields and the module. The 'Moduln' panel shows 'Test Function Module for Context' as the selected module.

Buffering Contexts

The principal aim of contexts is to avoid the repeated calculation of frequently-used data which is derived from other key data. This is achieved by storing the derived data in the context buffer. Data for each module is stored separately in the buffer.

The main difference between the context buffer and the database buffers in the database interface or the SAP table buffer is that it is only periodically refreshed (every hour on the hour). The system makes no attempt to update it synchronously, or even nearly synchronously. For this reason, you cannot use the buffer for every context, or for every module within a context. Rather, you should only use it for data which does not often change.

In the *Buffer type* column of the [Modules \[Page 651\]](#) table, you can set the type of buffer you want to use. This can be the context buffer itself (P), a temporary buffer (T), or no buffer at all (N).

The Individual Buffering Types

- Permanent (P)

This is the default buffer setting, in which the **cross-transaction application buffer** is used. To learn more about this buffer, consult the keyword documentation for EXPORT/IMPORT TO/FROM SHARED BUFFER. The cross-transaction application buffer can contain up to 128 context entries. These entries are structured using a process similar to LRU (Least Recently Used).

To display the contents of the context buffer on the current application server, choose *Goto → Display buffer* in the Context Builder. The buffer is reset every hour on the hour. You can also reset it manually in the Context Builder by choosing *Edit → Reset buffer*. This resets the buffer on **all application servers**.

- Temporary (T)

If you choose this buffer type, the derived data is only buffered within a transaction. **Within** a transaction the buffer can contain up to 1024 entries. These entries are exported in bundles into the cross-transaction application buffer. The results of the various instances of a context are stored in the same buffer within the program.

- No buffer (N)

If you choose *No buffer*, the derived data is not buffered. If you use this buffering type for all modules in a context, using the context will not improve system performance, but is merely a way of re-using program logic.

Permanent buffering can lead to problems when you are testing your Customizing settings. The Context Builder therefore allows you to de-activate the context buffer (buffering type P) during the current terminal session. To do this, choose *Settings → Buffering*. The system displays a dialog box. Select *Inactive*, and choose *Continue*. You can activate or de-activate the context buffer for a particular user by setting the user parameter BUF to Y or SPACE or N (default) using the *Maintain users* transaction (SU01).

Construction of the Buffer

The context buffer contains two buffer tables for each module:

- The I buffer (internal buffer), which saves each combination of module input parameters queried during the lifetime of the buffer, along with their derived values.

Buffering Contexts

- The E buffer (external buffer), which assigns the derived values in the module to the key values in the context. The E buffer is filled each time a query is made. The first time a query is made using a particular combination of module input parameters, the results are written to the I buffer without any reference to the key values for the context. If, however, the same combination of module input parameters is queried more than once (so that the combination already exists in the I buffer), the system writes the results to the E buffer along with the context key values). It is therefore possible for the results of a particular combination of module input parameters to appear more than once in the E buffer, since different combinations of key fields can lead to the same set of module input parameters.

The various modules within a context are linked by the derivation schema. The input parameters of a module are either key fields or output parameters of another module. The entries in the E buffer for modules which depend on other, previous modules are the same as the I buffer entries for those previous modules. The buffering type of the E buffer for the dependent module is the same as the lowest buffering type of any of the modules on which it depends. Thus, if one of the previous modules has buffering type N, the system will not store any entries in the E buffer of the dependent module.

When you access the context buffer, either during testing or using the DEMAND statement, the system first searches in the E buffer of each module, followed by the I buffer. Access to the E buffer is more direct, and quicker than using the I buffer, which usually needs to be accessed more often.

The inclusion of entries in the E buffer which have been queried more than once is an acceptance of data redundancy as the price for quicker access. The I buffer is a filter for the E buffer, only allowing redundant data when absolutely necessary. Together, the I and E buffers provide a balance of quick access time and high probability of finding an appropriate entry.

When you create your own contexts, you should bear in mind the advantages of this buffering concept. Try to include as many relationships between data objects in a single context for each program, rather than using several contexts in the same program. You can also combine a group of contexts as modules of a single, larger context. Within a context, the system buffers all intermediate results in an optimal form. If you are using separate contexts, there is no link between any of the individual buffers.

Deleting a Buffer

You can delete the contents of a context buffer as follows:

- in the Context Builder (Transaction SE33), choose *Edit → Delete buffer → Local server* to delete the buffer contents for a context on the current application server, or *Edit → Delete buffer → All servers* to delete the buffer contents for a context on all application servers.

You can use these functions when testing or buffering contexts.

- in ABAP programs, you can use the function modules `CONTEXT_BUFFER_DELETE_LOCAL` and `CONTEXT_BUFFER_DELETE` to delete the buffer contents for a context on the local server or all servers respectively.

You can use these function modules in conjunction with changes to database contents to ensure that the contents of the context buffer are always current.

The following is an example which you could use in asynchronous update:

```
DATA CONTEXT_NAME LIKE RS33F-FRMID.
CONTEXT_NAME = 'DOCU_TEST1'.
...
CALL FUNCTION 'CONTEXT_BUFFER_DELETE' IN UPDATE TASK
  EXPORTING
    CONTEXT_ID = CONTEXT_NAME
  EXCEPTIONS
    OTHERS      = 0.
....
COMMIT WORK.
```

In the example, the system calls `CONTEXT_BUFFER_DELETE` as an update module. You could use it as the last update call following the database changes before the `COMMIT WORK` statement. For more about asynchronous update, see [Programming Database Updates \[Page 667\]](#)

Fields

Fields

The following illustration shows the *Fields* table for the sample context DEMO_TRAVEL:

Felder					
	Name	Typ	Text	Defaultwert	Like
	CARRID	SPFLI-CARRID	Fluggesellschaft		L
	CONNID	SPFLI-CONNID	Flug-Nummer		L
	CITYFROM	SPFLI-CITYFROM	Ausgangsort		L
	AIRPFROM	SPFLI-AIRPFROM	Startflughafen		L
	CITYTO	SPFLI-CITYTO	Zielort		L
	AIRPTO	SPFLI-AIRPTO	Zielflugh.		L
	FLTIME	SPFLI-FLTIME	Flugzeit		L
	DEPTIME	SPFLI-DEPTIME	Abflug		L
	ARRTIME	SPFLI-ARRTIME	Ankunft		L
	DISTID	SPFLI-DISTID	Entf. in		L
	FLTYPE	SPFLI-FLTYPE	Charter		L
	DISTANCE	SPFLI-DISTANCE	Entfernung		L
	CARRNAME	SCARR-CARRNAME	Fluggesellschaft		L
	CURRCODE	SCARR-CURRCODE	Währ. Carrier		L
	NAME_FROM	SAIRPORT-NAME	Flughafen FROM		L
	NAME_TO	SAIRPORT-NAME	Flughafen TO		L

These, in turn, can become input parameters for further modules. You can enter new modules in the table directly, or fill it automatically by creating fields in the [Modules \[Page 651\]](#) table. You can delete all unneeded fields from the context.

All fields in a context must have a reference to the ABAP Dictionary.

To edit the table, use the selection column and the pushbuttons on the left hand side of the table.

The table columns have the following meanings:

- *Name*
The field names which you use to address the fields in the context.
- *Type*
The Dictionary reference for the field. This can be a column of a database table or structure or a type in a type group.
- *Text*
The short text for the field from the ABAP Dictionary.
- *Default value*
You can enter default values for key fields in this column. If you do this, the key fields of each instance of the context which you create in an application program will already be supplied with these values.
- *Like*
In this column, you specify the type of reference between the field and the ABAP Dictionary. Enter L if the field refers to a database table or structure (LIKE), or T if the

field refers to a type in a type group (TYPE). The system will normally assign the correct value automatically.

Modules

Modules




The following illustration shows the Modules table for the sample context DEMO_TRAVEL:

Name	Type	Table/module	Text	Message ID	Number	Variable 1	Variable 2
DEMO_CITIES	Table	DEMO_CITIES	Context DEMO_CITIES				
SALSPORT1	Table	SALSPORT	Airport 1				
SALSPORT2	Table	SALSPORT	Airport 2				
SCHNR	Table	SCHNR	Address	05	000	CARRID	
SPFLI	Table	SPFLI	Flight schedule				

Modules have input and output parameters, which form its [interface \[Page 653\]](#). Modules describe the relationship between key fields and their dependent fields. Together, the modules of a context determine all of its dependent fields, based on the key fields. Each individual module is responsible for determining a subset of the dependent fields. For example, one module might use some of the key fields to determine dependent fields that are then used as input parameters for further modules.

You can either enter new modules directly into this table, or let the system fill it by creating new fields in the [fields \[Page 649\]](#) table. To edit the table, use the selection column and the pushbuttons to the left of the table.

The table columns have the following meaning:

- *Name*
Contains the names of all modules.
- *Type*
Contains the type of each module. This can be:
 -  a table:
The fields of the primary key of the table are the input parameters. The other fields of the table are output parameters.
 -  a function module:
The import parameters of the function module are input parameters. The export parameters of function modules are output parameters.
 -  another context:
The key fields of contexts are input parameters. The dependent fields of contexts are output parameters.

You can have more than one module with the same type. This allows you to define the same relationship between different groups of fields.

- *Table/Module*
This column contains the name of the table, function module, or context that defines the relationship between the context fields.
- *Text*
This column contains a descriptive text for each module.

- *Message ID, Number, Variable 1,..., Variable 4, Type*

These columns contain the specification for a message in a message class (Message ID and Number) and its type (Type). In the columns *Variable 1,..., Variable 4*, you can enter context variables, whose contents then replace the placeholders '&' in the message. If the context is unable to fill all the required fields in a DEMAND statement (for example, because there is no dependent data in the database for the supplied key fields), the system sends the message. You can catch the message in the application program (see [Message handling in contexts \[Page 661\]](#)).

- *Buffer type*

This column contains the buffering type of each module. For more information, see [Buffering contexts \[Page 646\]](#).

Interfaces

Interfaces

When you select an entry in the [Modules \[Page 651\]](#) table, the system fills the *Interface* table with the corresponding input and output parameters for that module.

When you select a dependent field in the [Fields \[Page 649\]](#) table, the system finds the module which determines the field and fills the *Interface* table with the corresponding input and output parameters for that module.

For example, the interface for the module SPFLI in the sample context DEMO_TRAVEL looks like this:

[illegible]

Module SPFLI is a database table in the ABAP Dictionary. The *Parameters* column contains its input and output parameters, and the *Field name* column contains the associated fields in the context. The direction of the arrow in the *I/O* column shows which parameters are input parameters and which are output parameters.

Using Contexts in ABAP Programs

The following sections explain how to use contexts in your ABAP programs. As with logical databases, you will normally only use contexts supplied by SAP.

[Finding and Displaying a Context \[Page 655\]](#)

[Creating an Instance of a Context \[Page 657\]](#)

[Supplying Context Instances with Key Values \[Page 658\]](#)

[Querying Data from Context Instances \[Page 659\]](#)

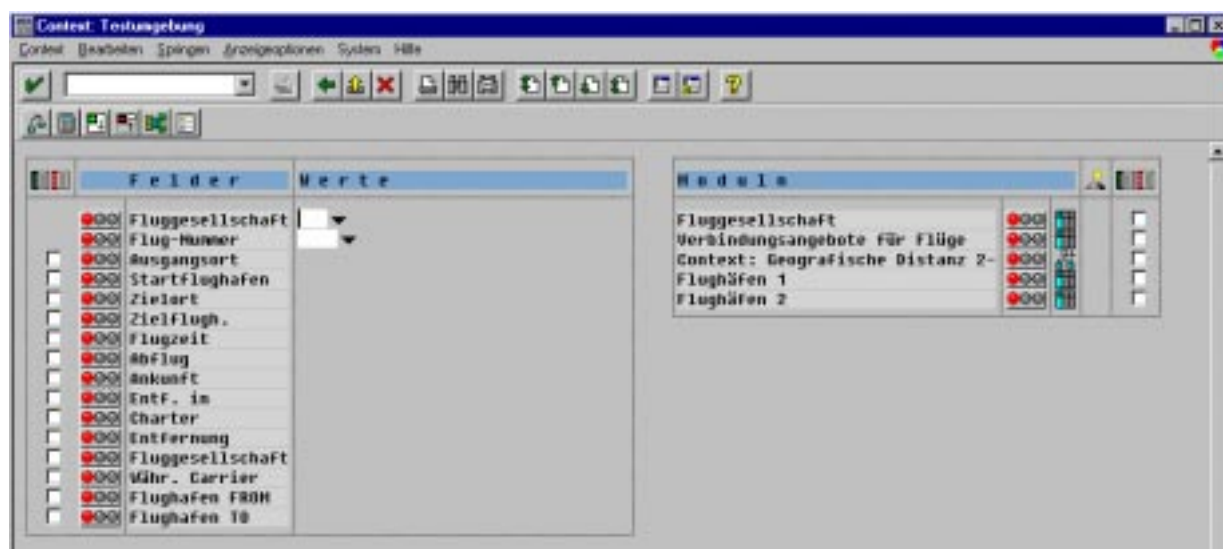
Finding and Displaying a Context

Finding and Displaying a Context

To find an existing context and to familiarize yourself with it, use the initial screen of the Context Builder (Transaction SE33).

- Use the possible entries help for the *Context* field to display a list of existing contexts.
- Use *Test* to display the names of the key fields and dependent fields, and the function of the context (see [Testing a Context \[Page 644\]](#)).
- Use *Network graphic* to display the derivation schema for the context.
- Use *Display* with the [Fields \[Page 649\]](#), [Modules \[Page 651\]](#) and [Interfaces \[Page 653\]](#) tables to display further information about the context field dependencies.

Enter the name DEMO_TRAVEL in the *Context* field on the initial screen of the Context Builder and choose *Test*.



You will see that the context has two key fields, from which the other fields are derived, and that a total of five modules are used to define the relationships between the fields. Four of these modules are database tables, and one is another context. If you choose *Network graphic*, you will see the names of the key fields (CARRID and CONNID) and their relationships to the modules.



To see the names of all context fields, display the [Fields \[Page 649\]](#) table in the Context Builder.

Finding and Displaying a Context

You now have the most important information which you need to work with the context. Further information such as message texts and the buffering types for the modules are contained in the [Modules \[Page 651\]](#) table.

Creating an Instance of a Context

Creating an Instance of a Context

As objects within the ABAP Workbench, contexts store the relationships between data, but no data themselves. In your ABAP programs, you work with instances of contexts. These are runtime instances of contexts containing the actual data.

To create a context instance, you must first declare the context in the program. You do this using the CONTEXTS statement:

Syntax

CONTEXTS <c>.

The context <c> must already exist as an object in the ABAP Workbench. The statement implicitly generates the special data type CONTEXT_<c>, which you can use to create context instances in a DATA statement:

Syntax

DATA <inst> TYPE CONTEXT_<c>.

This statement generates an instance <inst> of context <c>. You can create as many instances of a context as you like. Once you have created an instance, you can supply it with keywords (see [Supplying Context Instances with Key Values \[Page 658\]](#)).

```
REPORT RSGCON01.
```

```
CONTEXTS DEMO_TRAVEL.
```

```
DATA: DEMO_TRAVEL_INST1 TYPE CONTEXT_DEMO_TRAVEL,  
      DEMO_TRAVEL_INST2 TYPE CONTEXT_DEMO_TRAVEL.
```

This program extract generates two instances of the context DEMO_TRAVEL.

Supplying Context Instances with Key Values

Once you have created a context instance, you can supply values for its key fields. You do this using the SUPPLY statement:

Syntax

SUPPLY <key₁> = <f₁>... <key_n> = <f_n> TO CONTEXT <inst>.

This statement supplies the values <f_n> to key fields <key_n> of a context instance <inst>. The fields of the context are contained in the [Fields \[Page 649\]](#) table.

When you have specified values for the key fields, you can derive the dependent fields of the context instance (see [Querying Data from Context Instances \[Page 659\]](#)).

If you assign new key fields to a context instance after deriving the dependent fields, the derived values are automatically invalidated, and will be re-calculated the next time you derive dependent values.

```
REPORT RSGCON01.
CONTEXTS DEMO_TRAVEL.
DATA: DEMO_TRAVEL_INST1 TYPE CONTEXT_DEMO_TRAVEL,
      DEMO_TRAVEL_INST2 TYPE CONTEXT_DEMO_TRAVEL.
SUPPLY CARRID = 'LH'
      CONNID = '400'
      TO CONTEXT DEMO_TRAVEL_INST1.
SUPPLY CARRID = 'AA'
      CONNID = '017'
      TO CONTEXT DEMO_TRAVEL_INST2.
```

This program extract generates two instances of the context DEMO_TRAVEL and supplies both with key values.

Querying Data from Context Instances

Querying Data from Context Instances

Once you have supplied a context instance with key values, you can query its dependent values. You do this using the DEMAND statement:

Syntax

```
DEMAND <val1> = <f1>... <valn> = <fn> FROM CONTEXT <inst>
      [MESSAGES INTO <itab>].
```

This statement fills the fields <f_n> with the derived values <val_n> from context instance <inst>. The fields of the context are contained in the [Fields \[Page 649\]](#) table.

In doing this, the system carries out the following steps:

1. It checks to see whether the context instance already contains valid derived values. This is the case if the values have already been calculated in a previous DEMAND statement and the instance has not since been supplied with new key field values using the SUPPLY statement. In this case, these values are assigned to fields <f_n>.
2. If the context instance does not contain valid values, the system calculates new ones. It looks first in the context buffer (see [Buffering Contexts \[Page 646\]](#)) for data records with the right key field values for the current context instance. If it finds the corresponding values, the system copies them as valid values into the context instance and assigns them to fields <f_n>.
3. If there are no appropriate values in the context buffer, the system derives the values according to the context definition. The system also searches the context buffer during the derivation for intermediate results, which it uses if they are valid. When it has derived the values, the system saves all results in the context buffer, copies the values to the context instance and assigns them to fields <f_n>.
4. If the system cannot determine the dependent values, it terminates the process, sets fields <f_n> to their initial values and outputs the user message stored in the [Modules \[Page 651\]](#) table. You can handle these messages in your programs by using the MESSAGES option (see [Message Handling In Contexts \[Page 661\]](#)).

```
REPORT RSGCON01.

DATA: C_FROM LIKE SPFLI-CITYFROM,
      C_TO   LIKE SPFLI-CITYTO,
      C_TIME LIKE SPFLI-FLTIME.

CONTEXTS DEMO_TRAVEL.

DATA: DEMO_TRAVEL_INST1 TYPE CONTEXT_DEMO_TRAVEL,
      DEMO_TRAVEL_INST2 TYPE CONTEXT_DEMO_TRAVEL.

SUPPLY CARRID = 'LH'
      CONNID = '400'
      TO CONTEXT DEMO_TRAVEL_INST1.

SUPPLY CARRID = 'AA'
      CONNID = '017'
      TO CONTEXT DEMO_TRAVEL_INST2.

DEMAND CITYFROM = C_FROM
      CITYTO   = C_TO
```

Querying Data from Context Instances

```
      FLTIME  = C_TIME  
      FROM CONTEXT DEMO_TRAVEL_INST1.  
WRITE: / C_FROM, C_TO, C_TIME.  
DEMAND CITYFROM = C_FROM  
      CITYTO  = C_TO  
      FLTIME  = C_TIME  
      FROM CONTEXT DEMO_TRAVEL_INST2.  
WRITE: / C_FROM, C_TO, C_TIME.
```

This program generates two instances of context DEMO_TRAVEL, supplies them both with key values and reads three dependent values from each of them.

Message Handling in Contexts

If a context cannot derive dependent data when you test it or execute the DEMAND statement, it can send message to the user.

The way in which messages are handled in contexts depends on the type of module in which the context could not determine data. Table and function module modules handle messages differently. Other contexts used as modules can be broken down to table and function module level.

[Message Handling in Table Modules \[Page 662\]](#)

[Message Handling in Function Module Modules \[Page 664\]](#)

Message Handling in Table Modules

If you want to send user messages from table modules, you must first have defined messages for the individual modules in the [Modules \[Page 651\]](#) table. The system cannot output a message for a table module where none is defined, but will simply reset the corresponding dependent values.

When you query dependent data from context instances using the DEMAND statement, you can decide whether the system should handle the user message or whether you want to handle it yourself in the program.

Message Handling - System

If you want the system to handle the message, use the basic form of the DEMAND statement without the MESSAGES option. The system will then behave as though the following statement occurred after the DEMAND statement:

```
MESSAGE ID 'Message Id' TYPE 'T' NUMBER 'Number'
  WITH 'Variable 1'.... 'Variable 4'.
```

The system takes the arguments for the MESSAGE statement from the corresponding entries in the [Modules \[Page 651\]](#) table. Note that the system reaction to the various message types depends on the current user dialog (dialog screen, selection screen or list).

Message Handling - Program

If you want to catch the message in your program, use the DEMAND statement using the option MESSAGES INTO <itab>. To do this, you need to define an internal table <itab> with the structure SYMSG. The system deletes the contents of <itab> table at the beginning of the DEMAND statement. Whilst it is processing the context, the system does not output or react to any messages, but writes their ID to <itab>. If there are messages in <itab> following the DEMAND statement, the system sets the return code SY-SUBRC to a value unequal to zero.

This enables you to prevent automatic message handling with contexts and allows you to program your own using the entries in <itab>. This is important, for example, if you want to make particular fields on an entry screen ready for input again. In this case, you must base your message handling on the fields (using the FIELDS statement in screen flow logic, or the ABAP statement AT SELECTION-SCREEN for selection screens), and not on fields in the context.

Suppose the [Modules \[Page 651\]](#) table in context DOCU_TEST1 (see Using Tables as [Modules \[Page 636\]](#)) contained the following entries for module SPFLI:

Nachrichten-Id	Nr.	Variable 1	Variable 2	Variable 3	Variable 4	T
AT	107	CARRID	CONNID			E

To demonstrate system message handling, execute the following program:

```
REPORT RSGCON02.

DATA: C_FROM LIKE SPFLI-CITYFROM,
      C_TO   LIKE SPFLI-CITYTO.

CONTEXTS DOCU_TEST1.

DATA: CONTEXT_INST TYPE CONTEXT_DOCU_TEST1.

SUPPLY CARRID = 'XX'
      CONNID = '400'
      TO CONTEXT_CONTEXT_INST.
```

Message Handling in Table Modules

```
DEMAND CITYFROM = C_FROM  
      CITYTO    = C_TO  
      FROM CONTEXT CONTEXT_INST.
```

```
WRITE: / C_FROM, C_TO.
```

The program terminates with the following error message:

```
No entries found for key XX 0400
```

To demonstrate program message handling, execute the following program:

```
REPORT  RSGCON03.  
  
DATA: C_FROM LIKE SPFLI-CITYFROM,  
      C_TO   LIKE SPFLI-CITYTO.  
  
DATA ITAB LIKE SYMSG OCCURS 0 WITH HEADER LINE.  
CONTEXTS DOCU_TEST1.  
  
DATA: CONTEXT_INST TYPE CONTEXT_DOCU_TEST1.  
  
SUPPLY CARRID = 'XX'  
      CONNID  = '400'  
      TO CONTEXT CONTEXT_INST.  
  
DEMAND CITYFROM = C_FROM  
      CITYTO    = C_TO  
      FROM CONTEXT CONTEXT_INST MESSAGES INTO ITAB.  
  
WRITE: / C_FROM, C_TO.  
  
LOOP AT ITAB.  
  WRITE: / ITAB-MSGTY, ITAB-MSGID, ITAB-MSGNO,  
        ITAB-MSGV1, ITAB-MSGV2.  
ENDLOOP.
```

The program outputs the following:

```
E  AT  107  XX  0400
```

Instead of terminating with an error message, the program stores the message in <itab> where it is available for you to process further.

Message Handling in Function Module Modules

In order for user messages to be sent from function module modules, you must program exception handling in the function module using the MESSAGE... RAISING statement. If the exception handling in the function module is programmed using the RAISE statement, the system reacts with a runtime error. For further information about exception handling in function modules, see [Programming Function Modules \[Page 494\]](#)

The system sends the message specified in the MESSAGE... RAISING statement. Messages specified for function modules in the [Modules \[Page 651\]](#) table are ignored.

When you query dependent data from context instances using the DEMAND statement, you can decide whether the system should handle the user message or whether you want to handle it yourself in the program.

Message Handling - System

If you want the system to handle the message, use the basic form of the DEMAND statement without the MESSAGES option. The system will then behave as though the MESSAGE statement in the function module occurred after the DEMAND statement. Note that the system reaction to the various message types depends on the current user dialog (dialog screen, selection screen or list).

Message Handling - Program

If you want to catch the message in your program, use the DEMAND statement using the option MESSAGES INTO <itab>. To do this, you need to define an internal table <itab> with the structure SYMSG. The system deletes the contents of <itab> table at the beginning of the DEMAND statement. Whilst it is processing the context, the system does not output or react to any messages, but writes their ID to <itab>. If there are messages in <itab> following the DEMAND statement, the system sets the return code SY-SUBRC to a value unequal to zero.

This enables you to prevent automatic message handling with contexts and allows you to program your own using the entries in <itab>. This is important, for example, if you want to make particular fields on an entry screen ready for input again. In this case, you must base your message handling on the fields (using the FIELDS statement in screen flow logic, or the ABAP statement AT SELECTION-SCREEN for selection screens), and not on fields in the context.

The function module PERCENT has the following source code:

```
FUNCTION PERCENT.  
  RESULT = V1 - V2.  
  IF V1 = 0.  
    MESSAGE ID 'AT' TYPE 'E' NUMBER '012' RAISING DIV_ZERO.  
  ELSE.  
    RESULT = RESULT * 100 / V1.  
  ENDIF.  
ENDFUNCTION.
```

Context DOCU_TEST4 uses this function module as its only module, with key fields MAX and OCC for input parameters V1 and V2 and the dependent field PERCENT for the output parameter RESULT.

To demonstrate system message handling, execute the following program:

Message Handling in Function Module Modules

```
REPORT RSGCON04.  
DATA: INPUT_1 TYPE I,  
      INPUT_2 TYPE I,  
      PERCENTAGE TYPE I.  
CONTEXTS DOCU_TEST4.  
DATA: CONTEXT_INST TYPE CONTEXT_DOCU_TEST4.  
SUPPLY MAX = 00  
      OCC = 20  
      TO CONTEXT CONTEXT_INST.  
DEMAND PERCENT = PERCENTAGE  
      FROM CONTEXT CONTEXT_INST.  
WRITE: / 'Percentage: ', PERCENTAGE.
```

The program terminates with the following error message:

This is an error message

To demonstrate program message handling, execute the following program:

```
REPORT RSGCON05.  
DATA: INPUT_1 TYPE I,  
      INPUT_2 TYPE I,  
      PERCENTAGE TYPE I.  
CONTEXTS DOCU_TEST4.  
DATA: CONTEXT_INST TYPE CONTEXT_DOCU_TEST4.  
DATA ITAB LIKE SYMSG OCCURS 0 WITH HEADER LINE.  
SUPPLY MAX = 00  
      OCC = 20  
      TO CONTEXT CONTEXT_INST.  
DEMAND PERCENT = PERCENTAGE  
      FROM CONTEXT CONTEXT_INST MESSAGES INTO ITAB.  
WRITE: / 'Percentage: ', PERCENTAGE.  
LOOP AT ITAB.  
  WRITE: / ITAB-MSGTY, ITAB-MSGID, ITAB-MSGNO,  
         ITAB-MSGV1, ITAB-MSGV2.  
ENDLOOP.
```

The program outputs the following:

PERCENTAGE: 0

E AT 012

Instead of terminating with an error message, the program stores the message in <itab> where it is available for you to process further.

Working With Contexts - Hints

Creating Contexts

- You should include only a small number of key fields in your contexts.
- All of the module input parameters in the context should be either key fields or output parameters of other modules and should be structured hierarchically. If this is not the case, you should split the context into two or more smaller contexts. See also the hint in [Buffering Contexts \[Page 646\]](#)
- You should restrict the number of derived fields to those you really need, because:
 - Reading fewer fields reduces the load on the database
 - Fewer fields means that the buffer requires less space
 - The system can then read the buffer more quickly
 - You can always add more fields later if required.
- If you use a table, a function module or a context more than once as a module within a context, the module name should be made up as follows:<object name>_<suffix>. When you test your context, the suffix is automatically displayed after the long text.

Using Contexts

- Since the SUPPLY and DEMAND statements are not runtime-intensive, using them repeatedly causes no problems. In particular,
 - You should always use the SUPPLY command as soon as the key values are assigned. This reduces the risk of deriving obsolete values in the DEMAND statement. You do not need to make a preliminary query to see if the contents of the key fields have changed, since the system does this itself in the SUPPLY command.
 - You should always use the DEMAND statement immediately before using the derived values. This is to ensure that you always use up-to-date values.
- You should use local data objects as target fields for the DEMAND statement. This reduces the risk of obsolete values being used in error.

Programming Database Updates

This section describes how to program database updates in the R/3 System.

See also the descriptions in the ABAP Editor under Utilities → Help on → ABAP Overview → Description of Syntax and Concepts → Transaction processing.

For detailed information about the statements that you use for database updates, see the keyword documentation in the ABAP Editor.

Transactions and Logical Units of Work

In everyday language, a transaction is a sequence of actions that logically belong together in a business sense and which either procure or process data. It covers a self-contained procedure, for example, generating a list of customers, creating a flight booking, or sending reminders to customers. From the point of view of the user, it forms a logical unit.

The completeness and correctness of data must be assured within this unit. In the middle of a transaction, the data will usually be inconsistent. For example, when you transfer an amount in financial accounting, this must first be deducted from one account before being credited to another. In between the two postings, the data is inconsistent, since the amount that you are posting does not exist in either account. It is essential for application programmers to know that their data is consistent at the end of the transaction. If an error occurs, it must be possible to undo the changes made within a logical process.

In the R/3 System, there are three terms frequently used in this connection:

Database Logical Unit of Work (LUW)

A database LUW is the mechanism used by the database to ensure that its data is always consistent.

SAP LUW

An SAP LUW is a logical unit consisting of dialog steps, whose changes are written to the database in a single database LUW.

SAP Transaction

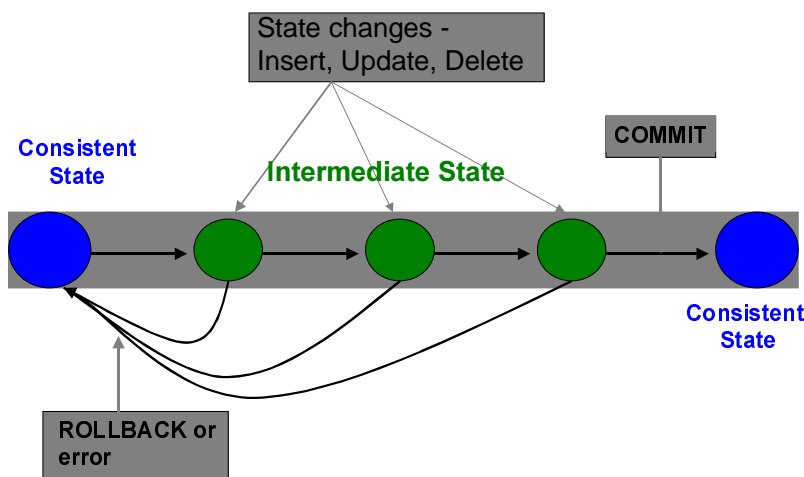
An SAP transaction is an application program that you start using a transaction code. It may contain one or more SAP LUWs.

The following sections of this documentation explain these three terms in more detail.

Database Logical Unit of Work (LUW)

Database Logical Unit of Work (LUW)

From the point of view of database programming, a database LUW is an inseparable sequence of database operations that ends with a database commit. The database LUW is either fully executed by the database system or not at all. Once a database LUW has been successfully executed, the database will be in a consistent state. If an error occurs within a database LUW, all of the database changes since the beginning of the database LUW are reversed. This leaves the database in the state it had before the transaction started.



The database changes that occur within a database LUW are not actually written to the database until after the database commit. Until this happens, you can use a database rollback to reverse the changes. In the R/3 System, database commits and rollbacks can be triggered either implicitly or using explicit commands.

Implicit Database Commits in the R/3 System

A work process can only execute a single database LUW. The consequence of this is that a work process must always end a database LUW when it finishes its work for a user or an external call. There are four cases in which work processes trigger an implicit database commit:

- When a dialog step is completed
Control changes from the work process back to the SAPgui.
- When a function module is called in another work process (RFC).
Control passes to the other work process.
- When the called function module (RFC) in the other work process ends.
Control returns to the calling work process.
- Error dialogs (information, warning, or error messages) in dialog steps.
Control passes from the work process to the SAPgui.

Explicit Database Commits in the R/3 System

There are two ways to trigger an explicit database commit in your application programs:

- Call the function module DB_COMMIT
The sole task of this function module is to start a database commit.
- Use the ABAP statement COMMIT WORK
This statement starts a database commit, but also performs other tasks (refer to the keyword documentation for COMMIT WORK).

Implicit Database Rollbacks in the R/3 System

The following cases lead to an implicit database rollback:

- Runtime error in an application program
This occurs whenever an application program has to terminate because of an unforeseen situation (for example, trying to divide by zero).
- Termination message
Termination messages are generated using the ABAP statement MESSAGE with the message type A or X. In certain cases (updates), they are also generated with message types I, W, and E. These messages end the current application program.

Explicit Database Rollbacks in the R/3 System

You can trigger a database rollback explicitly using the ABAP statement ROLLBACK WORK. This statement starts a database rollback, but also performs other tasks (refer to the keyword documentation for COMMIT WORK).

From the above, we can draw up the following list of points at which database LUWs begin and end.

A Database LUW Begins

- Each time a dialog step starts (when the dialog step is sent to the work process).
- Whenever the previous database LUW ends in a database commit.
- Whenever the previous database LUW ends in a database rollback.

A Database LUW Ends

- Each time a database commit occurs. This writes all of the changes to the database.
- Each time a database rollback occurs. This reverses all of the changes made during the LUW.

Database LUWs and Database Locks

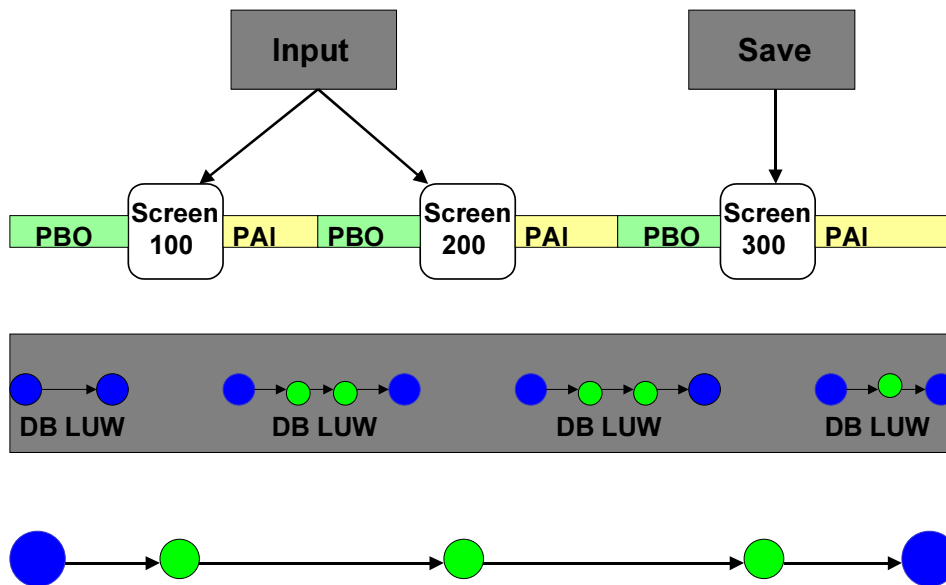
As well as the database changes made within it, a database LUW also consists of database locks. The database system uses locks to ensure that two or more users cannot change the same data simultaneously, since this could lead to inconsistent data being written to the database. A database lock can only be active for the duration of a database LUW. They are automatically released when the database LUW ends. In order to program SAP LUWs, we need a

Database Logical Unit of Work (LUW)

lock mechanism within the R/3 System that allows us to create locks with a longer lifetime (refer to [The R/3 Locking Concept \[Page 677\]](#).)

SAP LUW

The Open SQL statements INSERT, UPDATE, MODIFY, and DELETE allow you to program database changes that extend over several dialog steps. Even if you have not explicitly programmed a database commit, the implicit database commit that occurs after a screen has been processed concludes the database LUW. The following diagram shows the individual database LUWs in a typical screen sequence:

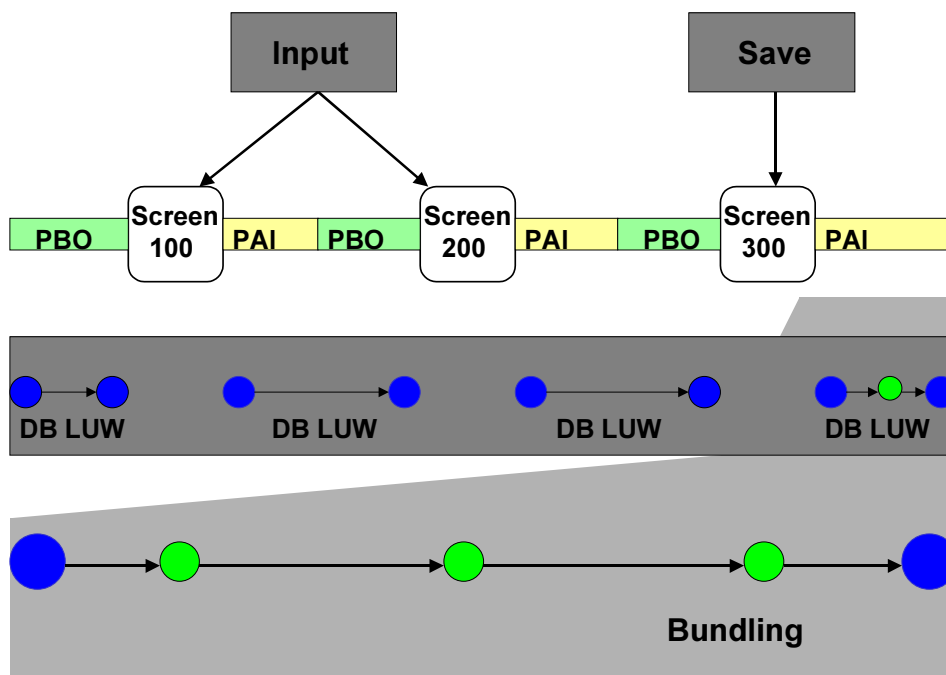


Under this procedure, you cannot roll back the database changes from previous dialog steps. It is therefore only suitable for programs in which there is no logical relationship between the individual dialog steps.

However, the database changes in individual dialog steps normally depend on those in other dialog steps, and must therefore all be executed or rolled back together. These dependent database changes form logical units, and can be grouped into a single database LUW using the bundling techniques listed below.

A logical unit consisting of dialog steps, whose changes are written to the database in a single database LUW is called an SAP LUW. Unlike a database LUW, an SAP LUW can span several dialog steps, and be executed using a series of different work processes. If an SAP LUW contains database changes, you should either write all of them or none at all to the database. To ensure that this happens, you must include a database commit when your transaction has ended successfully, and a database rollback in case the program detects an error. However, since database changes from a database LUW cannot be reversed in a subsequent database LUW, you must make all of the database changes for the SAP LUW in a single database LUW. To maintain data integrity, you must bundle all of your database changes in the final database LUW of the SAP LUW. The following diagram illustrates this principle:

SAP LUW



The bundling technique for database changes within an SAP LUW ensures that you can still reverse them. It also means that you can distribute a transaction across more than one work process, and even across more than one R/3 System. The possibilities for bundling database changes within an SAP LUW are listed below:

The simplest form of bundling would be to process a whole application within a single dialog step. Here, the system checks the user's input and updates the database without a database commit occurring within the dialog step itself. Of course, this is not suitable for complex business processes. Instead, the R/3 Basis system contains the following bundling techniques.

Bundling using Function Modules for Updates

If you call a function module using the `CALL FUNCTION... IN UPDATE TASK` statement, the function module is flagged for execution using a special update work process. This means that you can write the Open SQL statements for the database changes in the function module instead of in your program, and call the function module at the point in the program where you would otherwise have included the statements. When you call a function module using the `IN UPDATE TASK` addition, it and its interface parameters are stored as a log entry in a special database table called VBLOG.

The function module is executed using an update work process when the program reaches the `COMMIT WORK` statement. After the `COMMIT WORK` statement, the dialog work process is free to receive further user input. The dialog part of the transaction finishes with the `COMMIT WORK` statement. The update part of the SAP LUW then begins, and this is the responsibility of the update work process. The SAP LUW is complete once the update process has committed or rolled back all of the database changes.

For further information about how to create function modules for use in update, refer to [Creating Function Modules for Database Updates \[Page 690\]](#)

During the update, errors only occur in exceptional cases, since the system checks for all logical errors, such as incorrect entries, in the dialog phase of the SAP LUW. If a logical error occurs, the

program can terminate the update using the ROLLBACK WORK statement. Then, the function modules are not called, and the log entry is deleted from table VBLOG. Errors during the update itself are usually technical, for example, memory shortage. If a technical error occurs, the update work process triggers a database rollback, and places the log entry back into VBLOG. It then sends a mail to the user whose dialog originally generated the VBLOG entry with details of the termination. These errors must be corrected by the system administrator. After this, the returned VBLOG entries can be processed again.

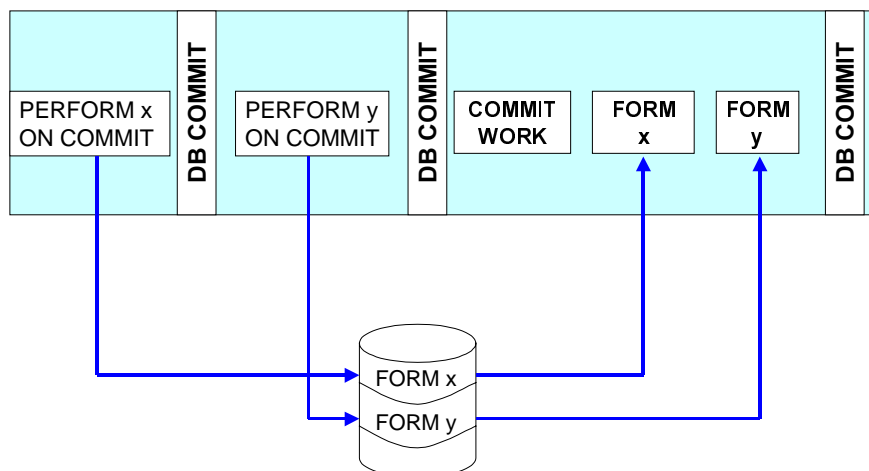
For further information about update administration, see [Update Administration \[Ext.\]](#)

This technique of bundling database changes in the last database LUW of the SAP LUW allows you to update the database asynchronously, reducing the response times in the dialog work process. You can, for example, decouple the update entirely from the dialog work process and use a central update work process on a remote database server.

Bundling Using Subroutines

The statement PERFORM ON COMMIT calls a subroutine in the dialog work process. However, it is not executed until the system reaches the next COMMIT WORK statement. Here, as well, the ABAP statement COMMIT WORK defines the end of the SAP LUW, since all statements in a subroutine called with PERFORM ON COMMIT that make database changes are executed in the database LUW of the corresponding dialog step.

Update in Dialog Work Process



The advantage of this bundling technique against CALL FUNCTION... IN UPDATE TASK is better performance, since the update data does not have to be written into an extra table. The disadvantage, however, is that you cannot pass parameters in a PERFORM... ON COMMIT statement. Data is passed using global variables and ABAP memory. There is a considerable danger of data inconsistency when you use this method to pass data.

Bundling Using Function Modules in Other R/3 Systems

Function modules that you call using CALL FUNCTION... IN BACKGROUND TASK DESTINATION... are registered for background execution in another R/3 System when the program reaches the next COMMIT WORK statement (using Remote Function Call). After the

SAP LUW

COMMIT WORK, the dialog process does not wait for these function modules to be executed (asynchronous update). All of the function modules that you register in this way are executed together in a single database LUW. These updates are useful, for example, when you need to maintain identical data in more than one database.

For further details, refer to the keyword documentation.

For more details of RFC processing, refer to the *Remote Communications* section of the *Basis Services* documentation.

SAP Transactions

An SAP LUW is a logical unit consisting of dialog steps, whose changes are written to the database in a single database LUW. In an application program, you end an SAP LUW with either the COMMIT WORK or ROLLBACK WORK statement. An SAP transaction is an application program that you start using a transaction code. It may contain one or more SAP LUWs. Whenever the system reaches a COMMIT WORK or ROLLBACK WORK statement that is not at the end of the last dialog step of the SAP transaction, it opens a new SAP LUW.

If a particular application requires you to write a complex transaction, it can often be useful to arrange logical processes within the SAP transaction into a sequence of individual SAP LUWs. You can structure SAP transactions as follows:

- With one or more SAP LUWs.

Transactions in this form consist entirely of processing blocks (dialog modules, event blocks, function module calls, and subroutines). You should be careful to ensure that external subroutines or function modules do not lead to COMMIT WORK or ROLLBACK WORK statements accidentally being executed.

- By inserting an SAP LUW

The ABAP statements CALL TRANSACTION (start a new transaction), SUBMIT (start an executable program), and CALL FUNCTION... DESTINATION (call a function module using RFC) open a new SAP LUW. When you call a program, it always opens its own SAP LUW. However, it does not end the LUW of the SAP transaction that called it. This means that a COMMIT WORK or ROLLBACK WORK statement only applies to the SAP LUW of the called program. When the new LUW is complete, the system carries on processing the first SAP LUW.

- By running two SAP LUWs in parallel

The CALL FUNCTION... STARTING NEW TASK statement calls a function module asynchronously in a new session. Unlike normal function module calls, the calling transaction carries on with its own processing as soon as the function module has started, and does not wait for it to finish processing. The function call is asynchronous. The called function module can now call its own screens and interact with the user.

The R/3 Lock Concept

The R/3 Lock Concept

Reasons for Setting Locks

Suppose a travel agent wants to book a flight. The customer wants to fly to a particular city with a certain airline on a certain day. The booking must only be possible if there are still free places on the flight. To avoid the possibility of overbooking, the database entry corresponding to the flight must be locked against access from other transactions. This ensures that one user can find out the number of free places, make the booking, and change the number of free places without the data being changed in the meantime by another transaction.

Lock Mechanisms in the Database System

The database system automatically sets database locks when it receives change statements (INSERT, UPDATE, MODIFY, DELETE) from a program. Database locks are physical locks on the database entries affected by these statements. You can only set a lock for an existing database entry, since the lock mechanism uses a lock flag in the entry. These flags are automatically deleted in each database commit. This means that database locks can never be set for longer than a single database LUW; in other words, a single dialog step in an R/3 application program.

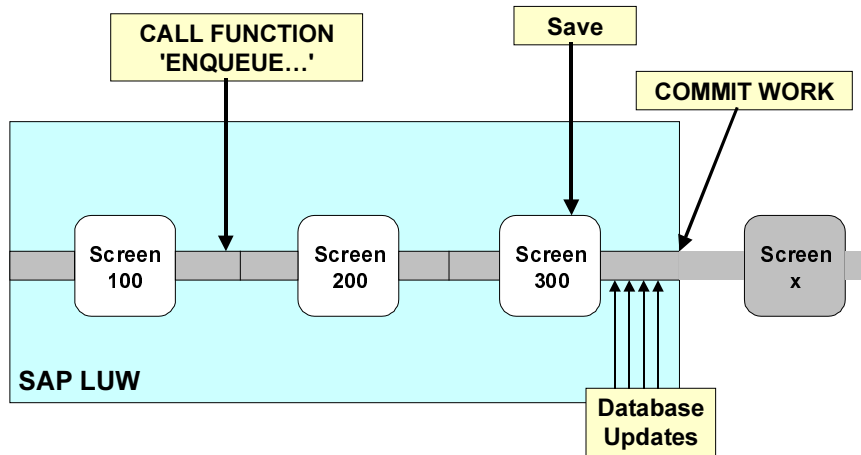
Physical locks in the database system are therefore insufficient for the requirements of an R/3 transaction. Locks in the R/3 System must remain set for the duration of a whole SAP LUW, that is, over several dialog steps. They must also be capable of being handled by different work processes and even different application servers. Consequently, each lock must apply on all servers in that R/3 System.

SAP Locks

To complement the SAP LUW concept, in which bundled database changes are made in a single database LUW, the R/3 System also contains a lock mechanism, fully independent of database locks, that allows you to set a lock that spans several dialog steps. These locks are known as **SAP locks**.

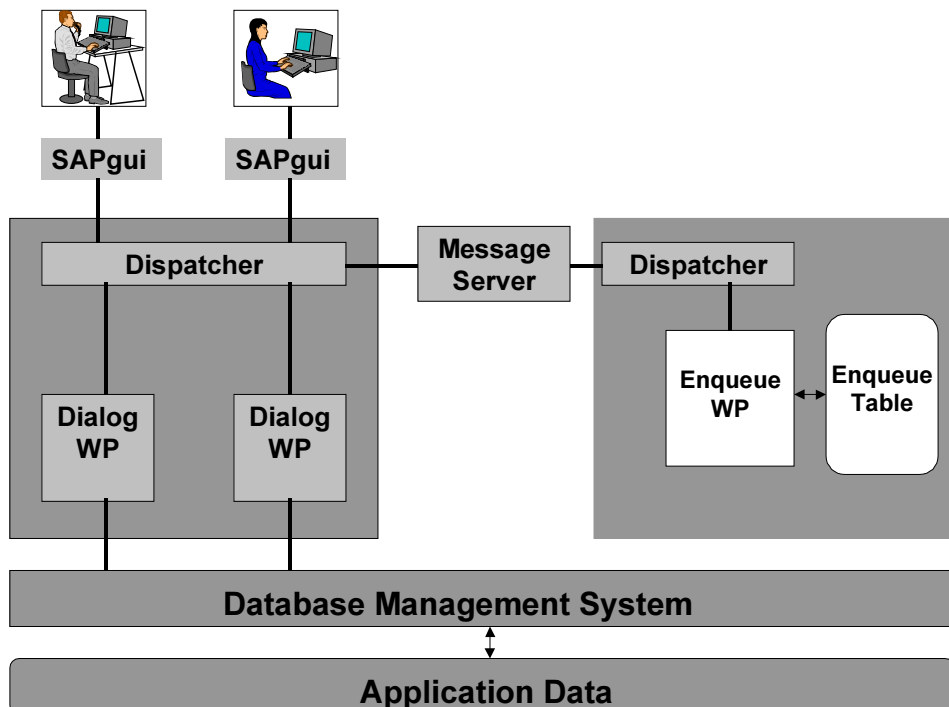
The SAP lock concept is based on **lock objects**. Lock objects allow you to set an SAP lock for an entire application object. An application object consists of one or more entries in a database table, or entries from more than one database table that are linked using foreign key relationships.

Before you can set an SAP lock in an ABAP program, you must first create a lock object in the ABAP Dictionary. A lock object definition contains the database tables and their key fields on the basis of which you want to set a lock. When you create a lock object, the system automatically generates two function modules with the names `ENQUEUE_<lock object name>` and `DEQUEUE_<lock object name>`. You can then set and release SAP locks in your ABAP program by calling these function modules in a `CALL FUNCTION` statement.



See also: [Example Transaction: SAP Locking \[Page 681\]](#).

These function modules are executed in a special enqueue work process. Within an R/3 System, enqueue work processes run on a single application server. This server maintains a central lock table for the entire R/3 System in its shared memory.



The enqueue function module sets an SAP lock by writing entries in the central lock table. If the lock cannot be set because the application object (or a part of it) is already locked, this is

The R/3 Lock Concept

reflected in the return code sy-subrc. The following diagram shows the components of the R/3 System that are involved in setting a lock.

Unlike the database, which sets physical locks, the SAP lock mechanism sets logical locks. This means that

- A locked database entry is not physically locked in the database table.
The lock entry is merely entered as a lock argument in the central R/3 lock table. The lock argument is made up of the primary key field values for the tables in the lock object. These are import parameters of the enqueue function module. The lock is independent of database LUWs. It is released either implicitly when the database update or the SAP transaction ends, or explicitly, using the corresponding dequeue function module. You can use a special parameter in the update function module to set the exact point at which the lock is released during the database update.
- A locked entry does not necessarily have to exist in a database table.
You can, for example, set a lock as a precaution for a database entry that is not written to the database until the update at the end of the SAP LUW.
- The effectiveness of the locks depends on cooperative application programming.
Since there are no physical locks in the database tables themselves, all programs that use the same application objects must look in the central table themselves for any locks. There is no mechanism that automatically prevents a program from ignoring the locks in the lock table.

Lock Types

There are two types of lock in the R/3 System:

- Shared lock
Shared locks (or read locks) allow you to prevent data from being changed while you are reading it. They prevent other programs from setting an exclusive lock (write lock) to change the object. It does not, however, prevent other programs from setting further read locks.
- Exclusive lock
Exclusive locks (or write locks) allow you to prevent data from being changed while you are changing it yourself. An exclusive lock, as its name suggests, locks an application object for exclusive use by the program that sets it. No other program can then set either a shared lock or an exclusive lock for the same application object.

Lock Duration

When you set a lock, you should bear in mind that if it remains set for a long time, the availability of the object to other transactions is reduced. Whether or not this is acceptable depends on the nature of the task your program is performing.

Remember in particular that setting too many shared locks without good reason can have a considerable effect on programs that work with database tables. If several programs running concurrently all set a shared lock for the same application object in the system, it can make it almost impossible to set an exclusive lock, since the program that needs to set that lock will be unable to find any time when there are no locks at all set for that object. Conversely, a single exclusive lock prevents all other programs from reading the locked object.

At the end of an SAP LUW, you should release all locks. This either happens automatically during the database update, or explicitly, when you call the corresponding dequeue function module. Locks that are not linked to a database update are released at the end of the SAP transaction.

Example Transaction: SAP Locking

Example Transaction: SAP Locking

The following example transaction shows how you can lock and unlock database entries using a lock object. It lets the user request a given flight (on screen 100) and display or update it (on screen 200). If the user chooses *Change*, the table entry is locked; if he or she chooses *Display*, it is not.

The example uses the lock object ESFLIGHT and its function modules ENQUEUE_ESFLIGHT and DEQUEUE_ESFLIGHT to lock and unlock the object.

For more information about creating lock objects and the corresponding function modules, refer to the *Lock objects* section of the *ABAP Dictionary* documentation.

The PAI processing for screen 100 in this transaction processes the user input and prepares for the requested action (*Change* or *Display*). If the user chooses *Change*, the program locks the relevant database object by calling the corresponding ENQUEUE function.

```
MODULE USER_COMMAND_0100 INPUT.
  CASE OK_CODE.
    WHEN 'SHOW'....
    WHEN 'CHNG'.
      * <...Authority-check and other code...>
      CALL FUNCTION 'ENQUEUE_ESFLIGHT'
        EXPORTING
          MANDT   = SY-MANDT
          CARRID  = SPFLI-CARRID
          CONNID  = SPFLI-CONNID
        EXCEPTIONS
          FOREIGN_LOCK = 1
          SYSTEM_FAILURE = 2
          OTHERS = 3.
      IF SY-SUBRC NE 0.
        MESSAGE ID SY-MSGID
          TYPE 'E'
          NUMBER SY-MSGNO
          WITH SY-MSGV1 SY-MSGV2 SY-MSGV3 SY-MSGV4.
      ENDIF.
  ...
```

Example Transaction: SAP Locking

The ENQUEUE function module can trigger the following exceptions:

FOREIGN_LOCK determines whether a conflicting lock already exists. The system variable SY-MSGV1 contains the name of the user that owns the lock.

The SYSTEM_FAILURE exception is triggered if the enqueue server is unable to set the lock for technical reasons.

At the end of a transaction, the locks are released automatically. However, there are exceptions if you have called update routines within the transaction. You can release a lock explicitly by calling the corresponding DEQUEUE module. As the programmer, you must decide for yourself the point at which it makes most sense to release the locks (for example, to make the data available to other transactions).

If you need to use the DEQUEUE function module call several times in a program, it makes good sense to write it in a subroutine, which you then call as required.

The subroutine UNLOCK_FLIGHT calls the DEQUEUE function module for the lock object ESFLIGHT:

```
FORM UNLOCK_FLIGHT.
  CALL FUNCTION 'DEQUEUE_ESFLIGHT'
    EXPORTING
      MANDT    = SY-MANDT
      CARRID   = SPFLI-CARRID
      CONNID   = SPFLI-CONNID
    EXCEPTIONS
      OTHERS   = 1.
  SET SCREEN 100.
ENDFORM.
```

You might use this for the BACK and EXIT functions in a PAI module for screen 200 in this example transaction. In the program, the system checks whether the user leaves the screen without having saved his or her changes. If so, the PROMPT_AND_SAVE routine sends a reminder, and gives the user the opportunity to save the changes. The flight can be unlocked by calling the UNLOCK_FLIGHT subroutine.

```
MODULE USER_COMMAND_0200 INPUT.
  CASE OK_CODE.
    WHEN 'SAVE'....
    WHEN 'EXIT'.
      CLEAR OK_CODE.
      IF OLD_SPFLI NE SPFLI.
        PERFORM PROMPT_AND_SAVE.
      ENDIF.
      PERFORM UNLOCK_FLIGHT.
      LEAVE TO SCREEN 0.
    WHEN 'BACK'....
```

Update Techniques

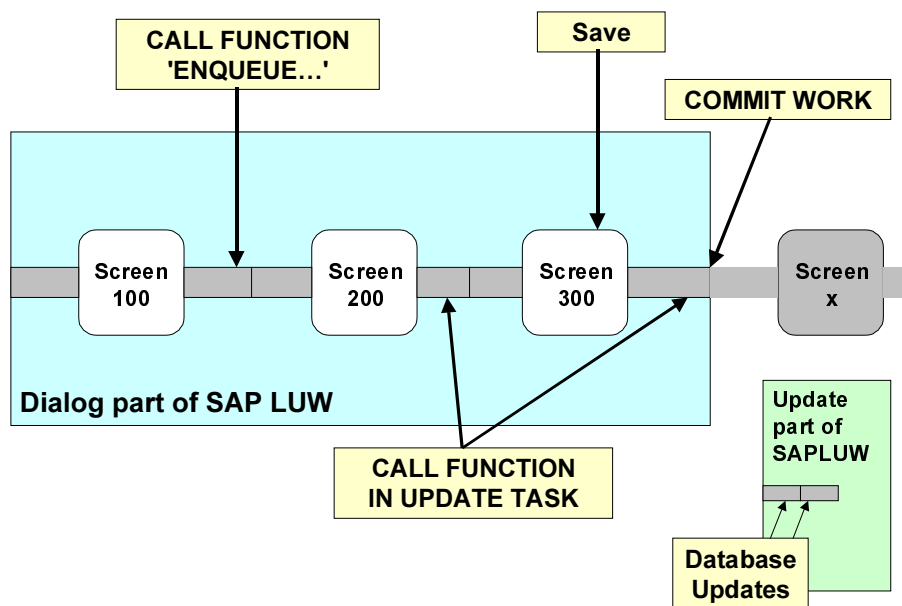
The main update technique for bundling database changes in a single database LUW is to use CALL FUNCTION... IN UPDATE TASK. This section describes various ways of updating the database.

A program can send an update request using COMMIT WORK

- To the update work process, where it is processed asynchronously. The program does not wait for the work process to finish the update ([Asynchronous Update \[Page 684\]](#)).
- For asynchronous processing in two steps ([Updating Asynchronously in Steps \[Page 687\]](#).)
- To the update work process, where it is processed synchronously. The program waits for the work process to finish the update ([Synchronous Update \[Page 688\]](#)).
- To its own work process locally. In this case, of course, the program has to wait until the update is finished ([Local Update \[Page 689\]](#).)

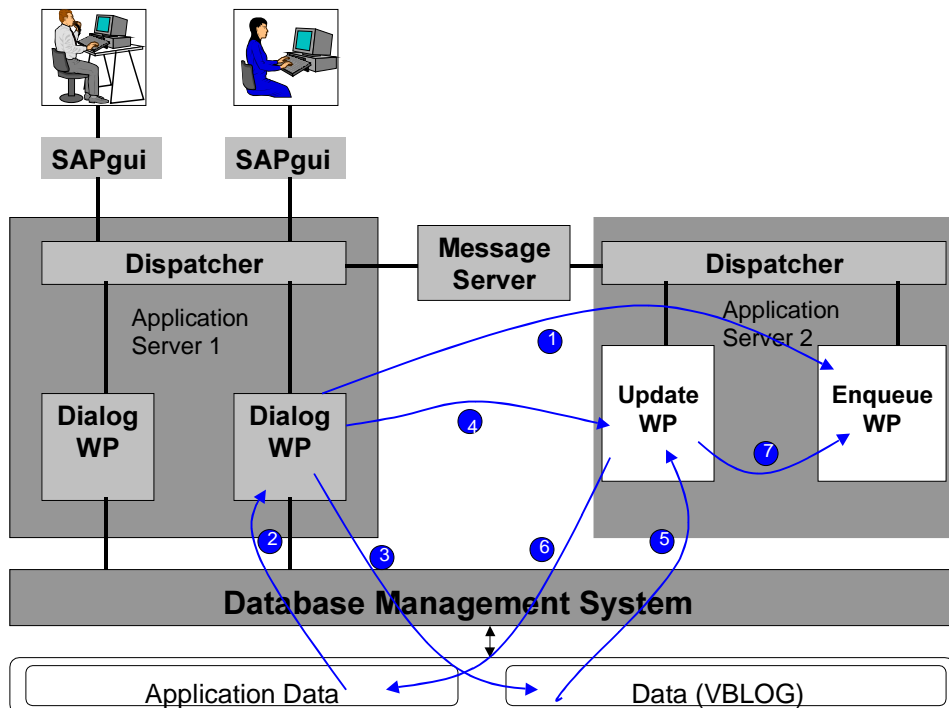
Asynchronous Update

A typical R/3 installation contains dialog work processes and at least one update work process. The update work processes are responsible for updating the database. When an ABAP program reaches a COMMIT WORK statement, any function modules from CALL FUNCTION... IN UPDATE TASK statements are released for processing in an update work process. The dialog process does not wait for the update to finish. This kind of update is called asynchronous update.



The following diagram shows a typical asynchronous update:

Asynchronous Update



For example, suppose a user wants to change an entry in a database table, or add a new one. He or she enters the necessary data, and then starts the update process by choosing **Save**. This starts the following procedure in the ABAP program:

1. Firstly, the program locks the database entry against other users, using the enqueue work process (or the message server in the case of a distributed system). This generates an entry in the lock table. The user is informed whether the update was successful, or whether the lock could not be set because of other users.
2. If the lock is set, the program reads the entry that is to be changed and modifies it. If the user has created a new entry, the program checks whether a record with the same key values already exists.
3. In the current dialog work process, the program calls a function module using `CALL FUNCTION... IN UPDATE TASK`, and this writes the change details as an entry in table VBLOG.
4. When the program is finished (maybe after further dialog steps), a `COMMIT WORK` statement starts the final part of the SAP LUW. The work process that is processing the current dialog step starts an update work process.
5. Based on the information passed to it from the dialog work process, the update work process reads the log entries belonging to the SAP LUW from table VBLOG.
6. The update work process passes this data to the database for updating, and analyzes the return message from the database. If the update was successful, the update work process triggers a database commit after the last database change and deletes the log entries from table VBLOG. If an error occurred, the update work process triggers a database rollback, leaves the log entries in table VBLOG, flags them as containing errors, and sends a SAPoffice message to the user, who should then inform the system administrator.
7. The corresponding entries in the lock table are reset by the update work process.

Asynchronous update is useful when response time from the transaction is critical, and the database updates themselves are so complex that they justify the extra system load of logging them in VBLOG. If you are running a transaction in a background work process, asynchronous update offers no advantages.

Updating Asynchronously in Steps

Updating Asynchronously in Steps

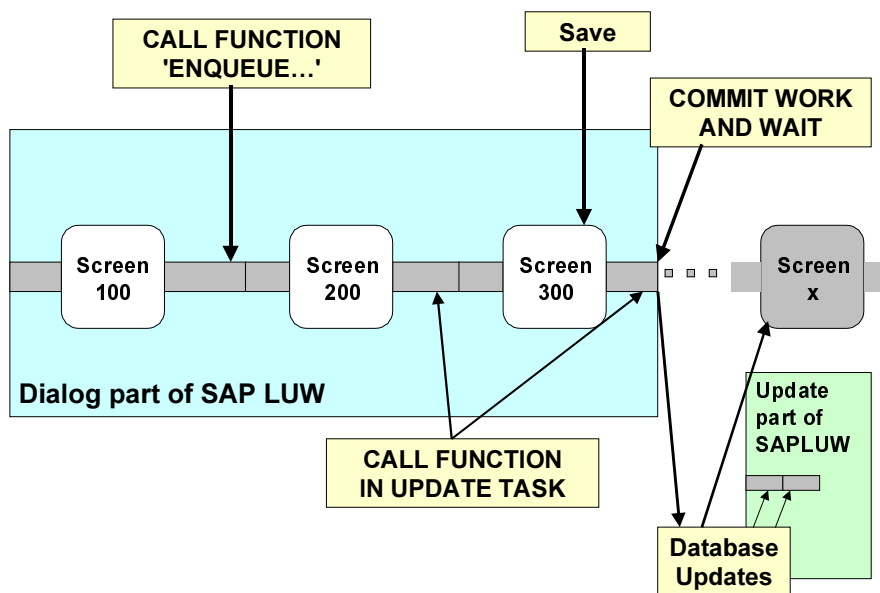
When you process a VBLOG entry asynchronously, you can do it in two update steps. This allows you to divide the contents of the update into primary and secondary steps. The primary step is called V1, the secondary step V2. The V1 and V2 steps of a log entry are processed in separate database LUWs. The entries in the lock table are usually deleted once the V1 step has been processed. There is no locking for the V2 step. Dividing up the update process allows you to separate time-critical updates that require database locks from less critical data updates that do not need locks. V2 steps receive lower priority from the dispatcher than V1 steps. This ensures that the time- and lock-critical updates are processed quickly, even when the system is busy.

If an error occurs during the V1 processing, the database rollback applies to all V1 steps in the log entry. The entire entry is replaced in table VBLOG. If an error occurs during V2 processing, all of the V2 steps in the log entry are replaced in table VBLOG, but the V1 updates are not reversed.

The system marks rolled-back function modules as error functions in the update task log. The error can then be corrected and the function restarted later. To access the update task log, choose *Tools* → *Administration* → *Monitoring* → *Update*. For further information about update administration, see the *Managing Updating* section of the *BC System Services* documentation.

Synchronous Update

In synchronous update, you do not submit an update request using `CALL FUNCTION... IN UPDATE TASK`. Instead, you use the ABAP statement `COMMIT WORK AND WAIT`. When the update is finished, control passes back to the program. Synchronous update works in the same way as bundling update requests in a subroutine (`PERFORM ON COMMIT`). This kind of update is useful when you want to use both asynchronous and synchronous processing without having to program the bundles in two separate ways. The following diagram illustrates the synchronous update process:



Use this type of update whenever the changed data is required immediately. For example, you may want to link SAP LUWs together where one LUW depends on the results of the previous one.

Local Update

Local Update

In a local update, the update program is run by the same work process that processed the request. The dialog user has to wait for the update to finish before entering further data. This kind of update is useful when you want to reduce the amount of access to the database. The disadvantage of local updates is their parallel nature. The updates can be processed by many different work processes, unlike asynchronous or synchronous update, where the update is serialized due to the fact that there are fewer update work processes (and maybe only one).

You switch to local update using the ABAP statement `SET UPDATE TASK LOCAL`. This statement sets a "local update switch". When it is set, the system interprets `CALL FUNCTION IN UPDATE TASK` as a request for local update. The update is processed in the same work process as the dialog step containing the `COMMIT WORK`. The transaction waits for the update to finish before continuing.

As an example, suppose you have a program that uses asynchronous update that you normally run in dialog mode. However, this time you want to run it in the background. Since the system response time is irrelevant when you are running the program in the background, and you only want the program to continue processing when the update has actually finished, you can set the `SET UPDATE TASK LOCAL` switch in the program. You can then use a system variable to check at runtime whether the program is currently running in the background.

By default, the local update switch is not set, and it is reset after each `COMMIT WORK` or `ROLLBACK WORK`. You therefore need to include a `SET UPDATE TASK LOCAL` statement at the beginning of each SAP LUW.

If you reset data within the local update, the `ROLLBACK WORK` statement applies to both the dialog and the update part of the transaction, since no new SAP LUW is started for the update.

Creating Update Function Modules

To create a function module, you first need to start the Function Builder. Choose *Tools* → *ABAP Workbench*, *Function Builder*. For more information about creating function modules, refer to the [ABAP Workbench Tools \[Ext.\]](#) documentation.

To be able to call a function module in an update work process, you must flag it in the Function Builder. When you create the function module, set the *Process Type* attribute to one of the following values:

- *Update with immediate start*
Set this option for high priority ("V1") functions that run in a shared (SAP LUW). These functions can be restarted by the update task in case of errors.
- *Update w. imm. start, no restart*
Set this option for high priority ("V1") functions that run in a shared (SAP LUW). These functions may not be restarted by the update task.
- *Update with delayed start*
Set this option for low priority ("V2") functions that run in their own update transactions. These functions can be restarted by the update task in case of errors.

To display the attributes screen in the Function Builder, choose *Goto* → *Administration*.

Defining the Interface

Function modules that run in the update task have a limited interface:

- Result parameters or exceptions are not allowed since update-task function modules cannot report on their results.
- You must specify input parameters and tables with reference fields or reference structures defined in the ABAP Dictionary.

Calling Update Functions

Synchronous or Asynchronous Processing?

Function modules that run in the update task can run synchronously or asynchronously. You determine this by the form of the commit statement you use:

- **COMMIT WORK**

This is the standard form, which specifies asynchronous processing. Your program does not wait for the requested functions to finish processing.

- **COMMIT WORK AND WAIT**

This form specifies synchronous processing. The commit statement waits for the requested functions to finish processing. Control returns to your program after all high priority (V1) function modules have run successfully.

The AND WAIT form is convenient for switching old programs to synchronous processing without having to re-write the code. Functionally, using AND WAIT for update-task updates is just the same as dialog-task updates with **PERFORM ON COMMIT**.

Parameter Values at Execution

In ABAP, you can call update-task function modules in two different ways. The way you choose determines what parameter values are used when the function module is actually executed. Parameter values can be set either at the time of the **CALL FUNCTION** statement, or at the time of the **COMMIT WORK**. The following sections explain.

[Calling Update Functions Directly \[Page 692\]](#)

[Adding Update Task Calls to a Subroutine \[Page 693\]](#).

The examples in these sections show asynchronous commits with **COMMIT WORK**.

Calling Update Functions Directly

To call a function module directly, use `CALL FUNCTION IN UPDATE TASK` directly in your code.

`CALL FUNCTION 'FUNCTMOD' IN UPDATE TASK EXPORTING...`

The system then logs your request and executes the function module when the next `COMMIT WORK` statement is reached. The parameter values used to execute the function module are those current at the time of the call.

```

a = 1.
CALL FUNCTION 'UPD_FM' IN UPDATE TASK EXPORTING PAR = A...
a = 2.
CALL FUNCTION 'UPD_FM' IN UPDATE TASK EXPORTING PAR = A...
a = 3.
COMMIT WORK.
```

Here, the function module `UPD_FM` is performed twice in the update task: the first time, with value 1 in `PAR`, the second time with value 2 in `PAR`.

Adding Update Task Calls to a Subroutine

Adding Update Task Calls to a Subroutine

You can also put the CALL FUNCTION IN UPDATE TASK into a subroutine and call the subroutine with:

```
PERFORM SUBROUT ON COMMIT.
```

If you choose this method, the subroutine is executed at the commit. Thus the request to run the function in the update task is also logged during commit processing. As a result, the parameter values logged with the request are those current at the time of the commit.

```

a = 1.
PERFORM F ON COMMIT.
a = 2.
PERFORM F ON COMMIT.
a = 3.
COMMIT WORK.

FORM f.
  CALL FUNCTION 'UPD_FM' IN UPDATE TASK EXPORTING PAR = A.
ENDFORM.
```

In this example, the function module UPD_FM is carried out with the value 3 in PAR. The update task executes the function module only once, despite the two PERFORM ON COMMIT statements. This is because a given function module, logged with the same parameter values, can never be executed more than once in the update task. The subroutine itself, containing the function module call, may not have parameters.

The method described here is not suitable for use inside dialog module code. However, if you do need to use a dialog module, refer to [Dialog Modules That Call Update Modules \[Ext.\]](#).

Special LUW Considerations

In the update-task queue, the system identifies all function modules belonging to the same [update transaction \[Ext.\]](#) (SAP LUW) by assigning them a common update key. At the next COMMIT WORK, the update task reads the queue and processes all requests with the predefined update key.

If your program calls an update-task function module, the request to execute the module (or the subroutine calling it) is provided with the update key of the current LUW and placed in the queue.

What happens with LUW's when update-task functions calls are embedded in modules (transactions or dialog modules) called by other programs? The following sections explain.

[Transactions that Call Update-Task Functions \[Page 695\]](#)

[Dialog Modules that Call Update-Task Functions \[Page 696\]](#)

Transactions that Call Update-Task Functions

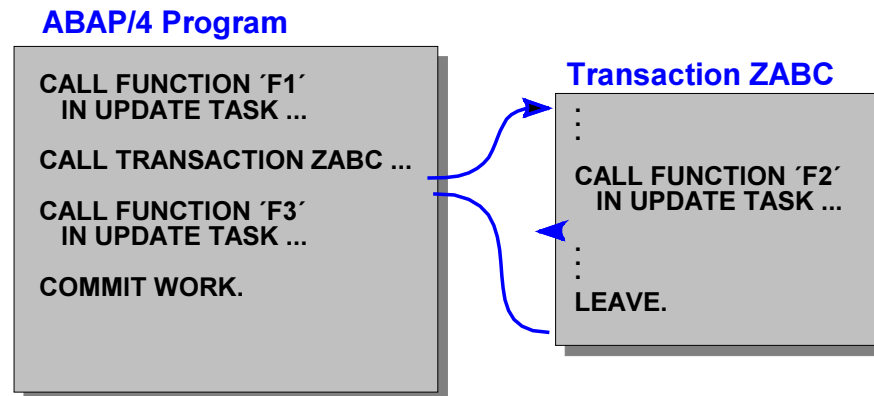
Transactions that Call Update-Task Functions

If your program calls another program that itself calls an update-task function module, you should be aware of the following:

Every called program begins a new [update transaction \[Ext.\]](#) (SAP LUW), and with it, a new update key. This key is used to identify all update-task operations requested during the called transaction (or report).

When returning from the program, the LUW of the calling program is restored together with the old update key.

If the called program does not contain its own COMMIT WORK, neither the requested database changes nor the calls to update-task function modules are performed. For example, in the following code, F1, F2, and F3 are update-task function modules:



Here, F1 and F3 are executed in the update task, because the COMMIT WORK for the main program triggers their execution. However, since transaction ZABC contains no COMMIT WORK statement, the function F2 is never executed by the update task.

Dialog Modules that Call Update-Task Functions

Unlike transactions and executable programs (reports), dialog modules do not start a new [update transaction \[Ext.\]](#). Calls to update-task function modules from a dialog module use the same update key as the ones in the calling program. The result is that calls to update-task function modules from a dialog module are executed only if a COMMIT WORK statement occurs in the calling program.

If you place a COMMIT WORK in a dialog module, it does commit changes to the database (for example, with UPDATE). However, it does not start the update task. If the dialog module calls update-task function modules, the function modules are not triggered until a COMMIT WORK in the calling program is reached.

If you are using dialog modules, try to avoid including calls to update-task function modules in subroutines called with PERFORM ON COMMIT. In general, any occurrence of PERFORM ON COMMIT (with or without update-task function calls) in a dialog module can be problematic.

The reason for this is that dialog modules run in their own roll area, and this roll area disappears when the module finishes. This means that all local data (including data used as parameter values when calling an update-task function module) disappears as soon as the commit in the main program is reached.

If you must use this method in a dialog module (i.e. include the call to an update-task function in a subroutine), you must ensure that the values of the actual parameters still exist when the update-task function actually runs. To do this, you can store the required values with EXPORT TO MEMORY and then import them back into the main program (IMPORT FROM MEMORY) before the COMMIT WORK statement.

Error Handling for Bundled Updates

Error Handling for Bundled Updates

Runtime errors can occur during execution of bundled updates. How are they handled? In general, COMMIT WORK processing occurs in the following order:

1. All dialog-task FORM routines logged with PERFORM ON COMMIT are executed.
2. All high-priority (V1) update-task function modules are executed.
The end of V1-update processing marks the end of the [update transaction \[Ext.\]](#). If you used COMMIT WORK AND WAIT to trigger commit processing, control returns to the dialog-task program.
3. All low-priority (V2) update-task function modules are triggered.
All background-task function modules are triggered.

Runtime errors can occur either in the system itself, or because your program issues an abend message (MESSAGE type 'A'). Also, the ROLLBACK WORK statement automatically signals a runtime error. The system handles errors according to where they occur:

- **in a FORM routine** (called with PERFORM ON COMMIT)
 - Updates already executed for the current update transaction are rolled back.
 - No other FORM routines will be started.
 - No further update-task or background-task functions will be started.
 - An error message appears on the screen.
- **in a V1 update-task function module** (requested IN UPDATE TASK)
 - Updates already executed for V1 functions are rolled back.
 - All further update-task requests (V1 or V2) are thrown away.
 - All background-task requests are thrown away.
 - Updates already executed for FORM routines called with PERFORM ON COMMIT are **not** rolled back.
 - An error message appears on the screen, if your system is set up to send them.
- **in a V2 update-task function module** (requested IN UPDATE TASK)
 - Updates already executed for the current V2 function are rolled back.
 - All update-task requests (V2) still to be executed are carried out.
 - All background-task requests still to be executed are carried out.
 - No updates for previously executed V1 or V2 function are rolled back.
 - No updates previously executed for FORM routines (called with ON COMMIT) are rolled back.
 - An error message appears on the screen, if your system is set up to send them
- **in a background-task function module** (requested IN BACKGROUND TASK DESTINATION)
 - Background-task updates already executed for the current DESTINATION are not rolled back.

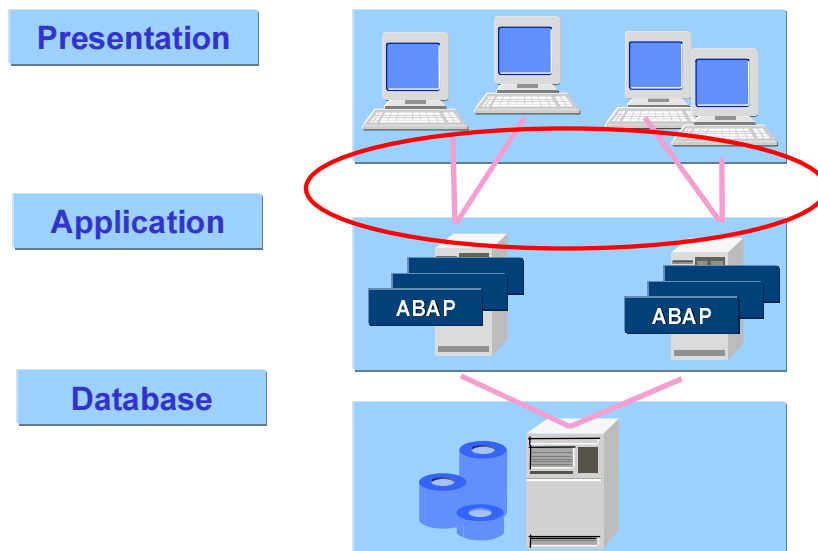
Error Handling for Bundled Updates

- All further background-task requests for the same DESTINATION are thrown away.
- No other previously-executed updates are not rolled back.
- No error message appears on the screen.

If your program detects that an error in remote processing has occurred, it can decide whether to resubmit the requests at a later time.

For information about RFC programming, see [Remote Communications \[Ext.\]](#).

ABAP User Interfaces



Screens

Processing User Input on a Screen

Processing User Input on a Screen

The Screen Painter allows you to place various elements on a dialog screen which allow users to make entries.

On a dialog screen, the user makes requests by selecting screen pushbuttons, menu functions, function keys, tool-bar pushbuttons, icons or the ENTER key.

How does a dialog program handle user requests?

When an action is performed, the system triggers the PROCESS AFTER INPUT event. The data passed includes field screen data entered by the user and a function code. A function code is a technical name that has been allocated in the Screen Painter or Menu Painter to a menu entry, a pushbutton, the ENTER key or a function key of a screen. An internal work field (ok-code) in the PAI module evaluates the function code, and the appropriate action is taken.

The following sections use the TZ20 transaction (development class SDWA) to show how you can use pushbuttons to control the way a transaction executes.

The final two sections of this chapter explain how radio buttons and check boxes can be used.

The following topics explain how you can include user functions in your transaction:

[Programming with Function codes \[Page 702\]](#)

[Programming with Radio Buttons \[Page 710\]](#)

[Programming with Check Boxes \[Page 711\]](#)

Programming with Function codes

The following topics guide you in using function codes:

[Setting up Function codes \[Page 703\]](#)

[Handling Function codes \[Page 706\]](#)

[Handling Field Selections \[Page 708\]](#)

[Sharing GUI Statuses \[Page 709\]](#)

Setting up Function codes

Setting up Function codes

To handle user requests in a dialog program, you must assign function codes to the relevant screen and window elements in the Screen Painter or the Menu Painter.

- In the Screen Painter:
 - screen pushbuttons
- In the Menu Painter:
 - menu functions
 - function keys
 - tool-bar pushbuttons and icons
 - the ENTER key

Screen Painter

In the Screen Painter, you can assign a function code by setting the *FctCode* attribute for a pushbutton field.

The function code (FctCode) FTCH has been set in the Screen Painter for the *Display* pushbutton in the TZ20 transaction.

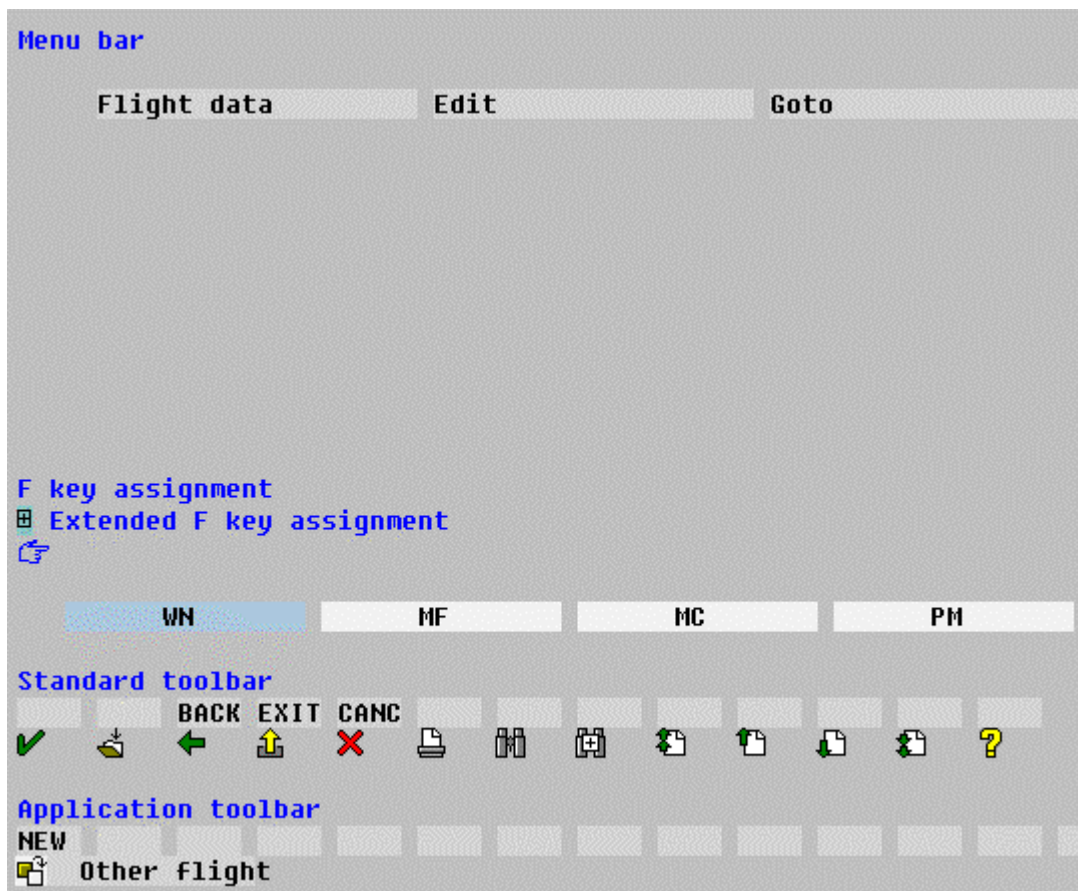
Screen Field Attributes							
Field type	Pushbutton						
Field name	%_AUTOTEXT002						
Field text	Display						
<input checked="" type="checkbox"/> With icon	Icon name		ICON_DISPLAY				
<input type="checkbox"/> Scroll.	Quick info		Display				
Line	3	Column	30	Length	19	Vis.len.	17
Groups						Height	1
FctCode	FTCH	FctType		LoopType		LoopDisp	0

Menu Painter

In the SAP system, the user interface of a program consists of one or more GUI status and a GUI title that you define in the Menu Painter. A GUI status is a set of dynpro elements needed for the transaction at a given time. A GUI title is the title bar text that appears at the top of a Windows session.

In order to be able to process the function code of a screen element that you define in the Menu Painter the program must specify the GUI status that should appear with the screen at PBO.

In the TZ20 transaction the *Other Flight* pushbutton is defined as a part of the menu bar in the TD0100 GUI status. The *Other Flight* pushbutton has the function code NEW in the Menu Painter.



You can set the GUI status in a dialog program with the key word SET PF-STATUS:

```
SET PF-STATUS <GUI_status>.
```

The <GUI_status> is a character string up to 8 characters and can be either a literal (in single quotes) or a variable.

You set the title of a screen or a dialog box with the key word SET TITLEBAR as follows:

```
SET TITLEBAR <title> WITH <p1> <p2> <p3> <p4>.
```

This statement can accept up to four parameter values. You enter the corresponding placeholders (using a &) in the title definition. For example, in a *Maintain Table* screen, the title might be defined as follows:

```
SET TITLEBAR 'ABC' WITH 'Customer'.
```

If the title 'ABC' is:

```
Maintain Table &
```

then the window title will be as follows:

```
Maintain Table Customer
```

The system fills & at runtime with the name of the specified table.

Setting up Function codes

If you do not set a user interface for a screen, the system displays that screen with the same GUI elements that were already set for the previous screen. If the transaction does not have a previous screen, or if you have not set any GUI status yet, the screen will be issued without a user interface.

For an example of how to program your PBO module in the flow logic of the corresponding screen, look at transaction TZ20. The single PBO module sets the GUI interface and title of the transaction. The TD0100 GUI status contains the definition of the *Other Flight* pushbutton.

```
MODULE STATUS_0100 OUTPUT.  
  SET PF-STATUS 'TD0100'.  
  SET TITLEBAR '100'.  
ENDMODULE.
```

For more information about defining a GUI status and titles, see [BC ABAP Workbench Tools \[Ext.\]](#).

Handling Function codes

When the user selects a function in a transaction, the system copies the function code into a specially designated work field called OK_CODE. This field is global in the ABAP module pool. The OK_CODE can then be evaluated in the corresponding PAI module.

The function code is always passed in exactly the same way, regardless of whether it comes from a screen's pushbutton, a menu option, function key or other GUI element.

In the Screen Painter, if you display the field list for a screen, the OK_CODE field is always the last field in the list. This field is initially nameless, and has field type *OK*. You can give this field any name, but it is traditionally called the OK_CODE. Whatever you name it, you must also include a definition for it as a global field in your module pool:

```
PROGRAM SAPMTZ20.
```

```
...
DATA: OK_CODE(4).
```

The program declaration for an OK_CODE is *four* characters long.

For an example of how to handle the function code, look at transaction TZ20.

```
*-----*
*   PAI Modules MTZ20I01                               *
*&-----*
*&       Module  USER_COMMAND_0100  INPUT              *
*&-----*
MODULE USER_COMMAND_0100 INPUT.
  CASE OK_CODE.
    WHEN 'FTCH'.
      SELECT SINGLE * FROM SPFLI WHERE CARRID = SPFLI-CARRID
                                     AND CONNID =
SPFLI-CONNID.
      IF SY-SUBRC NE 0. CLEAR SPFLI. ENDIF.
      CLEAR OK_CODE.
    WHEN 'NEW'.
      CLEAR: SPFLI, OK_CODE.
    WHEN 'CANC'.
      CLEAR OK_CODE.
      SET SCREEN 0. LEAVE SCREEN.
    WHEN 'EXIT'.
      CLEAR OK_CODE.
      SET SCREEN 0. LEAVE SCREEN.
    WHEN 'BACK'.
      CLEAR OK_CODE.
      SET SCREEN 0. LEAVE SCREEN.
  ENDCASE.
ENDMODULE.
```

For example, if the user selects the *Display* pushbutton, the FTCH function code is copied into the OK_CODE internal work field. The OK_CODE is then checked in the PAI module and, if it contains FTCH, a SELECT is executed in order to fetch the data to be displayed.

The SET SCREEN and LEAVE SCREEN statements control screen flow. The specification SET SCREEN 0 tells the system to go back a whole call level. In this small program, this means

Handling Function codes

exiting from the transaction altogether. Calling levels are described in [Controlling the Screen Flow \[Page 712\]](#).

After processing the function code, delete the contents of the field OK_CODE to avoid any unwanted function selection.

The function code currently active in a program can also be ascertained from the SY-UCOMM variable.

Handling Field Selections

Transactions often let users select a field to request a function. In some cases, field selection alone can trigger a function. At others, the user clicks on a field and then selects a menu option, function key or other button.

To get notification of field selection for your program:

- Field selection alone

When you provide the F2 key with a function code, the system passes the code to your program when the user selects a field. In this respect, double-clicking a field and single-clicking it with F2 are equivalent. The system responds to both selections by triggering PAI for the F2 function code.

- Field selection plus function request

Provide the screen elements with function codes, as usual. When the user selects or presses the element, the function code is passed in to the program.

When your program receives the relevant function code, it first needs to know what field has been selected. Use the GET CURSOR command:

```
GET CURSOR FIELD <field name>.
```

So you can find out what field the cursor is sitting in. The system sets the variable <field name> to the name of the screen field where the cursor is currently positioned.

Note that GET CURSOR returns an empty field-name parameter, if the user double-clicks on an area that is not a field.

When processing step-loops, use the LINE parameter to find out which loop block line contains the cursor.

```
GET CURSOR FIELD <field name> LINE <field2>.
```

The variable <field2> contains the number of the loop block line containing the cursor.

Field selection plus function request:

```
GET CURSOR FIELD selfield.  
IF SELFELD NOT SPACE.  
  CASE OK_CODE.  
    WHEN 'SELE'. PERFORM DISPLAY_FIELD_INFO USING SELFELD.  
    WHEN 'CHNG'. PERFORM MODIFY_FIELD USING SELFELD.  
    WHEN 'DELE'. PERFORM DELETE_FIELD USING SELFELD.  
  ENDCASE.  
  CLEAR OK_CODE.  
ENDIF.
```

Sharing GUI Statuses

Sharing GUI Statuses

In the ABAP Development Workbench, screens and user interfaces are independent of one another. You can, for instance, use the same interface for several screens. Some screens, however, might have fewer active functions than other screens. Instead of creating a new GUI-status for these screens, you can use the same status and deactivate one or more of its functions.

The system offers you an additional keyword to deactivate specific menu functions in a GUI-status. The format of this statement is:

```
SET PF-STATUS <GUI status> EXCLUDING <function codes>.
```

If you want to deactivate a single function, the <function codes> parameter is a single type C field. Enter the name of the appropriate function in quotes. To deactivate several functions, place these function codes in an internal table. The internal table must have the following structure:

```
DATA: BEGIN OF INTTAB OCCURS 20,  
      FUNCTION (4),  
      END OF INTTAB.
```

Specifying functions with EXCLUDING deactivates all the relevant GUI-items (menu item, pushbutton and function code) assigned the given code. Deactivated pushbuttons do not appear at all. Deactivated menu functions appear (as do function key assignments in the list displayed with the right mouse button), but only in grayed-out form. If the user selects them, nothing happens.

Programming with Radio Buttons

Radio buttons are simple input fields. They do not have associated function codes, and thus do not themselves trigger a PAI event. However, there are some special things to know about using them to interpret user requests.

Setting Up Radio Buttons

When you add radio buttons to a screen, the Screen Painter automatically creates a one-character screen field for the button. You should declare a corresponding one-character variable for the radio buttons in your ABAP module.

```
DATA: RADIO1, RADIO2, RADIO3.
```

By default, the system sets the first radio button in the group to on.

Checking the Radio Button Selection

Radio buttons are exclusive selection buttons that belong to a logical group. If the user clicks on one, all the other buttons in the group are automatically deselected by the system. If a radio button is set then it has the value 'X'. You do not have to program the deselection of radio buttons: the screen interface does it for you.

At the relevant PAI event, your check can assume that only one button is on at a time.

```
IF RADIO1 NE SPACE:  
  PERFORM PROCESS_RADIO1.  
ELSEIF RADIO2 NE SPACE:  
  PERFORM PROCESS_RADIO2.  
ELSE.  
  PERFORM PROCESS_RADIO3.  
ENDIF.
```

Setting Radio Button Values from the Program

When the user clicks on a radio button, the screen interface turns off all the other buttons in the group.

However, if you want to set radio buttons yourself (from the module pool), the screen interface is not available to deselect the rest. You must program this deselection yourself.

```
RADIO1 = SPACE.  
RADIO2 = 'X'.  
RADIO3 = SPACE.
```

Programming with Check Boxes

Check boxes are non-exclusive selection buttons. The user can turn on more than one at a time.

When you add check boxes to a screen, the Screen Painter automatically creates a one-character screen field for each box. You should declare a corresponding one-character variable for the boxes in your ABAP module.

```
DATA: CHECK1, CHECK2.
```

By default, all check boxes are initialized to off. If you want to initialize them to on, assign them a value:

```
CHECK1 = 'X'.  
CHECK2 = 'X'.
```

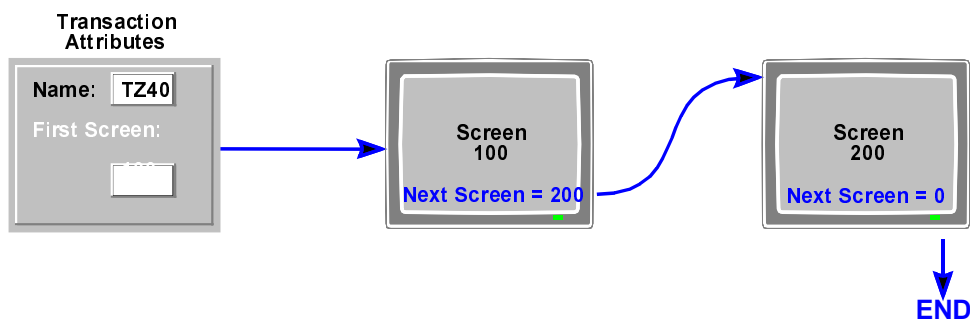
To check which boxes were selected by the user, you can query which boxes have a value not equal to space.

Controlling the Screen Flow

For the user, a transaction is a series of screens that appear one after another. In the transaction program, screens are chained together by a series of “next-screen” numbers. When you define the transaction, you specify the number of the first screen. Then, for each screen in the transaction, you can specify a “next screen” statically or dynamically:

- Static screen pointers

When you define a screen, you specify a *Next screen* attribute for it. This attribute gives the name of the screen that is to follow the current screen by default. However, the static attribute is overridden whenever a “next-screen” is set dynamically.



Static Screen Chain

- Dynamic screen sequence

Any screen can set its own “next screen” as part of screen processing. The ABAP commands for doing this are SET SCREEN and CALL SCREEN. When you set screens dynamically, you can string them together one after the other (as in a chain), or insert groups of them into the current chain.

The following topics provide information on handling screens in a transaction:

[Introduction to Screen Flow Control \[Page 713\]](#)

[Setting the Next Screen \[Page 716\]](#)

[Calling a New Screen Sequence \[Page 717\]](#)

[Leaving the Current Screen \[Page 719\]](#)

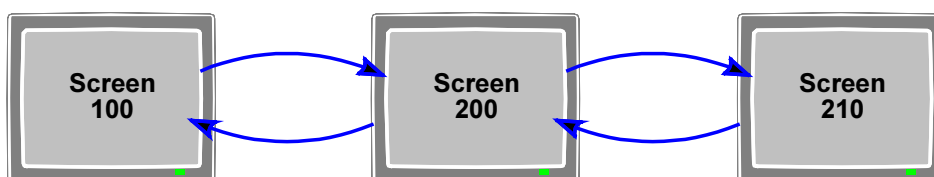
[Example Transaction: Setting and Calling Screens \[Page 720\]](#)

[Processing Screens in the Background \[Page 722\]](#)

Introduction to Screen Flow Control

As an example of controlling screen flow in a transaction, look at transaction TZ40. (This transaction is in development class SDWA and is delivered with the system.) TZ40 lets users display flight information and enter updates into the display.

TZ40 uses two screens and a dialog box (popup window) for getting user updates. The transaction always displays the first two screens (numbers 100 and 200). The third (210) however only appears under certain conditions. The possible flow of screens looks as follows:

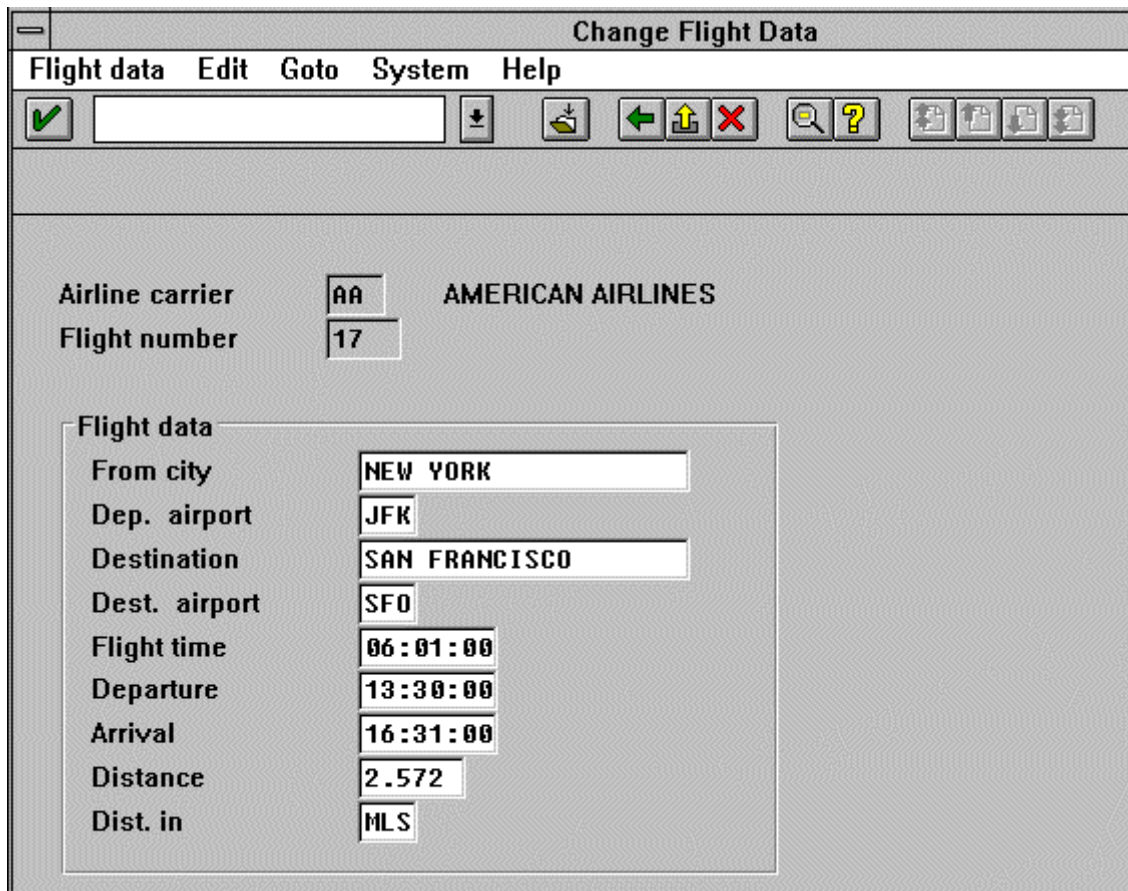


Possible Screen Flow

In practice, the user sees the following sequence:

- **Screen 100:** The user enters flight information and presses ENTER to request a display of flight details.

- **Screen 200:** The system displays complete details about the flight, in update mode. The user types over the display to enter the changes.



Change Flight Data

Flight data Edit Goto System Help

Airline carrier AA AMERICAN AIRLINES

Flight number 17

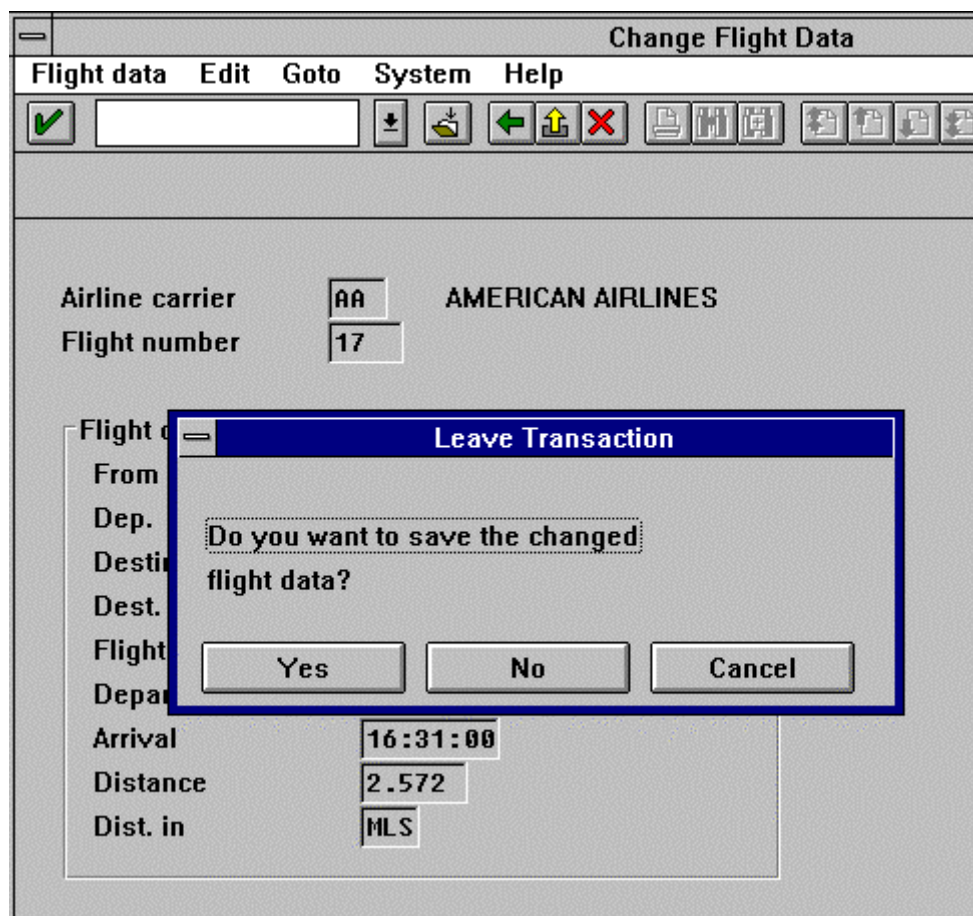
Flight data

From city	NEW YORK
Dep. airport	JFK
Destination	SAN FRANCISCO
Dest. airport	SFO
Flight time	06:01:00
Departure	13:30:00
Arrival	16:31:00
Distance	2.572
Dist. in	MLS

- **Screen 210**

Screen 210 appears only if the user tries to exit screen 200 without saving. The popup reminds the user to save the changes or cancel them (by specifying Yes or No).

Introduction to Screen Flow Control



To make this sequence of screens possible, transaction TZ40 must be able to call the dialog box screen conditionally.

An ABAP module can “branch to” or “call” the next screen. The difference lies in where you want control to go after processing the next screen. The relevant ABAP commands are:

```
SET SCREEN <screen-number>.
CALL SCREEN <screen-number>.
LEAVE SCREEN.
LEAVE TO SCREEN <screen-number>.
```

With SET SCREEN, the current screen simply specifies the next screen in the chain. Control branches to this next screen as soon as the current screen has been processed. Return from next screen to current screen is not automatic.

With CALL SCREEN, the current (calling) chain is suspended, and a next screen (or screen chain) is called in. The called screen can then return to the suspended chain with the statement LEAVE SCREEN TO SCREEN 0.

For complete information, see:

[Setting the Next Screen \[Page 716\]](#)

[Calling a New Screen Sequence \[Page 717\]](#)

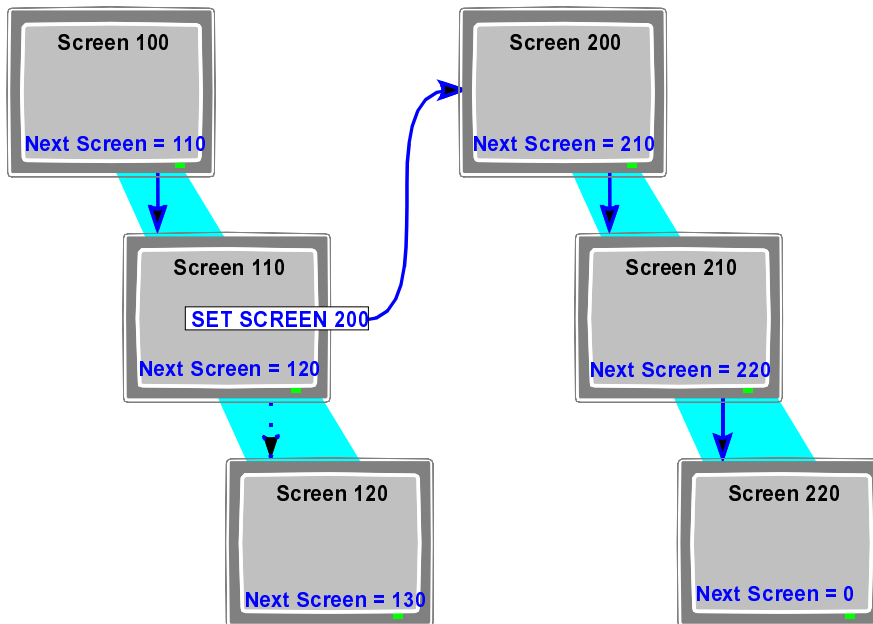
[Leaving the Current Screen \[Page 719\]](#)

Setting the Next Screen

Every screen has a static *Next screen* attribute that specifies the next screen to follow the current one. You can override this specification by using the SET SCREEN statement:

SET SCREEN <screen number>.

SET SCREEN tells the system to ignore the statically defined *Next screen* and use <screen number> as the next screen instead.



SET SCREEN

This override is temporary and has no effect on the attribute values stored in the Screen Painter.

A SET SCREEN statement merely specifies the next screen: it does not interrupt processing of the current screen. If you want to branch to the next screen without finishing the current one, use LEAVE SCREEN.

Note that you can specify the next-screen number with a variable:

```

DATA: REQSCRN LIKE SY-DYNNR VALUE '100'.
MODULE SET_NEXT_SCREEN.
  SET SCREEN REQSCRN.
ENDMODULE.
  
```

The system field SY-DYNNR always contains the number of the current screen.

Calling a New Screen Sequence

Calling a New Screen Sequence

Sometimes you want to insert a screen, or a whole sequence of screens, into the course of a transaction. For instance, you might want to let an user call a popup screen from the main application screen to let them enter secondary information. After they have completed their entries, the users should be able to close the popup and return directly to the place where they left off in the main screen. There are two methods for doing this:

- use the CALL SCREEN statement

The CALL SCREEN statement lets you insert such a sequence into the current one. Using this statement is described here.

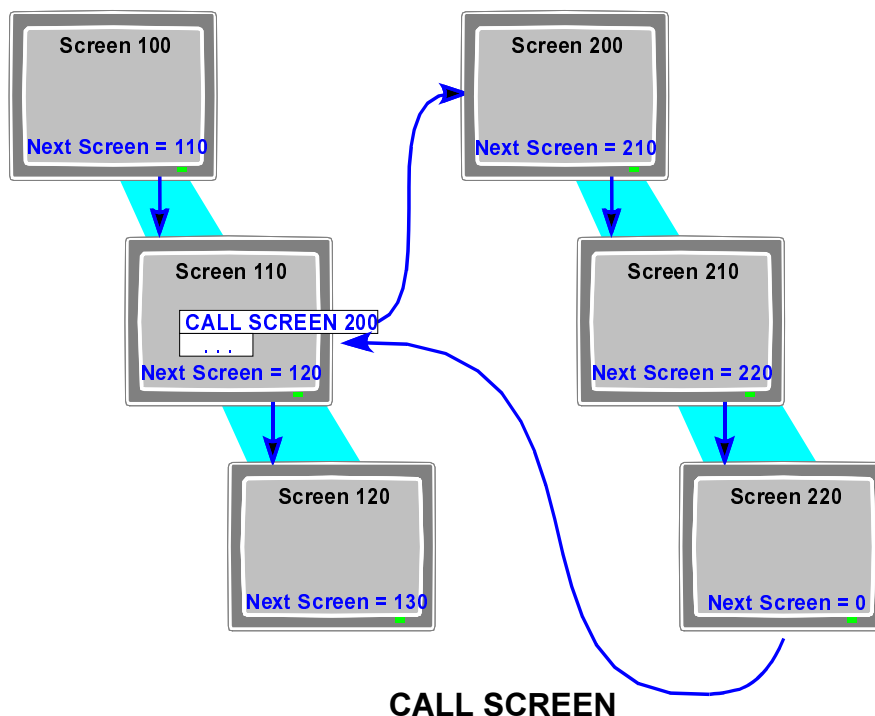
- call a dialog module

A dialog module is a callable sequence of screens that does not belong to a particular transaction. Dialog modules have their own module pools, and can be called by any transaction. For information on using dialog modules, see [Calling Dialog Modules](#) [Page 1331].

The syntax for calling up a new screen sequence is:

CALL SCREEN <screen number>.

You can think of CALL SCREEN as “stacking” a sequence, since the statement actually suspends the current sequence and starts a new one. The system continues with the new sequence until it is finished, at which point the suspended sequence is resumed. (Processing resumes with the statement directly after the CALL SCREEN.)



To call a screen as a dialog box (popup), use CALL SCREEN with the options STARTING AT, ENDING AT:

Calling a New Screen Sequence

```
CALL SCREEN <screen number>
STARTING AT <start column> <start line>
ENDING AT <end column> <end line>
```

The STARTING AT and ENDING AT options tell the system where to position the popup screen. The screen itself must be smaller than a regular screen.

In the ABAP world, each stackable sequence of screens is a "call mode". This is important because of the way you return from a given current sequence. To terminate a call mode and return to a suspended chain, set the "next screen" to 0 and leave to it:

LEAVE TO SCREEN 0.	or	SET SCREEN 0.
		LEAVE SCREEN.

When you return to the suspended chain, execution resumes with the statement directly following the original CALL SCREEN statement.

The original sequence of screens in a transaction is itself a calling mode. If you LEAVE TO SCREEN 0 in this sequence (that is, without having stacked any additional call modes), you return from the transaction altogether.

You can have up to 9 calling modes stacked at one time.

Leaving the Current Screen

Leaving the Current Screen

To discontinue processing for the current screen, use:

```
LEAVE TO SCREEN <screen number>.
```

or

```
SET SCREEN <number>.  
LEAVE SCREEN.
```

Both of these statements terminate processing for the current screen and go directly to <screen number>. If you use SET SCREEN without LEAVE SCREEN, the program finishes processing for the current screen before branching to <screen number>.

If you use LEAVE SCREEN without a SET SCREEN before it, you terminate the current screen and branch directly to the screen specified as the default next-screen in the screen attributes.

In “calling mode”, the special screen number 0 (LEAVE TO SCREEN 0) causes the system to jump back to the previous call level. That is, if you have called a screen sequence with CALL SCREEN, leaving to screen 0 terminates the sequence and returns to the calling screen. If you have not called a screen sequence, LEAVE TO SCREEN 0 terminates the transaction. For information on CALL SCREEN, see [Calling a New Screen Sequence \[Page 717\]](#).

Example Transaction: Setting and Calling Screens

To see a complete implementation of screen flow control, it is useful to look at how transaction TZ40 (development class SDWA) is organized.

Screen Flow Logic

To demonstrate how a transaction branches to or calls a screen, look at processing for screen 200. The handling of the exit commands (function codes BACK and EXIT) shows how. When handling a BACK or EXIT function code, the PAI module must check whether flight details have changed since the screen display or last save. If so, screen 200 must call up the popup 210 to prompt about saving. The relevant parts of the screen 200's flow logic are:

```
*-----*
* Screen 200: Flow Logic                      *
*&-----*
PROCESS AFTER INPUT.
MODULE EXIT_0200 AT EXIT-COMMAND.
*      (...<Field checks here>...)
MODULE USER_COMMAND_0200.
```

ABAP Code

The PAI modules for screen 200 follow. Transaction TZ40 offers all the return functions (*Back*, *Exit* and *Cancel*) as exit-commands. In Screen 200, however, only the Cancel function allows immediate exit from the screen. To effect a cancel, standard exit logic is used to tell the system to go back to screen 100:

```
*&-----*
*&   Module EXIT_0200 INPUT
*&-----*

MODULE EXIT_0200 INPUT.
CASE OK_CODE.
  WHEN 'CANC'.
    CLEAR OK_CODE.
    SET SCREEN 100.
    LEAVE SCREEN.
ENDCASE.
ENDMODULE.
```

All other function codes for screen 200 are handled as follows:

- The SAVE function triggers an update of the database.
- The EXIT and BACK functions trigger calls to the SAFETY_CHECK routine. This routine checks for unsaved data in the screen, and reminds the user to save if necessary.

Note the return technique. For the EXIT function, control returns from the transaction altogether (SET SCREEN 0). For the BACK function, the previous screen is set as the following screen. (SET SCREEN 100).

```
*&-----*
*&   Module USER_COMMAND_0200 INPUT
```


Example Transaction: Setting and Calling Screens

```

*&-----*
MODULE USER_COMMAND_0200 INPUT.
CASE OK_CODE.
  WHEN 'SAVE'.
    UPDATE SPFLI.
    IF SY-SUBRC = 0.
      MESSAGE S001 WITH SPFLI-CARRID SPFLI-CONNID.
    ELSE.
      MESSAGE A002 WITH SPFLI-CARRID SPFLI-CONNID.
    ENDIF.
  CLEAR OK_CODE.
  WHEN 'EXIT'.
    CLEAR OK_CODE.
    PERFORM SAFETY_CHECK USING RCODE.
    IF RCODE = 'EXIT'. SET SCREEN 0. LEAVE SCREEN. ENDIF.
  WHEN 'BACK'.
    CLEAR OK_CODE.
    PERFORM SAFETY_CHECK USING RCODE.
    IF RCODE = 'EXIT'. SET SCREEN 100. LEAVE SCREEN. ENDIF.
ENDCASE.
ENDMODULE.

```

Code for the SAFETY_CHECK routine follows. The CHECK statement compares current screen values to the saved screen values. If the values match, no save is needed, and the routine terminates.

If the values differ, SAFETY_CHECK calls the popup screen 210. The popup asks the user if he wants to save, and returns the answer (SAVE, EXIT and CANCEL) in the OK_CODE.

```

*-----*
* Subroutine SAFETY_CHECK          *
*-----*

FORM SAFETY_CHECK USING RCODE.
  LOCAL OK_CODE.
  RCODE = 'EXIT'.
  CHECK SPFLI NE OLD_SPFLI.
  CLEAR OK_CODE.
  CALL SCREEN 210 STARTING AT 10 5.
CASE OK_CODE.
  WHEN 'SAVE'. UPDATE SPFLI.
  WHEN 'EXIT'.
  WHEN 'CANC'. CLEAR SPFLI.
ENDCASE.
ENDFORM.

```

Processing Screens in the Background

You can suppress entire screens using SUPPRESS DIALOG. This command allows you to perform screen processing "in the background". The system carries out all PBO and PAI logic, but does not display the screen to the user.

Suppressing screens is useful when you are branching to list-mode from a transaction dialog step.

Use the SUPPRESS DIALOG command in the first module called from the screen's PBO logic. For example:

```
**** ABAP module processing for Screen 100
CALL SCREEN 110 STARTING AT 10 5

**** Screen 110 flow logic
PROCESS BEFORE OUTPUT.
MODULE DIALOG_WINDOW.

**** ABAP module processing
MODULE DIALOG_WINDOW OUTPUT.
SUPPRESS DIALOG.
LEAVE TO LIST-PROCESSING AND RETURN TO SCREEN 0.
WRITE: /
WRITE: /
ENDMODULE.
```

The SUPPRESS DIALOG statement lets you use the processing context for screen 110 as a framework for displaying the standard list output. If you don't use SUPPRESS DIALOG here, screen 110 is displayed, and is empty. When the user presses ENTER, the standard list output is displayed.

Modifying the Screen

Modifying the Screen

There is a number of different ways in which screens can be modified at runtime:

- Field attributes can be set in the dynpro
Temporary changes to field properties (e.g. input/output field, mandatory field) can be made from a dialog program. Fields can also be temporarily suppressed. Using this technique to modify screens dynamically often means you avoid the need to define additional screens.
- Field attributes can be changed with the help of the function field selection
The function field selection supports you in changing the attributes of screens dynamically.
- Displaying subscreens at runtime
A subscreen can be displayed in order to enhance an existing screen at runtime. A subscreen is used to selectively display certain fields. For example, you could have two subscreens, one containing *Material name* and *Material number* fields and the other containing *Customer name* and *Customer number* fields. One of these two subscreens is selected according to the input made by the user in the previous screen.
- Selecting a cursor position
The cursor can be positioned in a particular field on the screen from a dialog program, according to the user's entries.

You can find further information in the following sections:

[Setting Screen Field Attributes \[Page 724\]](#)

[Changing Screen Field Attributes with the Function Field Selection \[Page 727\]](#)

[Using Subscreens \[Page 743\]](#)

[Manipulating the Cursor \[Page 745\]](#)

Setting Screen Field Attributes

Every screen field has attributes that you set in the Screen Painter when you define the screen. At runtime, you may want to change these attributes, depending on what functions the user has requested in the previous screen. At runtime, attributes for each screen field are stored in a memory table called SCREEN. You do not need to declare this table in your program. The system maintains the table for you internally and updates it with every screen change.

The memory table SCREEN contains the following fields:

Name	Length	Description
NAME	30	Name of the screen field
GROUP1	3	Field belongs to field group 1
GROUP2	3	Field belongs to field group 2
GROUP3	3	Field belongs to field group 3
GROUP4	3	Field belongs to field group 4
ACTIVE	1	Field is visible and ready for input
REQUIRED	1	Field input is mandatory
INPUT	1	Field is ready for input
OUTPUT	1	Field is for display only
INTENSIFIED	1	Field is highlighted
INVISIBLE	1	Field is suppressed
LENGTH	1	Field output length is reduced
DISPLAY_3D	1	Field is displayed with 3D frames
VALUE_HELP	1	Field is displayed with value help

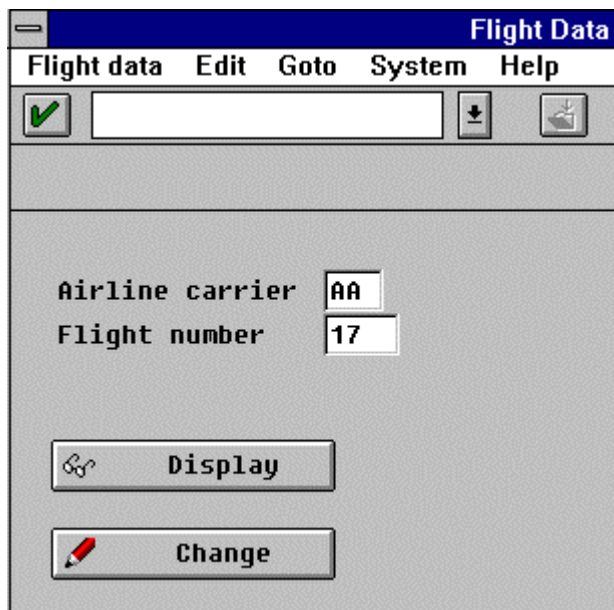
To activate a field attribute, set its value to 1. To deactivate it, set it to 0. When you set the ACTIVE attribute to 0, the system suppresses the field and turns off the ready for input attribute. The user can neither see the field nor enter values into it.

You can define values for each of these attributes in the *Attribs. for 1 field* section in the field list of the Screen Painter. If you need more information about attribute meanings, see [BC ABAP Workbench Tools \[Ext.\]](#).

As an example of modifying the screen dynamically, start with transaction tz50 (development class SDWA).

The transaction consists of two screens. In the first screen the user can enter flight identifiers and either request flight details (by pressing a *Display* pushbutton) or press the *Change* pushbutton to change the data of screen 200.

Setting Screen Field Attributes



The field attributes are now set dynamically, according to whether the *Display* button or the *Change* button was selected. In both cases the same screen is now called, but with different field attributes.

If the same attributes need to be changed for several fields at the same time, these fields can be grouped together. For example, in order to change the fields in screen 200 dynamically, we assign these fields in the Screen Painter to the group MOD. You can specify up to four modification groups for each field. The contents of the *Groups* field are stored in the SCREEN table.

The changes to the attributes of the fields in this group can be implemented in a PBO module:

```
MODULE MODIFY_SCREEN OUTPUT.
  CHECK MODE = CON_SHOW.
  LOOP AT SCREEN.
    CHECK SCREEN-GROUP1 = 'MOD'.
    SCREEN-INPUT = '0'.
    MODIFY SCREEN.
  ENDLOOP.
ENDMODULE.
```

The memory table SCREEN contains each field of the current screen together with its attributes.

The LOOP AT SCREEN statement puts this information in the header line of this system table.

In this example taken from transaction tz50, if the user chooses *Display* then SCREEN-INPUT is set to '0' and all fields belonging to the MOD group thus become display-only fields.

Because attributes have been changed, the MODIFY SCREEN statement is used to write the header line back to the table.

Changing Screen Field Attributes with the Function Field Selection

This topic describes how a special function *Field selection* (transaction SFAW and some function modules) support you in changing screen field attributes dynamically.

[Field Selection - Overview \[Page 728\]](#)

[Calling Field Selection \[Page 729\]](#)

[Combination Rules For Attributes \[Page 732\]](#)

[Screen Painter Attributes \[Page 734\]](#)

[Generating the Field Selection \[Page 735\]](#)

[Function Modules for Field Selection \[Page 736\]](#)

[Linking Fields \[Page 739\]](#)

[The Display Attribute 'Active' \[Page 741\]](#)

[Authorization for Field Selection \[Page 742\]](#)

Field Selection - Overview

The function *Field selection* allows you to change the attributes of screen fields dynamically at runtime. However, you should only use this option if you often need to assign different field attributes to the same screen for technical reasons. In this case, the same rules apply for all fields, so any field modification is clear.

The following basic rules apply:

- All fields involved in the field selection process are grouped together in field selection tables and maintained using the *Field selection* function.
- Maintenance is always by module pool and screen group.
- On screens belonging to the screen group "blank" ('_'), there is no dynamic field selection.
- Since the screen field attribute SCREEN-GROUP1 is reserved for central field selection, you cannot use it for another purpose at the same time.
- If you are working with special predetermined rules where any change is tantamount to a program change, do not use this function. Instead, make any changes in the program itself.

With field selection, you can activate or deactivate the following attributes at runtime:

- Input
- Output
- Mandatory
- Active
- Highlighted
- Invisible

You can also determine the conditions and the type of changes involved. During the event `PROCESS BEFORE OUTPUT`, you call a function module which checks the conditions and, if necessary, changes the attributes.

Field selection distinguishes between influencing fields and modified fields. Modified fields must, of course, be screen fields. All fields should be defined in the Data Dictionary and you should declare the corresponding tables globally in the module pool with the `TABLES` statement. At runtime, a function module analyzes the field contents of the influencing fields and then sets the attributes of the modified fields accordingly.

Combining Screens in Screen Groups

Rather than maintaining field selection separately for each screen of a program, you can combine logically associated screens together in a screen group. To assign a screen to a screen group, enter the group in the field **Screen group** on the Screen Painter attributes screen.

Calling Field Selection

Calling Field Selection

To call field selection, select *Tools* → *ABAP Workbench* → *Development* → *Other tools* → *Field selection*. Maintenance is by program and screen group.

Module pool



Screen group

Selection

☒ Influencing fields

☐ Modified fields

☐ Assign tables to screen group

 Display  Change

First, you must declare the table names of the fields involved. Choose *Assign tables to screen group* and enter the tables, for example:

Delete entry

Module pool

Screen group

Tables for field selection

Table name	Text
<input data-bbox="225 1559 392 1603" type="text" value="SPFLI"/>	<input data-bbox="611 1559 1262 1603" type="text" value="Connection offers for flights"/>

Save your entries and choose *Influencing fields* to enter the desired influencing fields into the list and optionally specify a NOT condition, a default value, and a field *Cust*, for example:

Choose Delete field Copy reference					
Module pool		SAPMTXXX			
Screen group		BILD			
Influencing fields					
Infl. field	Op.	Non-condition	Default	Cust	Text
SPFLI-CARRID	NE	LH		<input type="checkbox"/>	Airline

The NOT condition is to be understood as a preselection. If one of the fields satisfies the NOT condition, it is not relevant for the following screen modification. Using the NOT condition may improve performance.

Influencing field:	SPFLI-CARRID
NOT condition:	NE LH

SPFLI-CARRID is relevant for the field selection only if its contents are not equal to LH at runtime.

At runtime, the system uses the default value for the field modification if it cannot find the current value of the influencing field in the list of maintained values. You must define the default value yourself. This option allows you to maintain all the forms of an influencing field, which have the same influence, with a single entry.

By setting the field *Cust* accordingly, you can decide whether to allow customers to use the corresponding field for field selection or not. This field selection option is based on the predefined SAP field selection and allows customers to set screen attributes to suit their own requirements within a framework determined by application development. Many application areas have their own special Customizing transactions for this purpose (these are parameter transactions related to the Transaction SFAC; refer here to the documentation on the function module FIELD_SELECTION_CUSTOMIZE)

Then, choose *Modified fields* to enter all modifiable fields into a list, for example:

Choose Delete field Copy reference			
Module pool		SAPMTXXX	
Screen group		*	
Modifiable fields			
Modified field	Field	Cust	Text
SPFLI-AIRPFROM	2	<input type="checkbox"/>	Dep. airport

Calling Field Selection

You can again set the field *Cust* accordingly if you want to allow customers to use these modifiable fields in special Customizing transactions. If *Cust* is selected, customers can also modify the field.

Each of these influencing and modifiable fields has an internal number which is unique for each program. When you generate by pressing F16, the number is automatically placed in SCREEN-GROUP1 of the appropriate screens and cannot be changed in Screen Painter. This enables the system to establish a one-to-one relationship between the field name and SCREEN-GROUP1.

Finally, you create links between influencing and modifiable fields from the two lists: Specify which contents of an influencing field influences the modifiable field in which way.

To link fields, select the fields from both lists with *Choose* or double-click. If you select an influencing field, the list of modifiable fields appears and vice versa. From this list, select the desired link. A list appears in which you can enter the relevant conditions, for example:

Next modified field		Infl.field+	Delete entry
Module pool	SAPMTXXX		
Screen group	BILD		
Infl. field	SPFLI-CARRID	Airline	
Modified field	SPFLI-AIRPFROM	Dep. airport	

Contents	Input	Output	Active	Oblig.	Highl.	Invisible
LH	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

The entry above results in suppressing the display of field SPFLI-AIRPFROM on those screens, in whose PBO the corresponding field selection function modules are called and if SPFLI-CARRID then contains 'LH' (see [Function Modules for Field Selection \[Page 736\]](#)).

The function of the *Active* attribute is described in [The Display Attribute 'Active' \[Page 741\]](#).

Combination Rules For Attributes

If several influencing fields influence the same modified field, there must be a combination rule to determine how these influences are linked. You use the tables to below establish how a single field attribute is set, if it is activated and/or deactivated by different influences. The screen processor controls the combination of several attributes.

Input			
		Field 1	
		' _ '	'X'
Field 2	' _ '	' _ '	' _ '
	'X'	' _ '	'X'

Output			
		Field 1	
		' _ '	'X'
Field 2	' _ '	' _ '	' _ '
	'X'	' _ '	'X'

Active			
		Field 1	
		' _ '	'X'
Field 2	' _ '	' _ '	' _ '
	'X'	' _ '	'X'

Mandatory			
		Field 1	
		' _ '	'X'
Field 2	' _ '	' _ '	'X'
	'X'	'X'	'X'

Highlighted			
		Field 1	
		' _ '	'X'
Field 2	' _ '	' _ '	'X'
	'X'	'X'	'X'

Combination Rules For Attributes

Invisible			
		Field 1	
		'_'	'X'
Field 2	'_'	'_'	'X'
	'X'	'X'	'X'

Description of characters:

_ = switched off (off), X = switched on (on)

If Field 1 makes a screen field invisible (X), Field 2 cannot change this.

Screen Painter Attributes

With screen modification, the system takes into account not only the entries you make during field selection, but also any entries made in Screen Painter. This means that the result of the above combination is linked to the screen field attributes according to the same linking rules as described in [Combination Rules For Attributes \[Page 732\]](#).

To take advantage of the full dynamic modification range, you should use the following attributes in Screen Painter:

```
Input = 'X'  
Output = 'X'  
Mandatory = ' _'  
Invisible = ' _'  
Highlighted = ' _'.
```

Conversely, you cannot change the values defined on the screen in the following manner:

```
Input = ' _'  
Output = ' _'  
Mandatory = 'X'  
Invisible = 'X'  
Highlighted = 'X'
```

If you enter the following combination of influences, it is not really a valid combination, since the combination rules stipulate that the specified display attributes cannot be changed by another influencing field (or the screen).

```
Input = 'X'  
Output = 'X'  
Active = 'X'  
Mandatory = ' _'  
Highlighted = ' _'  
Invisible = ' _'
```

When you go into field selection again, any such ineffective influence are not displayed, except in the case where you have defined a default value for the influencing field; here, display and maintenance of such an influence can be useful.

Generating the Field Selection

If the list of modified fields has changed at all, you must generate the field selection. This produces consecutive numbers for the modified SCREEN-GROUP1 fields in the screens of the relevant module pool.

To do so, choose *Generate* in the transaction SFAW.

Function Modules for Field Selection

To activate field selection for a screen, at the PROCESS BEFORE OUTPUT event, you can call either FIELD_SELECTION_MODIFY_ALL or FIELD_SELECTION_MODIFY_SINGLE. Both these function modules determine the contents of the influencing fields, refer if necessary to the combination rules and execute the screen modification. FIELD_SELECTION_MODIFY_ALL executes the LOOP AT SCREEN statement itself. However, with FIELD_SELECTION_MODIFY_SINGLE, you must code this yourself and call the function module within this loop. You can thus perform your own additional screen modifications within the LOOP.

Examples of calling the function modules in the event PBO:

```
CALL FUNCTION 'FIELD_SELECTION_MODIFY_ALL'
  EXPORTING MODULEPOOL = MODULEPOOL
           SCREENGROUP = SCRGRP.
```

or

```
LOOP AT SCREEN.
  IF SCREEN_GROUP1 NE SPACE AND
     SCREEN-GROUP1 NE '000'.
    CALL FUNCTION 'FIELD_SELECTION_MODIFY_SINGLE'
      EXPORTING MODULEPOOL = MODULEPOOL
             SCREENGROUP = SCRGRP.
```

```
* Separate special rules
  MODIFY SCREEN.
ENDIF.
ENDLOOP.
```

or

as a), but includes your own **LOOP AT SCREEN** for special rules.

You must decide in each individual case which of the options b) or c) elicits the best performance.

Since the *Module pool* and *Screen group* parameters determine the field selection, you must maintain influences for these.

The *Module pool* parameter defines, in main memory, which loaded module pool you use to search for the current values of the influencing fields.

When you call a function module, you must not directly include the system fields SY-REPID and SY-DYNGR. Instead, transfer their contents to other fields at a suitable code position, for example:

```
MODULEPOOL = SY-REPID.
SCRGRP     = SY-DYNGR
```

Sometimes, the *Module pool* values may differ from the current SY_REPID value.

If the *Screen group* parameter is blank, the system uses the current contents of SY-DYNGR. This is not possible for the *Module pool* parameter because the value '_' (blank) prevents any field modification.

Regard a transaction resembling TZ50 in the development class SDWA (see [Setting Screen Field Attributes \[Page 724\]](#)).

Function Modules for Field Selection

Suppose the dynpro of the second screen contains the following module call in the PBO event:

```
PROCESS BEFORE OUTPUT.
```

```
...
```

```
MODULE MODIFY_SCREEN.
```

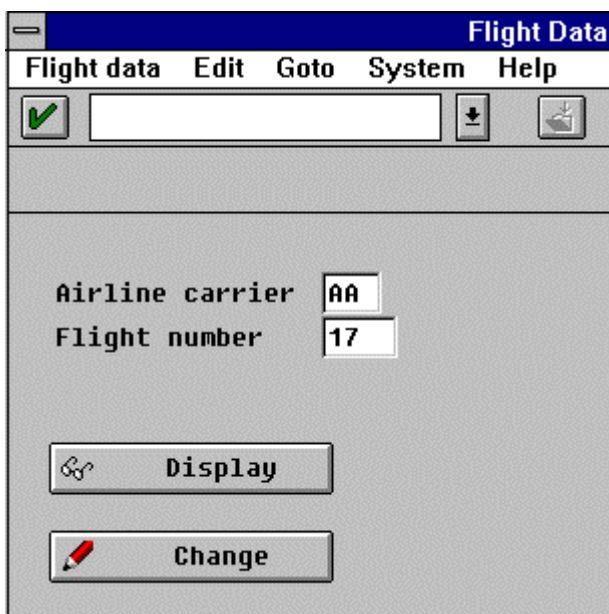
Suppose the module MODIFY_SCREEN contains the following function call:

```
MODULE MODIFY_SCREEN OUTPUT.
```

```
CALL FUNCTION 'FIELD_SELECTION_MODIFY_ALL'
  EXPORTING
    SCRENGROUP = 'SCREEN'
    MODULEPOOL  = 'SAPMTXXX'
  EXCEPTIONS
    OTHERS      = 1.
```

Suppose that for the screen group SCREEN and the module pool SAPMTXXX the influences in Transaction SFAW are maintained as shown in the figures of [Calling Field Selection \[Page 729\]](#).

After calling the transaction, suppose these entries are made:



After choosing *Change*, the following screen appears:

Function Modules for Field Selection

From city	FRANKFURT
Destination	NEW YORK
Dest. airport	JFK
Flight time	08:24:00
Departure	10:10:00
Arrival	11:34:00
Distance	6.162
Dist. in	KM

However, if instead of 'LH' as airline carrier 'AA' is entered, the following screen appears:

From city	NEW YORK
Dep. airport	JFK
Destination	SAN FRANCISCO
Dest. airport	SFO
Flight time	06:01:00
Departure	13:30:00
Arrival	16:31:00
Distance	2.572
Dist. in	MLS

When entering 'LH', the field SPFLI-AIRPFROM is invisible. When entering 'AA', it is visible as *Dep. airport*.

Linking Fields

Linking Fields

Every influencing field influences a field which can be modified regardless of other fields. A link of influencing fields is desired in some cases but then only possible via the definition of help fields which you must set in the application program before calling up the function module (see also [Combination Rules For Attributes \[Page 732\]](#)).

This restriction helps the transparency of the field selection.

Examples of Links

Assume the following fields:

Influencing fields:	F4711, F4712
Field that can be modified:	M4711

The following cases can only be implemented via a detour:

OR Condition and "Ready for Input"

If	F4711 = 'A' OR F4712 = 'B',
then	M4711 is ready for input.

Solution:

Define H4711 as an influencing field in SFAW;
define the following condition in SFAW:

```
if H4711 = 'AB'
then M4711 input on (that is, input = 'X')
```

In the application program, you must program the following before calling up the function module:

```
IF F4711 = 'A' OR F4712 = 'B'.
  H4711 = 'AB'.
ENDIF.
```

AND Condition and "Mandatory"

If	F4711 = 'A' AND F4712 = 'B',
then	M4711 obligatory and only then.

Solution:

Maintenance in the field selection:
If H4711 = 'AB',
then M4711 is a required-entry field
(H4711 = 'AB' only precisely with the above AND condition)

In the application program, you program:

```
IF F4711 = 'A' AND F4712 = 'B'
  H4711 = 'AB'
ELSE.
  H4711 = ....
ENDIF.
```

On the other hand, you can represent the following cases directly:

AND Condition and "Ready for Input"

If	F4711 = 'A' AND F4712 = 'B',
then	M4711 is ready for input.
thus:	if F4711 <> 'A' OR F4712 <> 'B'

Solution:

Screen: M4711 ready for input

Field selection:

Influencing field	F471 1	Value 'A'	Input = 'X'
		Value 'A1'	Input = ' '
		Value 'AX'	Input = ' '
Influencing field	F471 2	Value 'B'	Input = 'X'
		Value 'B1'	Input = ' '
		Value 'BX'	Input = ' '

OR Condition and "Mandatory"

If	F4711 = 'A' OR F4712 = 'B',
then	M4711 is a required-entry field.

Solution:

Screen: Mandatory is switched off

Influencing field	F4711 Value 'A',
	mandatory = 'X'
Influencing field	F4712 Value 'B',
	mandatory = 'X'

The possibility to define a NOT condition for an influencing field gives further variants of the field selection definition.

The Display Attribute 'Active'

The Display Attribute 'Active'

At present, this display attribute has only one consequence. If SCREEN-ACTIVE = '0' in the ABAP program, the following occurs at runtime with the MODIFY SCREEN statement:

```
SCREEN-INPUT = '0'  
SCREEN-OUTPUT = '0'  
SCREEN-INVISIBLE = '1'
```

If SCREEN-ACTIVE = '1', nothing at all occurs.

If, on the other hand, SCREEN-INPUT = '0', SCREEN-OUTPUT = '0' and SCREEN-INVISIBLE = '1', the internal result would be SCREEN-ACTIVE = '0' (without consequences).

Even when SCREEN-ACTIVE = '0', a module specified with the relevant FIELD statement in the screen flow logic always runs. However, a module with a deactivated field is not meant to run. If you do not intend this, you can separate the FIELD and MODULE statements with a period.

In the case of inactive fields with additional specifications like ON INPUT, ON REQUEST..., the screen modules do not run, whereas modules without additional specifications are always processed.

Authorization for Field Selection

The authorization object for *Field selection* is "central field selection" (S-FIELDSEL). This object consists of an activity and a program authorization group. The latter is taken from the program authorizations.

The following activities are allowed:

- '02' = Change
- '03' = Display
- '14' = Generate field selection information on screens
- '15' = Assign relevant tables to field selection

Using Subscreens

Using Subscreens

A subscreen is an independent screen that is displayed in an area of another ("main") screen. You might want to use a subscreen to vary certain fields in a main screen. For instance, you can define an area in a main screen where supplementary fields appear, depending on the input the user makes in a previous screen.

You create a subscreen as follows:

1. Adjust the frame of the subscreen within the "main" screen until it has the size you want. Name the subscreen in the *Field name* field.
2. Create a screen with screen type *Subscreen*.
3. Arrange the fields within the subscreen so that they appear in the main screen exactly where you want them to be.

If the subscreen is defined to be larger than the available area in the main screen, then only that part of the subscreen will be visible that fits in the area available (measured from the upper left corner).

To use a subscreen, you must call it in the flow logic (both PBO and PAI) of the main screen. The `CALL SUBSCREEN` statement tells the system to execute the PBO and PAI events for the subscreen as part of the PBO or PAI events of the main screen. You program the ABAP modules for a subscreen exactly as you would program modules for a main screen.

The flow logic of your main program should look as follows:

PROCESS BEFORE OUTPUT.

```
...
CALL SUBSCREEN <area> INCLUDING '<program>' '<screen>'.
...
```

PROCESS AFTER INPUT.

```
...
CALL SUBSCREEN <area>.
...
```

Area is the name of the subscreen area you defined in your main screen. This name can have up to ten characters. *Program* is the name of the program to which the subscreen belongs and *screen* is the subscreen's number. You can pass *Program* and *Screen* either as literals or as variables. The screen number must always be four characters long.

The subscreen is displayed in any screen whose PBO contains the statement `CALL SUBSCREEN <area> INCLUDING '<program>' '<screen>'`. Its PAI is processed when you include the `CALL SUBSCREEN <area>` statement in the PAI of the main screen.

You maintain subscreens like normal screens in the Screen Painter. In the screen attributes, you must set the subscreen option. Subscreens may not call other subscreens. However, a single main screen can contain several subscreen areas.

If you include a subscreen from a different module pool, you must pass the global data using `EXPORT ... TO MEMORY` and `IMPORT ... FROM MEMORY`.

Note the following restrictions that apply to subscreens:

- You cannot use the `CALL SUBSCREEN` statement between `LOOP` and `ENDLOOP` or between `CHAIN` and `ENDCHAIN`.

- A subscreen may not have a named OK_CODE field. Instead, the function codes are passed to the OK_CODE field of the main screen.
- Field names in subscreens must be unique in respect of other subscreens called from the main screen.
- Subscreens may not contain an AT EXIT-COMMAND module. EXIT handling is only possible in the main screen.
- Using the SET TITLEBAR, SET PF-STATUS, SET SCREEN and LEAVE SCREEN statements in a subscreen causes a runtime error.

The enhancement concept in the R/3 System contains screen exits. For further details of possible enhancements, start Transaction CMOD and choose *Utilities* → *SAP enhancements*. For full documentation, see the online manual for Transaction SMOD and the *Enhancements to the SAP Standard* documentation in the R/3 Library.

Manipulating the Cursor

Manipulating the Cursor

In the PBO event, you can tell the system to place the cursor in a certain screen field. Cursor positioning is one way to make your transaction more user-friendly.

Positioning the Cursor in the Screen

When displaying a screen, the system automatically places the cursor in the first field ready for input.

However, you can specify the field where the cursor is to appear yourself. You have two options.

1. Enter the appropriate field name as the *Cursor position* when you define the screen in the Screen Painter. The *Cursor position* field appears in the *Screen Attributes*.
2. You can also set the cursor using the ABAP command SET CURSOR FIELD. Use the SET CURSOR statement as follows:

SET CURSOR FIELD <field name>.

The <field name> can be an explicit field name in quotes, or a variable containing the field name. To place the cursor in a particular position in a field, use the OFFSET parameter, entering the character position for the cursor in <position>:

SET CURSOR FIELD <field name> OFFSET <position>.

The system offsets the cursor by the given position from the start of the line.

If you have a step-loop in your screen, you can place the cursor on a particular element in the step-loop block. Use the LINE parameter, entering the line of the loop block where you want to place the cursor:

SET CURSOR FIELD <fieldname> LINE <line>.

If you want, you can use the OFFSET and LINE parameters together.

Tab strips

Tab strips allow you to easily and comprehensively define different object components of an application on one screen and to navigate between them. In contrast to the classical screen change, the user at one glance sees all destinations he or she must call to accomplish the given task. Using this technique, the user can comprehend the structure of an application more intuitively as with conventional techniques such as the *Goto* menu. This reduces the learning expense and facilitates the usage.

Screen Composition: Tab Control

Screens Elements Dynamics Menu design System Help

Request Request type

Short text

Assignments Control Period Closing General data Investment management Design info

Company code Soesel & Partner GmbH &

Business area

Plant Helsinki

Object class

Profit center

Cost center respons.

WBS element

Request. cost center

Request. co. code

Sales order

Location/plant /

External order no.

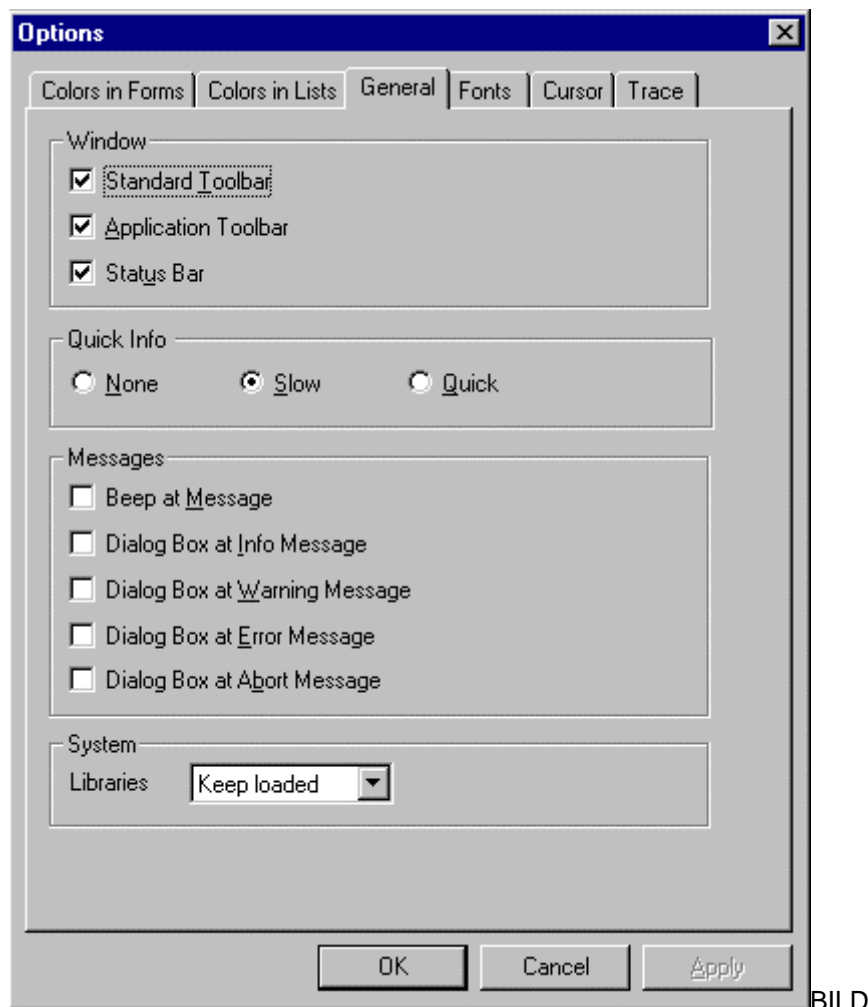
For more information on tab strips, see the following sections.

Using a tab strip

Using a tab strip

You use a tab strip to

- structure complex applications and show the navigation supported within. Use a tab strip whenever different object components or logical views for one object exist for which navigation covers more than one screen. Such applications are usually characterized by constant screen areas such as header data and changing contents in the lower screen areas (see Fig. 1).
- present large screens and the navigation between the boxes in a different way.
- display property sheets showing attributes of complex objects or applications. Usually, property sheets are displayed in dialog boxes.



You **must not** use a tab strip if

- the use of the object components is designed as a stand-alone application or if the tab strip environment (menus, pushbuttons, header data, and so on) cannot be kept constant.

- a certain processing sequence is required. A tab strip allows the user free navigation among all object components which contradicts a mandatory sequence.
- the object components can be processed dynamically. This is the case if, for example, entering a value on one tab page unexpectedly produces new tab titles.

Tab strip components

Tab titles

Keep the tab titles as short and meaningful as possible. Sort them according to the work flow from left to right.

As the advantage of tab strips consists of displaying all navigation possibilities at once, try not to define more tab titles than can be displayed. However, due to the complexity of R/3 transaction, this is not always possible.

If you define more tab titles than can be displayed, the user must scroll. For this purpose, the system automatically displays scroll buttons for horizontal scrolling at the upper right margin. To get a complete overview of all navigation targets, the user can call a list of all tab titles from a dialog menu (see Fig. 1).

The R/3 System does not support multiple-line tab titles. If multiple lines were used instead of the scroll buttons, due to the language dependency of the titles a different number of tab title lines would appear for each logon language. Since the tab page is of fixed size, the developer would have to save as many blank lines above the tab strip as might be needed under worst conditions. This would waste screen space and reduce readability.

If tab titles correspond to entries in the *Goto* or *Options* menus, delete these entries from the menus. The user must always be able to use the *Goto* menu to reach the screen that contains the tab strip.

For the duration of one processing session, the number as well as the contents of the tab titles should be the same.

The tab strip allows you to define icons as tab titles, however, only in combination with texts. If you used icons only, this would nullify the advantages of a transparent application structure stated above.

Tab page

The Screen Painter locks the first line of a tab page, which the system needs to draw the tab titles. Otherwise, the layout rules for creating screens apply.

A tab strip requires three additional screen lines for the tab titles line, the lower border, and the locked first line.

A tab page together with the tab titles texts corresponds to a group box. Therefore, you need no further box that includes all fields of a tab page.

When converting: Fields that are grouped on old screens using group boxes receive no more box; use the old box title as tab title of the tab page.

However, you can use several group boxes on one tab page. In this case, make sure that the tab title contains a meaningful description of the groups of fields combined on the tab page.

Environment of the tab strip

You must provide a constant environment of the tab strip. This means that if you go from one tab page to another, the menu bar and the application bar must not change. For special functions that are active only on one tab page, you must, therefore, use pushbuttons, which you can position anywhere on the respective page. These functions must not appear in the menu.

Leave a blank line between the header data and the tab strip, as usual.

Tab strip components

Navigation

The user perceives a screen that contains a tab strip as a unit. The individual tab pages thus are components of the screen, such as boxes or groups of fields. The function keys F3, F12, and F15 therefore do not refer to individual tab pages but to the underlying screen.

If the user navigates from a tab page to another screen, make sure to reactivate the previously active tab page when the user returns to the screen.

Programming tab strips

To create a tab strip in the R/3 System, make sure to meet the following prerequisites:

- GUI version 4.0 and higher
- frontend: Motif, Windows 95, MacOs, NT 3.51 and higher

Scrolling in the tab strip

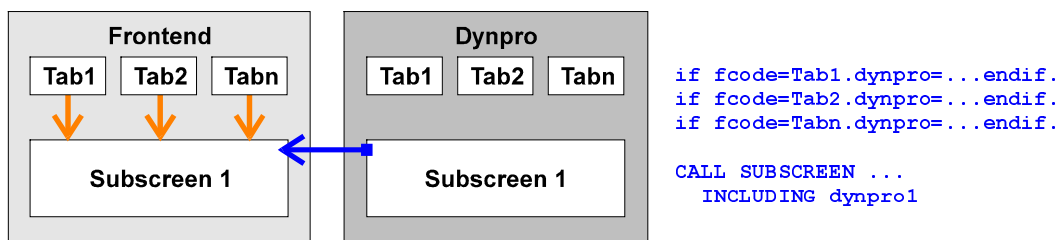
The GUI element *Tab strip* offers two alternatives for scrolling. In both cases, the tab titles are defined as pushbuttons and the tab page as subscreen. However, in one case, scrolling is achieved via the backend, in the other locally in the GUI.

Scrolling via the backend

All tab titles use one subscreen area and the application uses the function codes of the individual titles to determine which subscreen to display.

The advantage of this method is that field checks are executed only for the actually displayed area.

The disadvantage is that each click on a tab title triggers a backend communication.



Scrolling locally using the GUI

Define an individual subscreen area for each tab title. Define the function codes of the titles using the type "P tab strip code"

Programming tab strips

Line	1	Column	2	Length	9	Vis.len.	9
FctCode	TAB2			FctType		Height	1
Groups			Type of function code for pushbutton				
Dict							
Format							
Frm Dict	<input type="checkbox"/>						
Modific.	<input type="checkbox"/>						
ConvExit	<input type="checkbox"/>						
Param.ID	<input type="checkbox"/>						
<input type="checkbox"/> SET param. <input type="checkbox"/> GET param. <input type="checkbox"/> Up./lower <input type="checkbox"/> W/o template <input type="checkbox"/> Foreign key							

Type	Short text
E	Exit command (MODULE xxx AT EXIT-COMMAND)
S	System function
T	Call a transaction
	Normal application function
P	Local GUI functions
H	Internal use

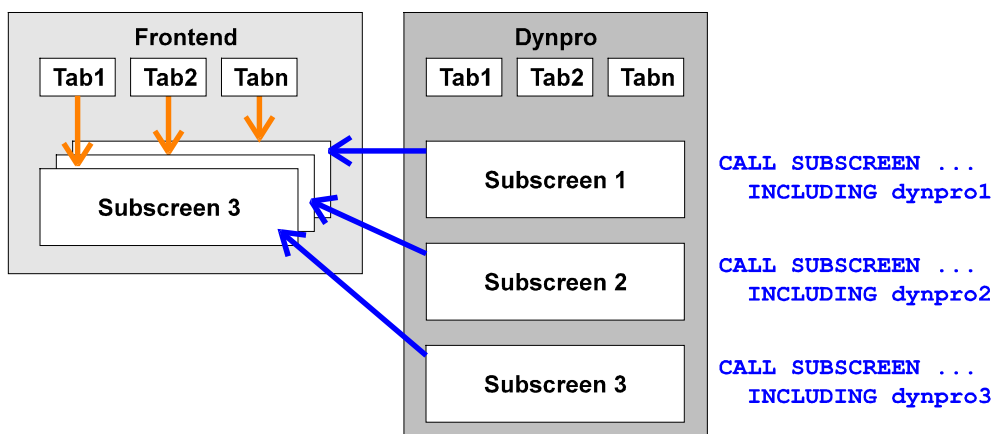
☒ ☐ ☐ ☐ ☒

In the flow logic, include all subscreens used into PBO. This ensures that all tab pages exist locally in the GUI.

With this method, the user scrolls locally in the GUI between the individual tab pages. The tab strip thus behaves like a large screen.

The advantage is that no backend communication is needed when the user clicks on a tab title. The disadvantage is that for each backend communication, the system triggers all screen checks. For example, if the user scrolls within a table control on the first tab page, the system may go to an empty required entry field on the fourth tab page.

Therefore, this method is recommended for display transaction rather than for change transactions.



Combining the two scrolling methods

You can use a combination of the two scrolling methods. For example, if for editing individual tab pages, extensive database selections are required, you can use the first scrolling method for these tab pages, while offering local scrolling for simple tab pages.

For information on usage and layout of tab strips, refer to transaction BIBS. Use *Elements* to find a sample program for tab strips and click on the pushbutton *Rules* for notes on how to lay out tab strips.

Defining a tab strip in the Screen Painter

Defining a tab strip in the Screen Painter

To define a tab strip at the current development stage, you must use the graphical Screen Painter.

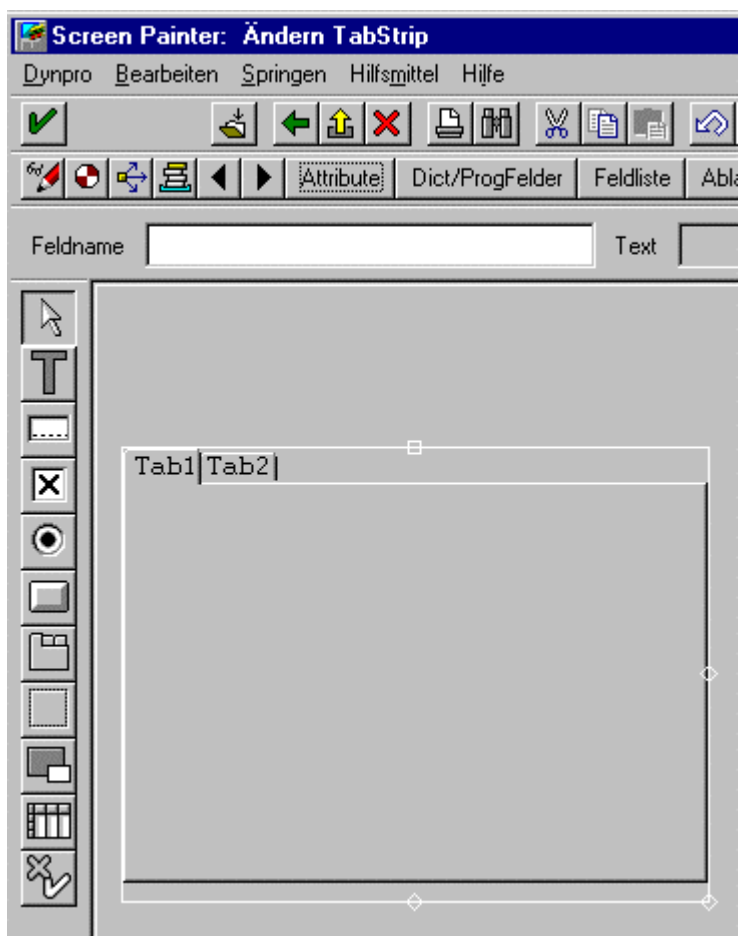
Execute the following steps to create a simple tab strip:

1. Set the tab strip borders.
2. Define the tab titles.
3. Define and allocate the subscreen area.
4. Program the flow logic.

Set the tab strip borders

To determine the tab strip area, proceed as follows:

1. Click on the tab strip icon in the elements palette of the Screen Painter.
2. Determine the area by dragging the mouse over the screen.
3. Specify the field name for the new screen element. The field name is the area name you use in the ABAP statement CONTROLS.



Define the tab titles

By default, the new tab strip contains two tab titles. Technically, tab titles are treated like pushbuttons. This becomes obvious when you double-click on a tab title in the attributes window. For each tab title, enter field name, field text, and function code.

You can also define tab titles as templates and fill them with texts at runtime.

To create additional tab titles, drag pushbuttons into the tab titles line.

Define the subscreen area

Each tab title must be allocated to a subscreen area.

Proceed as follows:

1. Select one or more tab titles.
2. Click on the subscreen icon.
3. Define the subscreen area within the tab strip by dragging the mouse. Caution: the first line of the tab strip is locked.

You can also allocate the subscreen by entering the name of the subscreen area into the reference field in the attributes window or in the field list (see Fig. 4).

Program the flow logic

Include the subscreens you use into the flow logic as follows:

PROCESS BEFORE OUTPUT.

```
...  
    CALL SUBSCREEN <area> INCLUDING <program> <screen>.
```

```
...
```

PROCESS AFTER INPUT.

```
...  
    CALL SUBSCREEN <area>.
```

```
...
```

In the module pool, declare the tab strip similar to the table control using the CONTROLS statement.

CONTROLS: MYTABSTRIP TYPE TABSTRIP.

In this example, MYTABSTRIP is the field name of the tab strip in the Screen Painter. At runtime, at present only the attribute ACTIVETAB is available, which determines the currently active tab page. You fill it with the function code of the corresponding tab title.

To activate the tab title whose function code is "F1", use the statement

MYTABSTRIP-ACTIVETAB = 'F1'.

The tab strip controlling is fully integrated into the screen concept, allowing you to use it for batch input.

Scrolling via the backend

Defining a tab strip in the Screen Painter

```

DATA: DYNPRONR(4) TYPE C,
      MODUL(30) TYPE C VALUE 'PROGRAMMNAME',
      OK_CODE LIKE SY-UCOMM
      ....

CONTROLS: TABSTRIP1 TYPE TABSTRIP.
...

Flow logic:

PROCESS BEFORE OUTPUT.
  MODULE SETPAGE.
  MODULE MODIFYSCREEN.
  CALL SUBSCREEN SUB1 INCLUDING MODUL DYNPRONR.
...

PROCESS AFTER INPUT.
  CALL SUBSCREEN SUB1.
  MODULE FCODE.
...

*-----*

MODULE SETPAGE OUTPUT.
  IF DYNPRONR IS INITIAL.
    DYNPRONR = '0200'.
    TABSTRIP1-ACTIVETAB = 'TAB1'.
  ENDIF.
ENDMODULE.

*-----*

MODULE MODIFYSCREEN OUTPUT.
  LOOP AT SCREEN.
    ...
    MODIFY SCREEN.
  ENDLOOP.

*-----*

MODULE FCODE.
  CASE OK_CODE.
    WHEN 'TAB1'.
      DYNPRONR = '0200'.
      TABSTRIP1-ACTIVETAB = OK_CODE.
      CLEAR OK-CODE.
    WHEN 'TAB2'.
      DYNPRONR = '0300'.
      TABSTRIP1-ACTIVETAB = OK_CODE.
    ....

```

Scrolling locally using the GUI

```

Ablauflogik:

PROCESS BEFORE OUTPUT.

```

Defining a tab strip in the Screen Painter

```
CALL SUBSCREEN sub1 INCLUDING 'modul1' '0100'.
CALL SUBSCREEN sub2 INCLUDING 'modul2' '0101'.
CALL SUBSCREEN sub3 INCLUDING 'modul3' '0102'.
...
PROCESS AFTER INPUT.
CALL SUBSCREEN sub1.
CALL SUBSCREEN sub2.
CALL SUBSCREEN sub3.
...
```

Further Notes

Further Notes

Compressing Screens Containing Tabs

Compressing screens does not change the size of the tab control itself, since the compression relates to the contents of a page (subscreen). If you are using local scrolling at the SAPgui within a tab, and a page is compressed so that there are no more elements visible, the system deletes the tab title. If a tab control contains no more tab titles, the control itself is deleted.

Resizing a tab or a tab page

Tab

You can only resize a tab within the range of its specified maximum and minimum size. The maximum size is specified in the Screen Painter. The minimum size is determined in the Screen Painter by the smallest minimum size for any of the tab pages, whereby you must also ensure that the data can be displayed properly. The size of the tab at any given time always lies between the maximum and minimum size and is calculated in relation to the window size. The size of the tab pages can be determined from the size of the tab itself.

Tab Page

The individual pages within a tab can be of different sizes. You can resize them within the range of their specified maximum and minimum size. The maximum size is derived from the definition of the element in the Screen Painter, where you must ensure that the page fits within the tab itself. You specify the minimum size in the Screen Painter. This should be large enough to ensure that the elements within the subscreen can still be displayed properly. The size of the page at any given time is determined by the size of the tab itself.

Dynamic Modifications

- You can use dynamic text on tab titles exactly as you would with a pushbutton.
- You can use dynamic screen modification techniques.

Using Tables in a Screen

ABAP offers two mechanisms for displaying and using table data in a screen. These mechanisms are *table controls* and *step loops*. Table controls and step loops are types of screen tables you can add to a screen in the Screen Painter. For example, the following screen contains a table control at the bottom:

Airline carrier	AA	AMERICAN AIRLINES				
Flight number	17					
From city	NEW YORK	JFK				
Destination	SAN FRANCISCO	SFO				
Flight time	06:01:00	Distance	2.572	MLS		
Departure	13:30:00	Arrival	16:31:00			

	Date	FlgtPrice	Curr.	Plane type	Capacity	Occupied
<input type="checkbox"/>	30.01.1995	689,66	USD	747-400	660	10
<input type="checkbox"/>	01.02.1995	689,66	USD	747-400	660	20
<input type="checkbox"/>	01.06.1995	689,66	USD	747-400	660	38
<input type="checkbox"/>	04.06.1995	689,66	USD	747-400	660	38
<input type="checkbox"/>						
<input type="checkbox"/>						
<input type="checkbox"/>						

This chapter describes how to program the screen flow logic and ABAP code that let you use screen tables. For information on using screen tables, see:

[Introduction \[Page 761\]](#)

[Using the LOOP Statement \[Page 763\]](#)

[Using Table Controls \[Page 770\]](#)

[Example Transaction: Table Controls \[Page 775\]](#)

[Using Step Loops \[Page 779\]](#)

Introduction

Introduction

This section provides a quick overview of how table displays are used in a screen.

For an example of the principles outlined here, see transactions TZ60 and TZ61, which demonstrate the use of table controls and step loops, respectively. (TZ60 and TZ61 are sample transactions in development class SDWA, which is delivered with your system).

Table Controls and Step Loops

Table controls and step loops are objects for screen table display that you add to a screen in the Screen Painter. From a programming standpoint, table controls and step loops are almost exactly the same. Table controls are simply enhanced step loops that display data with the Look and Feel of a table widget in a desktop application. With table controls, the user can:

- scroll through the table vertically and horizontally
- re-size the width of a column
- scroll within a field (when field contents are wider than the field)
- select table rows or columns
- re-order the sequence of columns
- save the current display settings for future use

Table controls also offer special formatting features (some automatic, some optional) that make tables easier to look at and use. These are for example:

- automatic table resizing (vertical and horizontal) when the user resizes the window
- separator lines between rows and between columns (vertical and horizontal)
- column header fields for all columns

One feature of step loops is that their table rows can span more than one line on the screen. By contrast, the rows in a table control are always single lines, but can be very long. (Table control rows are scrollable.)

In general, many of the features provided by the table control are handled locally by your system's SAPGUI frontend. You don't have to program any of these features (except vertical scrolling) in your ABAP transaction.

Screen Table Processing

A screen table is a repeated series of table rows in a screen. Each entry contains one or more fields, and all rows have the same field structure. Screen tables are either table controls or step loops. A table control displaying flight data might look as follows:

	Date	FlgtPrice	Curr.	Plane type	Capacity	Occupied
	30.01.1995	689,66	USD	747-400	660	10
	01.02.1995	689,66	USD	747-400	660	20
	01.06.1995	689,66	USD	747-400	660	38
	04.06.1995	689,66	USD	747-400	660	38

Screen Tables and the LOOP Dynpro Statement

You process a screen table by looping through it as you would through the rows of a internal table. To do this, you place a LOOP...ENDLOOP dynpro statement in the screen's flow logic. This loop usually contains a call to an ABAP module, but other flow logic commands are also allowed. The system executes the module with each pass through the loop.

The LOOP dynpro statement has a variety of forms. The two most important forms let you:

- loop through a screen table alone
- loop through a screen table and an internal table in parallel

Screen Tables and Program Fields

You can declare screen table fields as database fields, internal table fields, structure fields, or other program fields. Screen table fields appear once in the field-list for the screen, and once in your program. Thus, all rows in a screen table share the same set of fields (like a "header area") in your program. During a LOOP in the flow logic, the system copies all fields for a screen table row into or out of the relevant program fields.

What the LOOP Statement Does

The LOOP statement is responsible for getting screen table values passed back and forth between the screen and the ABAP program. As a result, you *must* code a LOOP statement in both the PBO and PAI events for every table in your screen. At the very least, an empty LOOP...ENDLOOP must be there.

The LOOP statement also drives scrolling. At PBO, LOOP uses a parameter that tells where in the table to start looping. This parameter thus causes the update of the next screen table display. (For table controls, the parameter is the TOP_LINE field in the table control structure. For step loops, it is the CURSOR parameter to the LOOP statement.) Both the ABAP program and the system can set this parameter.

Note that the number of rows displayed in the screen table can change. This happens when the screen table is resizable and the user changes the height of the window. In this case, the next LOOP at PAI alters the number of table rows passed to the ABAP program at PAI.

Using the LOOP Statement

The LOOP...ENDLOOP screen command lets you perform looping operations in the flow logic. You can use this statement to loop through both table controls and step loops. Between a LOOP and its ENDLOOP, you can use the FIELD, MODULE, SELECT, VALUES and CHAIN dynpro keywords. Most often, you use the MODULE statement to call an ABAP module.

You *must* code a LOOP statement in both the PBO and PAI events for each table in your screen. This is because the LOOP statement causes the screen fields to be copied back and forth between the ABAP program and the screen field. For this reason, at least an empty LOOP...ENDLOOP must be there.

There are two important forms of the LOOP statement:

- **LOOP.**
This statement loops through screen table rows, transferring the data in each block to and from the corresponding ABAP fields in your program. The screen table fields may be declared in ABAP as anything (database table, structure or individual fields) except as internal table fields.
With step loops, if you are implementing your own scrolling (for example, with F21-F24) you must use this statement.
- **LOOP AT <internal table>.**
This statement loops through an internal table and the screen table rows in parallel. The screen table fields often are, but need not be, declared as internal table fields.
With this LOOP, step loop displays appears with scroll bars. This scrolling is handled automatically by the system.

For more details on the different LOOP statements, see:

[Looping Directly Through a Screen Table \[Page 764\]](#)

[Looping Through an Internal Table \[Page 766\]](#)

Looping Directly Through a Screen Table

Use the simple form of the LOOP statement

```
LOOP.
...<actions>...
ENDLOOP.
```

to loop through the currently displayed rows of a screen table. If you are using a table control, include the additional WITH CONTROL parameter:

```
LOOP WITH CONTROL <table-control>.
...<actions>...
ENDLOOP.
```

This simple LOOP is the most general form of the LOOP statement. If you use this LOOP, you can declare the screen table fields as any type (internal table, database table, structure or individual fields). The simple LOOP copies the screen table fields back and forth to the relevant ABAP fields. If you want to manipulate the screen values in a different structure, you must explicitly move them to where you want them.

Each pass through the loop places the next table row in the ABAP fields, and executes the LOOP <actions> (usually ABAP module calls) for it. In a PBO event, the LOOP statement causes loop fields in the program to be copied row by row to the screen. In a PAI event, the fields are copied row by row to the relevant program fields.

In an ABAP module, use the system variable SY-STEPL to find out the index of the screen table row that is currently being processed. The system sets this variable each time through the loop. SY-STEPL always has values from 1 to the number of rows currently displayed. This is useful when you are transferring field values back and forth between a screen table and an internal table. You can declare a table-offset variable in your program (often called BASE, and usually initialized with SY-LOOPC) and use it with SY-STEPL to get the internal table row that corresponds to the current screen table row.

(In the example below, the screen fields are declared as an internal table. The program reads or modifies the internal table to get table fields passed back and forth to the screen.)

```
***SCREEN FLOW LOGIC***
PROCESS BEFORE OUTPUT.
  LOOP.
    MODULE READ_INTTAB.
  ENDLOOP.

PROCESS AFTER INPUT.
  LOOP.
    MODULE MODIFY_INTTAB.
  ENDLOOP.

***ABAP MODULES***
MODULE READ_INTTAB.
  IND = BASE + SY-STEPL - 1.
  READ TABLE INTTAB INDEX IND.
ENDMODULE.

MODULE MODIFY_INTTAB.
  IND = BASE + SY-STEPL - 1.
```

Looping Directly Through a Screen Table

```
    MODIFY INTTAB INDEX IND.  
ENDMODULE.
```

Remember that the system variable SY-STEPL only has a meaning within the confines of LOOP...ENDLOOP processing. Outside the loop, it has no valid value.

Looping Through an Internal Table

The statement

LOOP AT <internal table>.

loops through an internal table and a screen table in parallel. In particular, LOOP AT loops through the portion of the internal table that is currently visible in the screen. You can use this form of the LOOP statement for both table controls and step loops.

The complete syntax for this form of the LOOP statement is:

```
LOOP AT <internal table> CURSOR <scroll-var>
      [WITH CONTROL <table-control>]
      [FROM <line1>] [TO <line2>].
...<actions>...
ENDLOOP.
```

This form of LOOP loops through the internal table, performing <actions> for each row. For each internal table row, the system transfers the relevant program fields to or from the corresponding screen table row.

When using step-loops, omit the CURSOR parameter in the PAI event. The FROM and TO parameters are only possible with step loops. (See [Using Step Loops \[Page 779\]](#).) The WITH CONTROL parameter is only for use with table controls.

The following topics provide more information:

[How the System Transfers Data Values \[Page 767\]](#)

[Scrolling and the Scroll Variables \[Page 768\]](#)

How the System Transfers Data Values

How the System Transfers Data Values

With the LOOP AT <internal table> statement, the screen table need not be declared as the internal table. Screen tables can also be database tables, structures or other program fields. If you don't define the screen table as an internal table, you must make sure that the correct fields are moved to and from the internal table header as needed during looping.

Note also that the fields you use in the screen table may be only a subset of the corresponding dictionary or internal table fields. The system transfers fields as needed in the screen table. Processing (in detail) is as follows:

- **PBO processing**

- A row of the internal table is placed in the header area.
- All loop statements are executed for that row.

Loop statements are most often calls to ABAP modules. Where necessary, these modules should move the internal table fields to the relevant program fields (dictionary table or other fields).

Example transaction TZ60 does this in the DISPLAY_FLIGHTS routine:

```
MODULE DISPLAY_FLIGHTS OUTPUT.  
  SFLIGHT-FLDATE = INT_FLIGHTS-FLDATE.  
  SFLIGHT-PRICE  = INT_FLIGHTS-PRICE.  
  SFLIGHT-CURRENCY = INT_FLIGHTS-CURRENCY.  
  SFLIGHT-PLANETYPE = INT_FLIGHTS-PLANETYPE.  
  SFLIGHT-SEATSMAX = INT_FLIGHTS-SEATSMAX.  
  SFLIGHT-SEATSOCC = INT_FLIGHTS-SEATSOCC.  
ENDMODULE.
```

- The system transfers values from the program fields to the screen fields with the same names.

- **PAI processing**

- A row of the internal table is placed to the internal table header area.
- All screen table fields are transferred to the ABAP program fields with the same names.
- All loop <actions> are executed for the current internal table row.

Again, <actions> is usually a call to an ABAP module. Where necessary, this module should first move the program field values to the header area for the internal table. This step over-writes data values from the internal table with those from the screen. Step 1 is necessary for cases where the screen table fields are only a subset of the fields defined for the internal table. If you want to update the internal table with new screen values, you must have the original table row as a basis in the header area.

Remember: If you want to update the internal table with the contents of the header area, use the MODIFY statement in the ABAP module. The system does not automatically make this update for you.

Note that there is a dynpro-language MODIFY statement and an ABAP MODIFY statement. Do not use the dynpro-language MODIFY to update internal tables.

Scrolling and the Scroll Variables

Scrolling with the LOOP AT <internal table> statement can be either automatic or programmer-implemented. Scrolling with scroll bars is automatic, and managed by the system. Scrolling with function keys or other pushbuttons must be implemented by the programmer. (The example transaction TZ60 gives an example of both.)

Both for table controls and step loops, vertical scrolling is controlled by the <scroll-var> parameter. This section describes how to use this variable. Horizontal scrolling (available only with table controls) is handled locally by the SAPGUI front end. Your ABAP code need not support horizontal scrolling in any way.

The first time a PBO event is processed, your program should set <scroll-var> to tell the system where to start displaying. For table controls, <scroll-var> is the TOP_LINE field in the TABLEVIEW structure. For step loops, declare a local program variable for use as the CURSOR parameter.

Thereafter, the system sets <scroll-var> in two situations:

- updates <scroll-var> whenever the user scrolls with the scroll bar
- increments <scroll-var> with each pass through a LOOP AT block

This works because when the user scrolls with the scroll bar, the system takes the following actions:

1. Triggers PAI for the current screen display.

Scrolling triggers a PAI event just as if the user had performed an action that transmitted a function code. During the PAI for scrolling with the scroll bars, the OK_CODE field has no value. (If you offer scrolling with function keys, the OK_CODE contains the function code for the function key.)

- Transfers screen values for the *current* display to the ABAP program.
- Performs PAI processing for the current screen values, using the current value of <scroll-var>. Each time through the LOOP, the system increments <scroll-var> by one, so your program can access the relevant internal table entries automatically.

(As usual, if your program detects any errors here, it should send a warning or error message. PAI processing is then repeated until no more errors are found.)

2. Triggers PBO for the new screen display (as requested by the user scrolling).

For this PBO, the system updates <scroll-var> to the new value (as set by the user scrolling) and repeats LOOP AT processing for the screen. As a result, a new segment of the internal table is transferred to the screen.

```

***SCREEN FLOW LOGIC***
PROCESS BEFORE OUTPUT:
  LOOP AT INTTAB CURSOR TABCNTL-CURRENT_LINE.
  ENDLOOP.

PROCESS AFTER INPUT.
  LOOP AT INTTAB.
    MODULE MODIFY_INTTAB.
  ENDLOOP.
```

Scrolling and the Scroll Variables

```
***ABAP MODULE***  
MODULE MODIFY_INTTAB.  
  MODIFY INTTAB INDEX COUNT.  
ENDMODULE.
```

If you want to offer other scrolling options besides scroll bars (for example, the F21-F24 scroll buttons in the standard tool bar), you must program these yourself. To do this, you can set <scroll-var> yourself where necessary. The LOOP statement in PBO then updates the screen table accordingly at the next screen display.

Using Table Controls

The structure of table controls is different from step loops. A step loop, as a screen object, is simply a series of field rows that appear as a repeating block. A table control, as a screen object, consists of

- table fields (displayed in the screen)
- a control structure that governs the table display and what the user can do with it

The field list for screen 200 (in transaction TZ60) shows both:

Screen Painter: Display Screen SAPMTZ60 0200 Field List

Screen Edit Goto Utilities Settings System Help

✓ [] [↓] [↶] [↷] [✗] [🔍?] [🔄] [📄] [📄] [📄]

[✎] [🔄] [↔] [📚] [⏪] [⏩] [🖥️] Fullscreen Attribs. for 1 field List texts/templates

Hi	Field name	FType	Ln	Cl	Lg	Vlg	Ht	Roll	Fmt.	I	O	OutO	Dic
+	FLIGHTS	Table	11	3	73	73	9	<input type="checkbox"/>		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
-	SFLIGHT-FLDATE	Text	1	1	5	5	1			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
-	SFLIGHT-PRICE	Text	1	2	9	9	1			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
-	SFLIGHT-CURRENCY	Text	1	3	5	5	1			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
-	SFLIGHT-PLANETYPE	Text	1	4	11	11	1			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
-	SFLIGHT-SEATSMAX	Text	1	5	9	9	1			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
-	SFLIGHT-SEATSOCC	Text	1	6	10	10	1			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Here you see the control structure (the FLIGHTS field with *Ftyp* = *Table*) and the actual table fields (the SFLIGHT fields, with *Ftyp* = *Text* or *I/O*). The attributes for the FLIGHTS field (available when you press *Attribs. for 1 Field*) show the control structure fields you can set statically. Other control structure fields can be set dynamically from the ABAP program.

The following topics provide more information:

Declaring Table Controls in ABAP [Page 771]

[Setting Table Control Attributes \[Page 773\]](#)

Declaring Table Controls in ABAP

Declaring Table Controls in ABAP

When you use a table control in a screen, you must declare both the table control structure and table control fields in your ABAP program. For example, transaction TZ60 has:

TABLES: SFLIGHT.

CONTROLS: FLIGHTS TYPE TABLEVIEW USING SCREEN 200.

The CONTROLS statement defines a control structure of type TABLEVIEW. The system takes the initial values for the structure from the Screen Painter attributes for the given screen.

A TABLEVIEW structure contains the following fields:

Field name	Type	Purpose
FIXED_COLS	integer	Number of immovable columns at the left end of the table. All columns after the fixed columns are movable and can be reordered in the table.
LINES	integer	The number of displayable rows in a table.
TOP_LINE	integer	The row of the table where the screen display starts.
CURRENT_LINE	integer	The row currently being processed inside a loop. This field is an absolute (non-relative) index, with value TOP_LINE + SY_STEPL.-1
LEFT_COL	integer	The left-most non-fixed column. Since the user can scroll the non-fixed part of the display, this field holds the number of the first column to appear after the fixed columns. LEFT_COL gives the absolute (non-relative) value of the column, regardless of whether the user has re-ordered the sequence of the columns.
LINE_SEL_MODE	integer	Enables line selection. Values: 0=None, 1=Single-selection only, 2=Multiple-selection allowed.
COL_SEL_MODE	integer	Enables column selection. Values: 0=None, 1=Single-selection only, 2=Multiple-selection allowed
LINE_SELECTOR	char 1	Indicator: displays line-selection column. This is a column of ordinary check boxes that can be examined in the ABAP program. The system sets a box to X when the user clicks on it.
H_GRID	char 1	Indicator: displays horizontal gridlines
V_GRID	char 1	Indicator: displays vertical gridlines
COLS (OCCURS 10)	TAB_COLUMN	Embedded internal table: one table entry for each column in the table.

The fields in TAB_COLUMN structure describe a single field and its column in the screen table:

Field name	Type	Purpose
------------	------	---------

Declaring Table Controls in ABAP

SCREEN	SCREEN	Embedded SCREEN structure: all the fields from a single row of the SCREEN system table.
INDEX	integer	Current position of the column in the display (in case the user has re-order the sequence of columns).
SELECTED	char 1	Set (by the system) to X when the user clicks on the column.
VISLENGTH	int1	The visible length (in characters) of the field. Lengths of up to 255 characters are allowed.

The *screen* type refers to the SCREEN table, a system table that describes all the objects in a screen..

Setting Table Control Attributes

Setting Table Control Attributes

Use the fields in the TABLEVIEW control structure to set attributes for the screen table at runtime. For example, to set the scrolling variable TOP_LINE, transaction TZ60 uses the statement:

```
FLIGHTS-TOP_LINE = 1.
```

Initially, table control fields get their values from the attribute you specify for the control in the Screen Painter. Thereafter, most table control fields can be set in a variety of ways: by the system, by the ABAP program, or by the user customizing a display. The following table specifies exactly how each field may be set:

Field name	Initialized by	Later values can be set
FIXED_COLS	In ABAP or default=0	In ABAP.
LINES	In ABAP or default=0	In ABAP. Also, set automatically by the system if you use LOOP AT <int-table>.
TOP_LINE	In ABAP or default=1	In ABAP. Also by the system when the user moves the slider on the vertical scroll bar.
CURRENT_LINE	Default=0	No modifications in ABAP allowed. Set by the system during LOOPing. (CURRENT_LINE=TOP_LINE + SY_STEPL-1)
LEFT_COL	In ABAP or default=0.	In ABAP. Also by the system when the user moves the slider on the horizontal scroll bar.
LINE_SEL_MODE	Screen Painter or ABAP program.	In ABAP.
COL_SEL_MODE	Screen Painter or ABAP program.	In ABAP.
LINE_SELECTOR	Screen Painter or ABAP program.	In ABAP.
H_GRID	Screen Painter or ABAP program.	In ABAP.
V_GRID	Screen Painter or ABAP program.	In ABAP.
COLS_SCREEN	Screen Painter, ABAP program, or user customization	In ABAP.
COLS_INDEX	Screen Painter, ABAP program, or user customization	In ABAP.
COL_SELECTED	In ABAP.	In ABAP.

Setting Table Control Attributes

If needed, you can always reset a table control to the initial values specified in the Screen Painter. To do this, use the statement `REFRESH CONTROL <table-control> FROM SCREEN <screen>`. See the online ABAP keyword help for more information.

Example Transaction: Table Controls

Example Transaction: Table Controls

As an example of programming with table controls, look at the code for transaction TZ60. TZ60 (development class SDWA) displays flight information using two screens. In the first, the user specifies a flight connection and requests a display. In the second (screen 200), the transaction displays all flights scheduled for the connection:

The screenshot shows the SAP transaction TZ60 interface. It includes input fields for flight details and a table of flight data.

Flight Details:

- Airline carrier: **AA** AMERICAN AIRLINES
- Flight number: **17**
- From city: **NEW YORK** **JFK**
- Destination: **SAN FRANCISCO** **SFO**
- Flight time: **06:01:00** Distance: **2.572** **MLS**
- Departure: **13:30:00** Arrival: **16:31:00**

Flight Data Table:

	Date	FlgtPrice	Curr.	Plane type	Capacity	Occupied
	30.01.1995	689,66	USD	747-400	660	10
	01.02.1995	689,66	USD	747-400	660	20
	01.06.1995	689,66	USD	747-400	660	38
	04.06.1995	689,66	USD	747-400	660	38

In looking at the code, the main things to notice here are:

- how table data is fetched and passed around

The fields in the table control are declared as database fields (table SFLIGHTS) in the ABAP program. To store several SFLIGHTS rows at a time, the program uses the internal table INT_FLIGHTS.

- During PAI for screen 100, the program selects all flights for the given connection and stores them in internal table INT_FLIGHTS.
- AT PBO for screen 200, the LOOP statement loops through INT_FLIGHTS, calling module DISPLAY_FLIGHTS for each row. DISPLAY_FLIGHTS transfers an INT_FLIGHTS row to the SFLIGHTS work area. The SFLIGHTS fields match the screen table names, so at the end of each loop pass, the system transfers the ABAP values to the screen table fields.

- use of table control fields to direct scrolling

A table control contains information for managing scrolling. This includes fields telling how many table rows are filled and which table row should be first in the screen display into the table. For both step loops and table controls, the system manages scrolling with the scroll bars automatically. This includes scrolling the actual screen

Example Transaction: Table Controls

display, as well as resetting scrolling variables. However, the example program offers scrolling with the F21-F24 keys. In this case, the program must implement these functions explicitly.

- During PAI for screen 100, the program initializes the “first-table-row” variable (field TOP_LINE) to 1.
- At PAI for screen 200, the program resets scrolling variables, especially the TOP_LINE field, if the user has scrolled with the F21-F24 function keys.

Screen Flow Logic

The screen flow logic (PAI only) for screen 200 in transaction TZ60 looks as follows.

```
*-----*
* Screen 200: Flow Logic                               *
*&-----*
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0200.
  LOOP AT INT_FLIGHTS WITH CONTROL FLIGHTS
                                CURSOR FLIGHTS-CURRENT_LINE.
    MODULE DISPLAY_FLIGHTS.
  ENDLOOP.
*
*
PROCESS AFTER INPUT.
  LOOP AT INT_FLIGHTS.
    MODULE SET_LINE_COUNT.
  ENDLOOP.
  MODULE USER_COMMAND_0200.
```

ABAP Code

The following section of the TOP-module code shows the definitions relevant to the following code.

```
*-----*
* INCLUDE MTD40TOP                                     *
*&-----*
PROGRAM SAPMTZ60 MESSAGE-ID A&.
TABLES: SFLIGHT.
<...Other declarations...>
DATA INT_FLIGHTS LIKE SFLIGHT OCCURS 1 WITH HEADER LINE.

DATA: LINE_COUNT TYPE I.

CONTROLS: FLIGHTS TYPE TABLEVIEW USING SCREEN 200.
```

The following PBO module transfers internal table fields to the proper screen table fields at PBO.

```
*&-----*
*&   Module DISPLAY_FLIGHTS OUTPUT
*&-----*
MODULE DISPLAY_FLIGHTS OUTPUT.
  SFLIGHT-FLDATE   = INT_FLIGHTS-FLDATE.
  SFLIGHT-PRICE    = INT_FLIGHTS-PRICE.
  SFLIGHT-CURRENCY = INT_FLIGHTS-CURRENCY.
  SFLIGHT-PLANETYPE = INT_FLIGHTS-PLANETYPE.
```


Example Transaction: Table Controls

```

SFLIGHT-SEATSMAX = INT_FLIGHTS-SEATSMAX.
SFLIGHT-SEATSOCC = INT_FLIGHTS-SEATSOCC.
ENDMODULE.

```

At PAI, the program must loop again, to transfer screen table fields back to the program. During this looping, the program can use SY_LOOPC to find out how many rows were transferred. SY_LOOPC is a system variable telling the total number of rows currently showing in the display.

```

*&-----*
*&   Module SET_LINE_COUNT INPUT
*&-----*
MODULE SET_LINE_COUNT INPUT.
  LINE-COUNT = SY-LOOPC.
ENDMODULE

```

Transaction TZ60 lets the user press function keys (F21-F24) to scroll the display. The system handles scrolling with the scroll bars automatically, but not scrolling with function keys. So PAI for screen 200 must implement function-key scrolling explicitly:

```

MODULE USER_COMMAND_0200 INPUT.
  CASE OK_CODE.
    *   WHEN 'CANC'...
    *   WHEN 'EXIT'...
    *   WHEN 'BACK'...
    *   WHEN 'NEW'...
    *   WHEN 'P--'.
      CLEAR OK_CODE.
      PERFORM PAGING USING 'P--'.
    WHEN 'P-'.
      CLEAR OK_CODE.
      PERFORM PAGING USING 'P-'.
    WHEN 'P+'.
      CLEAR OK_CODE.
      PERFORM PAGING USING 'P+'.
    WHEN 'P++'.
      CLEAR OK_CODE.
      PERFORM PAGING USING 'P++'.
  ENDCASE.
ENDMODULE.

```

The PAGING routine causes the system to scroll the display by re-setting the table control field TOP_LINE to a new value. The TOP_LINE field tells the LOOP statement where to start looping at PBO.

```

*&-----*
*&   Form PAGING
*&-----*
FORM PAGING USING CODE.
  DATA: I TYPE I.
         J TYPE I.

  CASE CODE.
    WHEN 'P--'. FLIGHTS-TOP_LINE = 1.
    WHEN 'P-'. FLIGHTS-TOP_LINE = FLIGHTS-TOP_LINE - LINE_COUNT.
               IF FLIGHTS-TOP_LINE LE 0.
                 FLIGHTS-TOP_LINE = 1. ENDIF.

```

Example Transaction: Table Controls

```

    WHEN 'P+'.    I = FLIGHTS-TOP_LINE + LINE_COUNT.
                  J = FLIGHTS-LINES - LINE_COUNT + 1.
                  IF J LE 0.
                    J = 1. ENDIF.
                  IF I LE J.
                    FLIGHTS-TOP_LINE = I.
                  ELSE. FLIGHTS-TOP_LINE = J.
                  ENDIF.

    WHEN 'P++'.

    FLIGHTS-TOP_LINE = FLIGHTS-LINES - LINE_COUNT + 1.
    IF FLIGHTS-TOP_LINE LE 0.
      FLIGHTS-TOP_LINE = 1. ENDIF.

  ENDCASE.
ENDFORM.

```

Using Step Loops

Using Step Loops

A step loop is a repeated series of field-blocks in a screen. Each block can contain one or more fields, and can extend over more than one line on the screen.

Step-loops as structures in a screen do not have individual names. The screen can contain more than one step-loop, but if so, you must program the LOOP...ENDLOOPs in the flow logic accordingly. The ordering of the LOOP...ENDLOOPs must exactly parallel the order of the step-loops in the screen. The ordering tells the system which loop processing to apply to which loop. Step-loops in a screen are ordered primarily by screen row, and secondarily by screen column.

Transaction TZ61 (development class SDWA) implements a step-loop version of the table you saw in transaction TZ60. Screen 200 for TZ61, shows the following step loop:

Airline carrier	AA	AMERICAN AIRLINES				
Flight number	64					
From city	SAN FRANCISCO	SFO				
Destination	NEW YORK	JFK				
Flight time	05:21:00	Distance	2.572	MLS		
Departure	09:00:00	Arrival	17:21:00			

Flights						
Flgt date	FlgtPrice	Curr.	Plane type	Max. capacit	Occupied	
30.01.1995	689,66	USD	A319	220	10	
01.02.1995	689,66	USD	A319	220	20	
01.06.1995	689,66	USD	A319	220	38	
04.06.1995	689,66	USD	A319	220	38	

See the TZ61 code for a demonstration of how to use step loops.

Static and Dynamic Step Loops

Step-loops fall into two classes: static and dynamic. Static step-loops have a fixed size that cannot be changed at runtime. Dynamic step-loops are variable in size. If the user re-sizes the window, the system automatically increases or decreases the number of step-loop blocks displayed. In any given screen, you can define any number of static step-loops, but only a single dynamic one.

You specify the class for a step-loop in the Screen Painter. Each loop in a screen has the attributes *Looptype* (*fixed*=static, *variable*=dynamic) and *Loopcount*. If a loop is fixed, the *Loopcount* tells the number of loop-blocks displayed for the loop. This number can never change.

Programming with static and dynamic step-loops is essentially the same. You can use both the LOOP and LOOP AT statements for both types.

Looping in a Step Loop

When you use LOOP AT <internal-table> with a step loop, the system automatically displays the step loop with vertical scroll bars. The scroll bars, and the updated (scrolled) table display, are managed by the system.

Use the following additional parameters if desired:

- FROM <line1> and TO <line2>

These parameters limit the parts of the internal table that can be displayed or processed in the step loop. If you use one or both of these, declare <line1> and <line2> in ABAP with LIKE SY-TABIX. If you don't use them, the system simply uses the beginning and/or end of the internal table as the limit.

- CURSOR <scroll-var>

The CURSOR parameter tells which internal table row should be the first in the screen display. <Scroll-var> is a local program variable that can be set either by your program or automatically by the system. The value of <scroll-var> is absolute with respect to the internal table (that is, not relative to the FROM or TO values).

The CURSOR <scroll-var> parameter is required in the PBO event to tell the system where to start displaying.

Checking User Authorizations

Checking User Authorizations

Much of the data in an R/3 system has to be protected so that unauthorized users cannot access it. Therefore the appropriate authorization is required before a user can carry out certain actions in the system. When you log on to the R/3 system, the system checks in the user master record to see which transactions you are authorized to use. An authorization check is implemented for every sensitive transaction.

If you wish to protect a transaction that you have programmed yourself, then you must implement an authorization check. This means you have to:

- allocate an authorization object in the definition of the transaction;
- program an AUTHORITY-CHECK.

AUTHORITY-CHECK OBJECT <authorization object>

 ID <authority-field1> FIELD <field-value1>

 ID <authority-field2 > FIELD <field-value2>

 ...

 ID <authority-fieldn> FIELD <field-valuen>.

The **OBJECT** parameter specifies the authorization object.

The **ID** parameter specifies an authorization field (in the authorization object).

The **FIELD** parameter specifies a value for the authorization field.

The authorization object and its fields have to be suitable for the transaction. In most cases you will be able to use the existing authorization objects to protect your data. But new developments may require that you define new authorization objects and fields.

The following topics provide more information:

[Defining an Authority Check \[Page 782\]](#)

[Defining Authorization Objects \[Page 785\]](#)

[Defining Authorization Fields \[Page 786\]](#)

Defining an Authority Check

As an example of handling authority checks, start with transaction tz80, one of the example transactions delivered with each system.

This example from the Flight Reservation system consists of two screens. In the first screen the user can enter flight identifiers and request flight details (by pressing the *Display* pushbutton) or press the *Change* pushbutton to change the flight data.

The S_CARRID authorization object is allocated to the transaction tz80. This authorization object contains two fields. Generic values can be entered in the first field *Airline carrier*. In the second field *Activity* you can select between create(01), change(02) and display(03).

When you have programmed a new transaction, you can specify an authorization object in the definition of the transaction code. Next to the entry field for the authorization object is a pushbutton labelled *Values*. You use this pushbutton to enter the field values for the authorization object that should be checked when the transaction is started.

Defining an Authority Check

The screenshot shows the SAP transaction T281 'Authorizations'. The main fields are:

- Transaction code: T281
- Transaction text: Authorizations
- Program: SAPMT280
- Screen number: 100
- Authorization object: S_CARRID

On the right, a pop-up window titled 'Values of check object' displays the following data:

Fields	Values
ACTVT	03
CARRID	*

At the bottom of the pop-up are green and red checkmark buttons.

A user wanting to call transaction tz80 needs an authorization in his user master record for the S_CARRID authorization object. It has to contain the value *display* (03) for the ACTVT field and a value for the *Airline carrier* field.

A more sophisticated authorization check is possible using the AUTHORITY-CHECK statement.

Within the transaction tz80 you can display and change flight data only if you have the appropriate authorization for the S_CARRID authorization object in your user master record.

```
*&-----*
*&  Module  USER_COMMAND_0100 INPUT
*&-----*

MODULE USER_COMMAND_0100 INPUT.
  CASE OK_CODE.
    WHEN 'SHOW'.
      AUTHORITY-CHECK OBJECT 'S_CARRID'
                        ID 'CARRID' FIELD '*'
                        ID 'ACTVT' FIELD '03'.

      IF SY-SUBRC NE 0. MESSAGE E009. ENDIF.
      MODE = CON_SHOW.
      SELECT SINGLE * FROM SPFLI
                        WHERE CARRID = SPFLI-CARRID
                        AND  CONNID = SPFLI-CONNID.

      IF SY-SUBRC NE 0.
        MESSAGE E005 WITH SPFLI-CARRID SPFLI-CONNID.
      ENDIF.
      CLEAR OK_CODE.
      SET SCREEN 200.
    WHEN 'CHNG'.
      AUTHORITY-CHECK OBJECT 'S_CARRID'
                        ID 'CARRID' FIELD '*'
                        ID 'ACTVT' FIELD '02'.

      IF SY-SUBRC NE 0. MESSAGE E010. ENDIF.
      MODE = CON_CHANGE.
      SELECT SINGLE * FROM SPFLI
                        WHERE CARRID = SPFLI-CARRID
                        AND  CONNID = SPFLI-CONNID.

      IF SY-SUBRC NE 0.
        MESSAGE E005 WITH SPFLI-CARRID SPFLI-CONNID.
      ENDIF.
```

```
    OLD_SPFLI = SPFLI.  
    CLEAR OK_CODE.  
    SET SCREEN 200.  
  ENDCASE.  
ENDMODULE.          " USER_COMMAND_0100 INPUT
```

If you select the *Display* function in the transaction, the `AUTHORITY-CHECK` statement checks for the value '03' in the `ACTVT` field in the `S_CARRID` authorization object. If you select the *Change* function, then the value '02' has to be present in the `ACTVT` field.

You can find more information about setting up authorizations for user master records in [BC Users and Authorizations \[Ext.\]](#).

Defining Authorization Objects

Defining Authorization Objects

If there is no suitable authorization object for a transaction that needs protecting, you can set up an authorization object and fields yourself.

Authorization objects can be defined from the *ABAP Development Workbench* menu by selecting *Development* → *Other tools* → *Authorization objs* → *Objects*.

Each authorization object must be assigned to an object class. These classes are organized according to the components of the system. You can assign a new object to the object class of the component with which you are working or create a new class. If you do so, select class names that begin with Y or Z to avoid conflicts with SAP names.

To define a new authorization object, enter a unique object name and the fields that belong to the object.

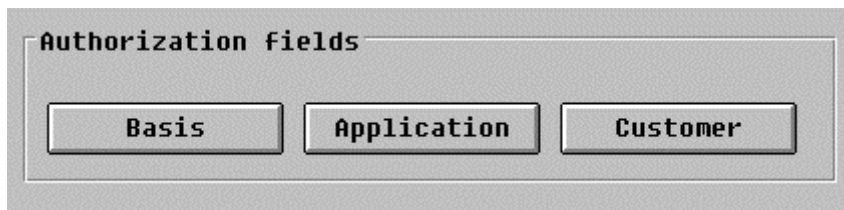
The second sign of the object name shouldn't be an underscore. The underscore is used by SAP objects and might be overwritten when new releases are installed.

Defining Authorization Fields

Each object consists of up to ten fields, which the programmer can define. Programmers can define authorization fields in the *ABAP Development Workbench* menu by selecting *Development* → *Other tools* → *Authorization objs* → *Fields*.


Before you define a field, check that you have defined the data dictionary structure ZAUTHCUST. This structure is required for customer defined authorization fields. For further information on defining this structure, see [BC Users and Authorizations \[Ext.\]](#).

To define new authorization fields, enter the name of the field and associate an ABAP dictionary data element with it. To avoid having your fields deleted when you install a new release, define your fields only with the customer function in field maintenance. The customer function automatically saves your fields in the ZAUTHCUST structure.



You can often reuse fields defined by SAP in your own authorization objects. It is not required that you define your own fields if you define a new authorization object. For example you can use the SAP field ACTVT in your own authorization objects to represent a wide variety of actions in the system.

Programming Field- and Value-Help

F1 or  provide the user with help text for the field currently containing the cursor. F4 or the combo box, where available, offer a list of values which may be input in this field. The user can place the cursor in the list on the chosen value and press F2 or double-click with the mouse to copy this value into the field.

The information that is output using this mechanism is held in the ABAP repository. In dialog programs the screen processor automatically displays the help text for the data element underlying each field. The F4 list of possible values often refers to the fixed set of values valid for a domain or to data in the related value table. The standard list is frequently adequate for applications you develop yourself too. However, you also have the option of setting up documentation and lists of possible values that are more detailed or simply different, as appropriate for your dialog program.


You can program help texts and possible values lists using the PROCESS ON HELP-REQUEST (POH) and PROCESS ON VALUE-REQUEST (POV) events.

These events can be executed in the screen process logic in addition to the PROCESS BEFORE OUTPUT and PROCESS AFTER INPUT events.

- PROCESS ON HELP-REQUEST:

Syntax

```
PROCESS ON HELP-REQUEST.  
  FIELD <field> MODULE <module>.  
  FIELD <field> MODULE <module>...
```

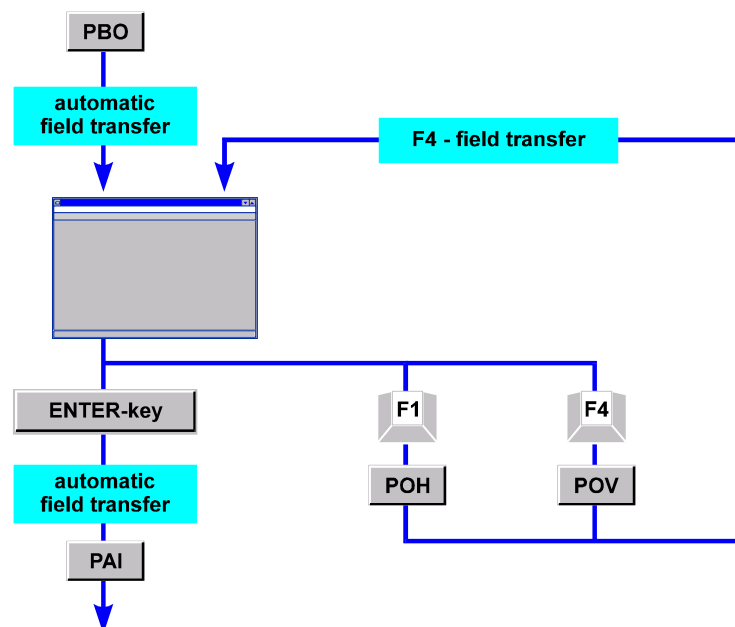
A call to a module is assigned to a screen field through a FIELD statement. If you press F1 or  then the POH module belonging to the field on which the cursor is positioned will be executed.

- PROCESS ON VALUE-REQUEST:

Syntax

```
PROCESS ON VALUE-REQUEST.  
  FIELD <field> MODULE <module>.  
  FIELD <field>          MODULE <module>...
```

The POV event occurs when the user presses F4 or activates the combo box in a screen field. If there is a FIELD statement for this field then the specified POV module is called instead of the help processor.



When a screen is called, the PROCESS BEFORE OUTPUT event is executed. An automatic field transport to the screen then occurs. If you select the F1 help or the F4 values list in the screen the system sends the dynpro once again, but without executing the PBO event and without executing a field transfer for all fields to the dynpro. There will merely be a transport after POV for the contents of the field on which the cursor is placed and which were selected via F2 or a double-click.

If you require additional field transports for the POV processing, then you can use the DYNP_VALUES_READ and DYNP_VALUES_UPDATE function modules.

Other function modules simplify the context-specific programming of help texts and input values.

The following topics provide more information:

[Customizing F4-Value-Request \[Ext.\]](#)

[Example: Programming Value-Help \[Ext.\]](#)

[Customizing F1-Help \[Ext.\]](#)

[Function Modules for Field-Help \[Ext.\]](#)

User-specific F1 Help

The ABAP Workbench provides many different ways of making the F1 help context-sensitive:

- Data element documentation

You can extend the F1 Help by creating a data element supplement in the ABAP Dictionary. In the Screen Painter, open the field list and place the cursor on the field for which you want to write extra documentation. Choose *Goto → Documentation → Data element doc*. Here, you can extend the existing field help. The extended form of the help for this field is then displayed systemwide.

- Data element supplement in the Screen Painter

If you only want to extend the field help in one particular transaction, position the cursor on a field in the field list and choose *Goto → Documentation → DE supplement doc*. A dialog box appears in which the system proposes a number as the identified for the data element supplement. You can then write your additional documentation, which appears as well as the help text stored in the ABAP Dictionary.

- Using the event PROCESS ON HELP-REQUEST

If the event PROCESS ON HELP-REQUEST (POH) does not occur in the flow logic of a screen, the field documentation is displayed as it occurs in the ABAP Dictionary. However, if the POH event does exist, it is processed whenever the user calls the F1 help for a particular field. The following coding example displays a data element supplement for a particular field:

```
PROCESS ON HELP-REQUEST.  
  FIELD XY WITH 'Number'.
```

You can also use variables for the data element supplement as follows:

```
PROCESS ON HELP-REQUEST.  
  FIELD XY WITH <Variable>.
```

The data element supplement specified in the variable <variable> is then displayed for field XY.

If you cannot determine the data element supplement until the F1 event itself, you can use a module to find out its identifier and then write it into a variable as follows:

```
PROCESS ON HELP-REQUEST.  
  FIELD XY MODULE XYZ WITH <Variable>.
```

Context-sensitive Value Help

Value help is an integral function within the R/3 System. It provides help for users when they do not know (or only partially know) the value that they want to enter in a field.

To ensure that this function behaves in the same way throughout the system, there are several standard mechanisms within the R/3 System that are used to define value help for a screen field. In many cases, these standard procedures are sufficient to produce the desired results.

To assign value help to a field, you can either create it explicitly for the current screen, or use the attributes of the ABAP Dictionary field that underlies the screen field.

Linking Value Help to a Screen

The system contains the following mechanisms:

- User-defined input help in the PROCESS ON VALUE-REQUEST (POV) section of the flow logic.

A module assigned to a field in the Screen Painter has the highest priority when the system is working out what to display as F4 help.

Example:

```
PROCESS ON HELP-REQUEST.  
  FIELD SPFLI-AIRPFROM  
    MODULE VAL_REQ_AIRPORT.
```

The PROCESS ON VALUE-REQUEST event allows you to program your own value help. If you specify a FIELD statement for the input field after PROCESS ON VALUE-REQUEST, the corresponding module is executed.

After this keyword, you can define what happens when the user presses F4.

Development class SDWA contains an example program demonstrating user-defined input help with PROCESS ON VALUE-REQUEST.

We recommend that, from Release 4.0, you use search helps as an alternative to PROCESS ON VALUE-REQUEST, even though the POV method is still possible. Defining search helps is much easier than PROCESS ON VALUE-REQUEST, since the system takes over some of the standard operations, such as getting field contents from the screen. It also ensures that the F4 help has a uniform look and feel throughout the system. Furthermore, it means that you do not have to reassign input help to fields on each screen.

For further information about search helps, see the online documentation for the *ABAP Dictionary*.

Despite the introduction of search helps (and search help exits), there are still cases in which you need to use parts of the standard F4 functions directly. Within user-defined input help, you can use function modules that support not only search helps, but also the other forms of input help. These all have the prefix F4IF_.

- Automatic input help for input check "FIELD... SELECT..." in the flow logic.

Context-sensitive Value Help

The "FIELD... SELECT..." construction only allows the user to input values that are contained in a check table. You can place the values already entered into further screen fields or into global fields.

For more information about how to use the SELECT statement, refer to the keyword documentation in the flow logic editor.

The selection and formatting of the hit list can be affected by a text table or a search help for the check table.

- Automatic input help for the input check "FIELD... VALUES..." in the flow logic.

Just like "FIELD... SELECT...", "FIELD... VALUES..." restricts the permitted input values to certain values or intervals. The values and intervals are displayed in the input help.

If you use the keyword NOT, the input help is no longer available.

Linking Value Help to the ABAP Dictionary

The system contains the following mechanisms:

- Direct link of a search help to the ABAP Dictionary field.
- Value help using a check table.

The value help is provided for all fields that are checked against the table. If the check table has a search help, it is displayed. Otherwise, only the key values for the check table are displayed. If there is a text table for the check table, the texts are displayed in the logon language.

- Linking a search help to a data element

For details about how to link a search help to an ABAP Dictionary element, refer to the *Linking a Search Help with a Screen Field* section of the *ABAP Dictionary* documentation.

- Fixed value help for fields that refer to a domain with fixed values.

The fixed values defined in the domain are displayed. When the user selects a value, it is copied directly into the screen field. You can enter fixed values in the ABAP Dictionary; choose *Goto* → *Fixed values*.

- Fixed input help for fields with a certain elementary type.

The system automatically provides special input help for date (DATS) and time (TIMS) fields. These allow users to select values from calendar or configurable clock.

As a rule, value help programmed from a screen takes precedence over that in the ABAP Dictionary. If more than one type of ABAP Dictionary input help conflict, the one listed higher in the above list takes priority.

The technique of displaying a value table as input help has been withdrawn in Release 4.0, since it sometimes led to unexpected results, especially where the value table had several key fields. It was not possible to restrict the other key fields, which meant that the environment of the field was not considered, as is normal with check tables.

If you used this kind of search help and found it useful, you can recreate it by attaching a search help to the data elements of the domain. If you used to use a help view, use this as the selection method. Otherwise, you can enter the value table as the selection method.

Using Search Helps to Adapt the Value Help

If you need more flexibility than standard search helps provide, you can use a search help exit to call a function module in which you design the value help you need.

A search help exit is a function module, provided by the developer, which you can call at predetermined points during F4 processing. It allows you to affect how the help display is processed, or replace steps in it with your own.

The function module can change the attributes of the search help, the selection options that are used to preselect the hit list, the hit list itself, and also the subsequent steps in the F4 processing.

The function module that you build into your search help as a search help exit must have the same interface as the function module F4IF_SHLP_EXIT_EXAMPLE. It may also have further optional parameters (in particular, any number of EXPORTING parameters).

The appropriate place to call the search help exit is always before the interaction steps when you call the value help. This is the point at which the input help is displayed for the user, and thus where only actions relevant to the user are possible. The following events are defined at which you can call the search help exit:

- Before the elementary search help is displayed (in the case of collective search helps). Event SELONE.

For details of the events, refer to the documentation for function module F4IF_SHLP_EXIT_EXAMPLE.

This makes it possible to make the search help dependent on the transaction, on other system variables, or even on the state of the radio buttons on the screen.

(This is the only event in which the search help exit is called for collective search helps. All other events call the search help exit for the selected elementary search help.)

- After an elementary search help has been selected (event PRESEL1).
- Before the value selection dialog box is displayed (event PRESEL).

This enables you to change the contents of the dialog box, or to suppress it altogether.

- Before the value selection (event SELECT).

The value selection involves no user interaction. However, it is a step that must often be taken over completely by the function module because the data cannot be selected easily from a table or view.

Context-sensitive Value Help

- Before the hit list is displayed (event DISP).

This makes it possible, for example, to hide certain entries or fields of a table from the user depending on his or her authorization.

To be able to work effectively in any of the above steps, the function module needs to receive all of the internal information that depends on a search help and the tables of the F4 processor as its parameters. For more information, see the function module documentation for F4IF_SHLP_EXIT_EXAMPLE.

Certain search help functions are requested repeatedly in similar ways. One example of this is the possibility to set the search help that will be used dynamically. Standard function modules have been written for these cases, which you can use either directly as search help exits, or call from within a search help exit. Such function modules all have the prefix F4UT_.

Selection Screens

Working with Selection Screens

Selection screens are one of the three types of screen in the R/3 System, along with dialog screens and lists. You use them whenever you want the user to enter either a single value for a field or fields, or to enter selection criteria.

[What are Selection Screens? \[Page 796\]](#)

[Defining Selection Screens \[Page 800\]](#)

[Calling Selection Screens \[Page 846\]](#)

[Using Selection Criteria in Programs \[Page 857\]](#)

What are Selection Screens?

Function in the R/3 System

ABAP programs use screens to obtain input values from users. The most general type of screen is a dialog screen, which you create using the Screen Painter and Menu Painter (see [BC - ABAP Development Workbench: Tools \[Ext.\]](#)). These tools allow you to create screens for data input and output. Each dialog screen requires its own flow logic.

You often use screens purely for **data input**. In these cases, you can use a selection screen. Selection screens are a standardized user interface in the R/3 System for entering single field values (input parameters) and complex selection restrictions (selection criteria). Input parameters are primarily used to control the program flow, while users can enter selection criteria to restrict the amount of data read from the database. You can pre-define sets of entry values for any selection screen. These are called variants (see [Pre-Setting Selections Using Variants \[Page 864\]](#)). Texts displayed on selection screens are language-dependent text elements (see [Selection Texts \[Page 154\]](#)). If you call an executable program (report) using the SUBMIT statement in another ABAP program (see [Calling Executable Programs \(Reports\) \[Page 1113\]](#)), the selection screen fields also serve as a data interface.

Defining and Calling Selection Screens

You define selection screens in ABAP programs using simple ABAP statements which allow you to create input fields, checkboxes and radio buttons and control the screen layout. If you want to create a screen exclusively for data input, you do not need to create it using the normal dialog programming tools. When you create a selection screen, the system automatically carries out the tasks of the Screen Painter and Menu Painter.

The rules for calling and defining selection screens in ABAP programs depend on the program type:

- Executable program (type 1) without logical database
You can use a single **standard selection screen** and as many **user-defined selection screens** as you wish. The system automatically calls the standard selection screen when the program starts. However, you need to call user-defined selection screens yourself using the CALL SELECTION-SCREEN statement. The standard selection screen is always screen 1000. You can assign any number apart from 1000 to the user-defined selection screens.
- Executable program (type 1) with logical database
The **standard selection screen** for an executable program linked to a logical database is made up of the logical database selections and the program selections. For more information about the relationship between the standard selection screen and logical databases, see [Standard Selection Screens and Logical Databases \[Page 797\]](#).
- Module pools (type M) and function modules (type F)

You can only use user-defined selection screens in module pools and function modules. These can have any number apart from 1000. You can only call a selection screen from a function module using the CALL SELECTION-SCREEN statement. In a module pool, you can use a selection screen as the initial screen for a transaction.

Standard Selection Screens and Logical Databases

Standard Selection Screens and Logical Databases

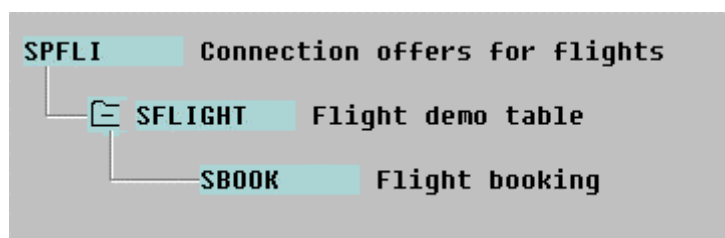
Logical database programs normally also contain statements defining selection screens. Whenever you write an executable program (report) which has a logical database specified in its attributes, the selection screen will automatically contain the relevant input fields when you run the program. A particular feature of selection screens with logical databases are its free restrictions, which allow the user to make field selections dynamically which are not a static part of the logical database program.

Static Selections in Logical Databases

If an executable program (report) is linked to a logical database, its selection screen is made up of the logical database selections and the program selections. The selections for the logical database are defined in its selection part, using the same ABAP statements as the program selections (see [Editing Selections \[Page 1278\]](#)).

The input fields for the logical database selections which actually depend on the database tables you declare in your program with the TABLES statement. For further information about logical database selections, see [Features and Maintenance of Logical Databases \[Page 1246\]](#).

The logical database F1S is attached to the following executable program. F1S has the structure:



Assume the following program:

```
REPORT SAPMZTST.
```

```
TABLES SPFLI.
```

After starting SAPMZTST, the selection screen appears as follows:

These are the input fields for selection criteria and parameters for the columns of the database table SPFLI. The statements that define this screen (SELECT-OPTIONS and PARAMETERS) are coded in the logical database program.

Now, assume the following report program:

```
REPORT SAPMZTST.
```

```
TABLES SBOOK.
```

Standard Selection Screens and Logical Databases

After starting SAPMZTST, the selection screen appears as follows:

Carrier ID	<input type="text"/>	To	<input type="text"/>	
From	<input type="text"/>			
To	<input type="text"/>			
Departure date	<input type="text"/>	To	<input type="text"/>	
Booking ID	<input type="text"/>	To	<input type="text"/>	

The system not only displays the input fields of the selection criteria connected to the database table SBOOK, but also those criteria connected to tables SPFLI and SFLIGHT.

The logical database uses its own selections to restrict the amount of data read from the database. For an example of this, see [Example of a Logical Database \[Page 1265\]](#). You should take advantage of the selection criteria defined in the logical database, and only use program-specific selections if those in the logical database are insufficient for your requirements. On the selection screen, the input fields for the program-specific selections appear below the selections for the logical database.

Free Restriction of Logical Databases

Logical databases can provide a feature that allows the report user to specify dynamic selections, which are not coded with SELECT-OPTIONS statements in the logical database program. If a logical database provides dynamic selections, the user accesses these selections by clicking *User selections* in the application toolbar of the selection screen. A new selection screen or a screen displays, where the user can select the database fields that he wants to specify selection criteria for.

For the logical database F1S, the screen for dynamic selections might appear as follows:

Dynamic selections				
Connection number	<input type="text"/>	To	<input type="text"/>	


Dynamic selections limit the database accesses of logical database programs by using dynamic WHERE conditions (see [Specifying Conditions for Line Selection at Runtime \[Page 563\]](#)).

The possibility of dynamic selections must be coded in the logical database program (see [Editing Selections \[Page 1278\]](#)). If a logical database allows for dynamic selections (*User selections* appears in the application toolbar), you can use the ABAP Development Workbench to define which database tables and columns you want to allow the user to define dynamic selections (a selection view) for. To find out which database tables offer dynamic selections, choose *Tools* → *ABAP Development Workbench* → *Development* → *Programming environ.* → *Logical databases*

Standard Selection Screens and Logical Databases

→ *Extras* → *Dynamic selections*. On the next screen, you then see a list of the names of these database tables.

For the logical database F1S, only the database table SPFLI provides dynamic selections:

Tables for dynamic selections	
Table	Description
SPFLI	Connection offers for flights
	

Defining Selection Screens

You include the statements to define selection screens in the declaration part of your ABAP program.

In defining selection screens, you must distinguish between standard selection screens for executable programs (reports) and user-defined selection screens for all types of program.

[Standard and User-defined Selection Screens \[Page 801\]](#)

You use the following three selection statements to define your own selection screens:

[PARAMETERS - Defining Input Fields for Variables \[Page 803\]](#)

[SELECT-OPTIONS - Defining Selection Criteria \[Page 815\]](#)

[SELECTION-SCREEN - Formatting \[Page 832\]](#)

Standard and User-defined Selection Screens

Standard and User-defined Selection Screens

To create a user-defined selection screen, use the following statements:

Syntax

```
SELECTION-SCREEN BEGIN OF <numb> [TITLE <title>] [AS WINDOW].
```

...

```
SELECTION-SCREEN END OF <numb>.
```

All of the PARAMETERS, SELECT-OPTIONS and SELECTION-SCREEN statements occurring **between** these two statements define a **user-defined selection screen** with screen number <numb>.

You may only use the PARAMETERS, SELECT-OPTIONS and SELECTION-SCREEN statements outside the above BEGIN OF... END OF enclosure in executable programs (reports). All unenclosed occurrences of these statements in an executable program make up the **standard selection screen**. For the sake of clarity, you should group together all of the statements making up the standard selection screen before defining further selection screens.

The number <numb> is the screen number of the user-defined selection screen, which can be any four-digit number apart from 1000. This is reserved for the standard selection screen.

Ensure that you do not accidentally assign a number to a dialog screen which is already in use for one of your selection screens.

The TITLE <title> addition allows you to specify a title for a user-defined selection screen. <Title> can either be a static text symbol or a dynamic character field. If you use a character field, you must not define it using the DATA statement - the system generates it automatically. You can fill the character field dynamically during the INITIALIZATION event. The title of a standard selection screen is always the same as that of the executable program.

You can call a user-defined selection screen as a modal dialog box using the AS WINDOW addition. You define the window when you call the screen (see [Calling Selection Screens \[Page 846\]](#)). When you use the AS WINDOW addition, warnings and error messages associated with the selection screen are also output as modal dialog boxes, and not in the status line of the selection screen.

If you defined more than one selection screen in a program, you can re-use elements of one selection screen in another using the following statement:

Syntax

```
SELECTION SCREEN INCLUDE      BLOCKS <block> |  
    PARAMETERS <p> |  
    SELECT-OPTIONS <selcrit> |  
    COMMENT <comm> |  
    PUSH-BUTTON.<push>.
```

You can use any of the following as re-useable elements, as long as they have already been declared in another selection screen:

- A block with name <block> (see [Creating Blocks of Elements \[Page 841\]](#))
- A parameter with name <p> (see [Basic Form of the PARAMETERS Statement \[Page 804\]](#))

- Selection criteria with name <selcrit> (see [Basic Form of the SELECT-OPTIONS Statement \[Page 821\]](#))
- Comments with name <comm> (see [Comments \[Page 836\]](#))
- Pushbuttons with name <push> (see [Creating Pushbuttons on the Selection Screen \[Page 844\]](#))

```
REPORT SELSCREENDEF.  
...  
PARAMETERS PAR1....  
SELECT-OPTIONS SEL1 FOR....  
...  
SELECTION-SCREEN BEGIN OF SCREEN 500 AS WINDOW.  
  PARAMETERS PAR2....  
  SELECT-OPTIONS SEL2 FOR....  
  ...  
SELECTION-SCREEN END OF SCREEN 500.  
SELECTION-SCREEN BEGIN OF SCREEN 600 TITLE TEXT-100.  
  SELECTION-SCREEN INCLUDE: PARAMETERS PAR1,  
    SELECT-OPTIONS SEL1.  
  PARAMETERS PAR3....  
  SELECT-OPTIONS SEL3....  
  ...  
SELECTION-SCREEN END OF SCREEN 600.
```

Three selection screens - the standard selection screen and two user-defined selection screens - are defined in this program. The program must have type 1 in order for a standard selection screen to be generated. Selection screen 500 is defined to be called as a modal dialog box. Selection screen 600 contains text symbol 100 as its title, and uses elements PAR1 and SEL1 from the standard selection screen.

For further examples, see [Calling Selection Screens \[Page 846\]](#).

PARAMETERS - Defining Input Fields for Variables

If you want to enable the user to enter values for variables on the selection screen, you must define the variables using the PARAMETERS statement. Each variable defined with the PARAMETERS statement appears as an input field on the selection screen.

You can use the PARAMETERS statement for both standard and user-defined selection screens (see [Standard and User-defined Selection Screens \[Page 801\]](#)). The variants of the PARAMETERS statement which are important for program-specific selections are explained in the following sections:

[Basic Form of the PARAMETERS Statement \[Page 804\]](#)

[Assigning Default Values to Parameters \[Page 806\]](#)

[Suppressing Display of Parameters \[Page 807\]](#)

[Allowing Parameters to Accept Upper and Lower Case \[Page 808\]](#)

[Making Parameters Required Input Fields \[Page 809\]](#)

[Creating Checkboxes on the Selection Screen \[Page 810\]](#)

[Creating Radio Button Groups on the Selection Screen \[Page 811\]](#)

[Using Default Values from SAP Memory \[Page 812\]](#)

[Assigning Matchcode Objects to Parameters \[Page 813\]](#)

[Assigning Parameters to a Modification Group \[Page 814\]](#)

There are further variants of the PARAMETERS statement which you can use to define database-specific selections in logical databases. For further information, see the keyword documentation and [Features and Maintenance of Logical Databases \[Page 1246\]](#)).

Basic Form of the PARAMETERS Statement

You declare fields with the PARAMETERS statement the same way as you declare fields with the DATA statement (see [The DATA Statement \[Page 119\]](#)). Fields declared with the PARAMETERS statement are called parameters. Normally, an input field for each parameter appears on the corresponding selection screen. Values typed into these input fields by the report user are assigned to the corresponding parameters while the system processes the selection screen.

To declare a parameter and its data type, use the PARAMETERS statement as follows:

Syntax

PARAMETERS <p>[(<length>)] <type> [<decimals>].

This statement creates a parameter <p>. The additions <length>, <type>, and <decimals> are the same as for the DATA statement (see

The position of the statement in the declaration part of the program determines the selection screen to which the input field belongs (see [Standard and User-defined Selection Screens \[Page 801\]](#)).

Parameters cannot have data type F. The data type F is not supported on the selection screen.

The system displays an input field on the corresponding selection screen for <p>. You can change the comments on the left side of the input fields by using text elements as described in [Selection Texts \[Page 154\]](#).

```
REPORT SAPMZTST.
TABLES SPFLI.
PARAMETERS: WORD(10) TYPE C,
             DATE TYPE D,
             NUMBER TYPE P DECIMALS 2,
             CONNECT LIKE SPFLI-CONNID.
```

This example creates four parameters for the standard selection screen: a character field, WORD, of length 10; a date field, DATE, of length 8; a packed number field, NUMBER, with two decimals; and a field, CONNECT, which refers to the ABAP Dictionary structure SPFLI-CONNID. When the user starts the executable program (report) SAPMZTST, input fields for the four declared fields will appear on the standard selection screen as follows:

WORD	<input type="text"/>
DATE	<input type="text"/>
NUMBER	<input type="text"/>
CONNECT	<input type="text"/>

You can use parameters, for example, to control the program flow, or in connection with SELECT statements to enable the report user to determine selection criteria for database accesses (see [Choosing the Lines to be Read \[Page 559\]](#)).

Basic Form of the PARAMETERS Statement

```

/
TABLES SPFLI.
PARAMETERS: LOW LIKE SPFLI-CARRID,
             HIGH LIKE SPFLI-CARRID.
SELECT * FROM SPFLI WHERE CARRID BETWEEN LOW AND HIGH.
.....
ENDSELECT.
```

In this example, the system reads all rows from the database table SPFLI, where the contents of field CARRID are between the limits LOW and HIGH. The report user can enter the fields LOW and HIGH on the selection screen.

If you use the PARAMETERS statement to define WHERE conditions, you must code all possible selection options into your executable program (report). For complex selections, it is beneficial to work with selection criteria that are stored in internal tables instead (see **Error! Reference source not found.**bc290e.doc041).

Assigning Default Values to Parameters

To assign default values for input fields to be displayed on the selection screen, you use the DEFAULT option of the PARAMETERS statement. The syntax is as follows:

Syntax

```
PARAMETERS <p>..... DEFAULT <f>.....
```

<f> can be either a literal or a field name. If you specify a field name, then the system processes the contents of this field as a default value. The report user can change the default values on the selection screen.

The system transports the default values before the event INITIALIZATION (see [INITIALIZATION \[Page 1213\]](#)) to the parameters. Therefore, you should use field names instead of literals as default values only for fields, which are already filled when the user starts the program. Such fields are, for example, system fields (see [System-Defined Data Objects \[Page 117\]](#)).

```
REPORT SAPMZTST.  
  
PARAMETERS: VALUE TYPE I DEFAULT 100,  
             NAME LIKE SY-UNAME DEFAULT SY-UNAME,  
             DATE LIKE SY-DATUM DEFAULT '19950627'.
```

If the name of the program user is FRED, the selection screen appears as follows:

VALUE	<input type="text" value="100"/>
NAME	<input type="text" value="FRED"/>
DATE	<input type="text" value="06/27/1995"/>

Note that the default value of the field DATE appears formatted according to the user's master record on the selection screen.

Suppressing Display of Parameters

Suppressing Display of Parameters

To suppress the display of a parameter on the selection screen, use the NO-DISPLAY option of the PARAMETERS statement. The syntax is as follows:

Syntax

```
PARAMETERS <p>..... NO-DISPLAY.....
```

The parameter is created and can be given a value for the standard selection screen either internally with the DEFAULT option when you define it or during the INITIALIZATION event (see [INITIALIZATION \[Page 1213\]](#)). If you start an executable program (report) using the SUBMIT statement, the calling program can also pass the value. When you use user-defined selection screens, you can pass a value to the parameter at any time before calling the selection screen.

If you want to display a parameter only under certain conditions, for example, depending on the values that the report user entered in other input fields of the selection screen, do **not** use the NO-DISPLAY option. With this option, the parameter is not an element of the selection screen and you cannot make it visible with the MODIFY SCREEN statement (see [Assigning Parameters to a Modification Group \[Page 814\]](#)).

To make a parameter a hidden element of the selection screen, declare it without the NO-DISPLAY option and suppress its display by using the MODIFY SCREEN statement.

Allowing Parameters to Accept Upper and Lower Case

To allow the user to enter a parameter value in either upper or lowercase, use the LOWER CASE option of the PARAMETERS statement. The syntax is as follows:

Syntax

```
PARAMETERS <p>..... LOWER CASE.....
```

Without the LOWER CASE option, the system changes all input values to uppercase.

If you use the LIKE option to reference a field from the ABAP Dictionary, the parameter receives all attributes of the ABAP Dictionary field. These attributes cannot be changed, and you cannot use the LOWER CASE option. The possibility for entering either upper or lowercase values must be defined in the ABAP Dictionary.

```
PARAMETERS: FIELD1(10),  
             FIELD2(10) LOWER CASE.
```

```
WRITE: FIELD1, FIELD2.
```

Assume the following input values on the selection screen:

FIELD1	upper
FIELD2	lower

The output appears as follows:

UPPER lower

The contents of FIELD1 is changed to uppercase.

Making Parameters Required Input Fields

Making Parameters Required Input Fields

To make a parameter a required input field, use the OBLIGATORY option of the PARAMETERS statement. The syntax is as follows:

Syntax

```
PARAMETERS <p>..... OBLIGATORY.....
```

When you use this option, a question mark appears in the input field for parameter <p>. The user cannot continue with the program without entering a value in this field on the selection screen.

```
PARAMETERS FIELD(10) OBLIGATORY.
```

The resulting selection screen appears as follows:

A screenshot of a selection screen in SAP. It shows a label 'FIELD' on the left and an input field on the right. Inside the input field, there is a question mark '?' indicating that the field is mandatory.

Creating Checkboxes on the Selection Screen

To define a checkbox for parameter input, use the option AS CHECKBOX of the PARAMETERS statement. The syntax is as follows:

Syntax

```
PARAMETERS <p>..... AS CHECKBOX.....
```

The parameter <p> is created with type C of length 1. In this case, the use of the additional options TYPE and LIKE is not allowed.

Valid values for <p> are ' ' and 'X'. These values are assigned to the parameter when the user clicks the checkbox on the selection screen.

```
PARAMETERS: A AS CHECKBOX,  
             B AS CHECKBOX DEFAULT 'X'.
```

In this example, two checkboxes appear on the left side of the selection screen with the field names appearing on their right. Checkbox B has the default value of 'X'.



When the user clicks these boxes, the values 'X' or ' ' are assigned to the respective parameters.

If you use the LIKE option of the PARAMETERS statement to refer to an ABAP Dictionary field of type CHAR of length 1 and valid values 'X' and ' ' (defined in the field's domain, see the documentation [BC - ABAP Dictionary \[Ext.1\]](#)), the parameter will appear automatically as a checkbox on the selection screen.

Creating Radio Button Groups on the Selection Screen

Creating Radio Button Groups on the Selection Screen

To define groups of radio buttons for parameter input, use the RADIOBUTTON GROUP option of the PARAMETERS statement. The syntax is as follows:

Syntax

PARAMETERS <p>..... RADIOBUTTON GROUP <radi>.....

The parameter <p> is created with type C of length 1 and assigned to the group <radi>. The maximum length of the string <radi> is 4. Use of the additional option LIKE is allowed, but you must refer to a field of type C with length 1.

You must assign at least two parameters to each <radi> group. Only one parameter per group can have a default value assigned with the DEFAULT option. This value must be 'X'.

When the user clicks a radio button on the selection screen, the respective parameter is activated (assigned the value 'X'), while all others of the same group are made inactive (assigned the value ' ').

```
PARAMETERS: R1 RADIOBUTTON GROUP RAD1,  
             R2 RADIOBUTTON GROUP RAD1 DEFAULT 'X',  
             R3 RADIOBUTTON GROUP RAD1,  
             S1 RADIOBUTTON GROUP RAD2,  
             S2 RADIOBUTTON GROUP RAD2,  
             S3 RADIOBUTTON GROUP RAD2 DEFAULT 'X'.
```

In this example R1, R2, and R3 form one group of radio buttons and S1, S2, and S3 form another group. On the selection screen, R2 and S3 are activated while all others are inactive.



R1	<input type="radio"/>
R2	<input checked="" type="radio"/>
R3	<input type="radio"/>
S1	<input type="radio"/>
S2	<input type="radio"/>
S3	<input checked="" type="radio"/>

If no DEFAULT option is used, the first parameter of each group (namely R1 and S1) would be activated and assigned the value 'X'.

Using Default Values from SAP Memory

The MEMORY-ID option of the PARAMETERS statement allows you to use a default value from the global SAP memory (SPA/GPA parameters). The syntax is as follows:

Syntax

PARAMETERS <p>..... MEMORY ID <pid>.....

When you use this option, the value stored at that point under the name <pid> in the global user-related SAP memory appears as the default value for <p> on the selection screen.

<pid> can be up to 3 characters long and must not be enclosed in quotation marks.

You use the global SAP memory to pass values retained beyond transaction limits between programs. This memory is available to a user for the entire duration of a terminal session and any parallel sessions use the same memory. The SAP memory is thus more comprehensive than the transaction-bound ABAP memory (see [Data Clusters in ABAP Memory \[Page 363\]](#)).

Parameters can be set user specific in the user's master records with the name <pid>. For further information about the SAP memory, refer to the keyword documentation for SET/GET PARAMETER and to [Passing Data Between Programs \[Page 1338\]](#).

The following program stores a value under the name "HK" in the global SAP memory:

```
REPORT SAPMZTS1.
```

```
SET PARAMETER ID 'HK' FIELD 'Test Parameter'.
```

This value is used in the following executable program (report) as a default value for the parameter TEST:

```
PROGRAM SAPMZTS2.
```

```
PARAMETERS TEST(16) MEMORY ID HK.
```

The selection screen of SAPMZTS2 looks as follows:

TEST	Test Parameter
------	----------------

Assigning Matchcode Objects to Parameters

Assigning Matchcode Objects to Parameters

To assign a matchcode object to a parameter, use the MATCHCODE OBJECT option of the PARAMETERS statement. The syntax is as follows:

Syntax

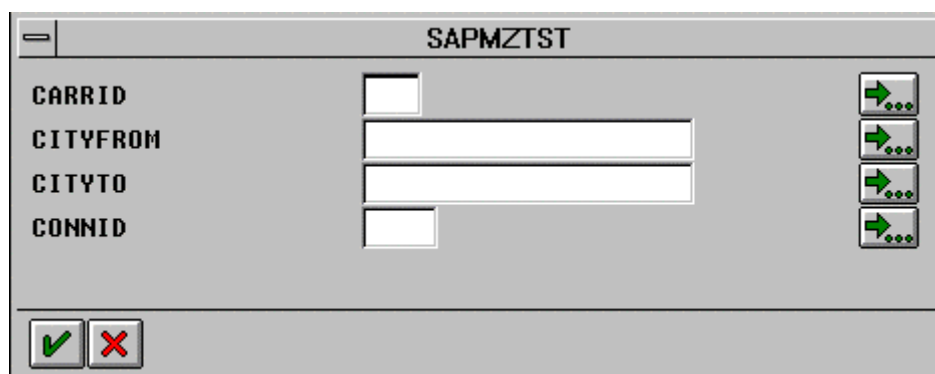
```
PARAMETERS <p>..... MATCHCODE OBJECT <obj>.....
```

The name of the matchcode object <obj> should be a four-character variable name not enclosed in quotation marks.

If you use this option, the possible entries pushbutton appears after the input field of the parameter <p>. When the user presses this button, the effect is to perform a matchcode selection for the input field. For further information about matchcodes, refer to the documentation [BC - ABAP Dictionary \[Ext.\]](#).

```
PARAMETERS CONN(10) MATCHCODE OBJECT SPFL.
```

After clicking the possible entries pushbutton, the following dialog box appears on the selection screen:



The screenshot shows a dialog box titled 'SAPMZTST'. It contains four input fields with corresponding matchcode selection buttons (green arrows with three dots) to their right:

- CARRID: A small input field.
- CITYFROM: A medium-length input field.
- CITYTO: A medium-length input field.
- CONNID: A small input field.

At the bottom left of the dialog box are two buttons: a green checkmark and a red 'X'.

The matchcode object SPFL is used to find the flight number. By positioning the cursor on a field, entering a value, and choosing *Return*, the user gets a list of all flight numbers matching the value entered, e.g. all flights of a particular carrier.

Assigning Parameters to a Modification Group

To assign a parameter to a modification group, which can be used for later screen modifications, use the MODIF ID option of the PARAMETERS statement as follows:

Syntax

```
PARAMETERS <p>..... MODIF ID <key>.....
```

The name of the modification group <key> should be a three-character variable name without quotes.

The MODIF ID option always assigns <key> to the column SCREEN-GROUP1 of the internal table SCREEN.

Parameters assigned to a modification group can be processed as an entire group with the LOOP AT SCREEN/MODIFY SCREEN statements during the AT SELECTION-SCREEN OUTPUT event (see [PBO of the Selection Screen \[Page 1224\]](#)).

For more information about modification groups and the internal table SCREEN, see the keyword documentation for LOOP AT SCREEN or the section on screen modifications in the documentation [BC ABAP Workbench Tools \[Ext.\]](#).

```
PARAMETERS: TEST1(10) MODIF ID SC1,  
            TEST2(10) MODIF ID SC2,  
            TEST3(10) MODIF ID SC1,  
            TEST4(10) MODIF ID SC2.
```

```
AT SELECTION-SCREEN OUTPUT.
```

```
LOOP AT SCREEN.  
  IF SCREEN-GROUP1 = 'SC1'.  
    SCREEN-INTENSIFIED = '1'.  
    MODIFY SCREEN.  
    CONTINUE.  
  ENDIF.  
  IF SCREEN-GROUP1 = 'SC2'.  
    SCREEN-INTENSIFIED = '0'.  
    MODIFY SCREEN.  
  ENDIF.  
ENDLOOP.
```

In the PARAMETERS statement, the parameters TEST1 and TEST3 are assigned to the group SC1, while TEST2 and TEST4 are assigned to the group SC2. During the AT SELECTION-SCREEN OUTPUT event, the INTENSIFIED field of the internal table SCREEN is set to 1 or 0, according to the contents of the GROUP1 field. On the selection screen, the lines for TEST1 and TEST3 are highlighted while those for TEST2 and TEST4 are not, as shown below:



SELECT-OPTIONS - Defining Selection Criteria

SELECT-OPTIONS - Defining Selection Criteria

Selection criteria allow you to handle complex selection restrictions easily in your ABAP programs. Each selection criterion is normally assigned to a particular column in a database table. However, you can also assign selection criteria to internal fields in a program. A selection criterion may only apply to one database table. However, a database table can have more than one selection criterion linked to it (for example, a criterion for each column). You should use selection criteria whenever you need the user to enter complex selections, since they save you from having to write lengthy logical expressions (see [Using Selection Criteria in Programs \[Page 857\]](#)).

Internally, selection criteria are stored in special internal tables called [Selection Tables \[Page 816\]](#). You create selection criteria (and thus selection tables) using the SELECT-OPTIONS statement. Each selection criteria which you define using SELECT-OPTIONS appears as a separate set of fields in which you can enter selection restrictions.

You can use the SELECT-OPTIONS statement for both standard and user-defined selection screens (see [Standard and User-defined Selection Screens \[Page 801\]](#)). If your program has a link to a logical database, you should note the special rules for handling selection criteria (see [Program-specific Selection Criteria and Logical Databases \[Page 820\]](#)).

The following sections explain the variants of the SELECT-OPTIONS statements which you can use for program-specific selections.

[Basic Form of the SELECT-OPTIONS Statement \[Page 821\]](#)

[Assigning Default Values to Selection Criteria \[Page 825\]](#)

[Restricting the Selection Table to One Line \[Page 827\]](#)

[Restricting the Selection Table to Single Value Selection \[Page 828\]](#)

[Preventing the Transfer of Selection Criteria to Logical Databases \[Page 829\]](#)

[Further Options for Selection Criteria \[Page 831\]](#)

There are other variants of the SELECT-OPTIONS statement which you can use to define database-specific selections in logical databases. For further information, see the keyword documentation and [Features and Maintenance of Logical Databases \[Page 1246\]](#).

Selection Tables

For each SELECT-OPTIONS statement, the system creates a selection table. The purpose of selection tables is to store complex selection limits in a standardized manner. They can be used in several ways. Their main purpose is to transfer the selection criteria directly to database tables using the WHERE clause of Open SQL statements (see [Using Selection Criteria in Programs \[Page 857\]](#)).

As well as [Selection Tables \[Page 816\]](#) which you create using SELECT-OPTIONS, you can use the [RANGES \[Page 818\]](#) statement to create internal tables with the structure of [Selection Tables \[Page 816\]](#). You can use these tables with restrictions in the same way as [Selection Tables \[Page 816\]](#) which you create using SELECT-OPTIONS.

A selection table is an internal table with a header line. Its line structure is a field string of four components, namely SIGN, OPTION, LOW, and HIGH. Each line of the selection table represents conditions for data selection:

- SIGN

The data type of SIGN is C with length 1. SIGN is a flag that denotes for each line whether the result of the line condition is to be included or to be excluded from the resulting set of all lines. Possible values are I and E.

- I stands for “inclusive” (inclusion criterion - operators are not inverted)
- E stands for “exclusive” (exclusion criterion - operators are inverted)

- OPTION

The data type of OPTION is C with length 2. OPTION contains the selection operator. The following operators are available:

- If HIGH is empty, you can use EQ, NE, GT, LE, LT, CP, and NP. These operators are described in [Programming Logical Expressions \[Page 235\]](#). The operators CP and NP do not have the scope they have in normal logical expressions. They are only available if wildcards ('*' or '+') are used in the input fields. No escape symbol is defined.
- If HIGH is filled, you can use BT (between) and NB (not between). These operators function as BETWEEN and NOT BETWEEN (see [Checking Whether a Field Belongs to a Range \[Page 243\]](#)).

- LOW

The data type of LOW is the same as the column type of the database table, to which the selection criterion is attached.

- If HIGH is empty, the contents of LOW defines a single value selection. In combination with the operator in OPTION, it specifies a condition for the database selection.
- If HIGH is filled, the contents of LOW and HIGH specify the upper and lower limits for an interval selection. In combination with the operator in OPTION, the interval specifies a condition for the database selection.

- HIGH

Selection Tables

The data type of HIGH is the same as the column type of the database table, to which the selection criterion is attached. The contents of HIGH specifies the upper limit for an interval selection. In combination with the operator in OPTION, the interval specifies a condition for the database selection.

If the selection table contains more than one line, the system performs the data selection according to the following rules:

1. Form the union of sets defined on the lines that have SIGN field equal to I (inclusion).
2. Subtract the union of sets defined on the lines that have SIGN field equal to E (exclusion).
3. Select the resulting set.

If the selection table consists only of lines in which the SIGN field equals E, the system selects all data outside the set specified in the lines.

RANGES

You can use the RANGES statement to create internal tables with the same structure as [Selection Tables \[Page 816\]](#).

Syntax

RANGES <seltab> FOR <f>.

This statement creates a selection table <seltab> which refers to the column <f> of a database table or to an internal field <f>. The selection table <seltab> must be filled in the program. You do not have to declare the database table additionally in the program with the TABLES statement.

The RANGES statement is a short form of the following statements:

```
DATA: BEGIN OF <seltab> OCCURS 10,
      SIGN(1),
      OPTION(2)
      LOW LIKE <f>,
      HIGH LIKE <f>,
      END OF <seltab>.
```

Internal tables created with RANGES have the same structure as selection tables, but they do not have the same functionality.

Selection tables created with the RANGES statement are

- not part of the selection screen: They cannot be used for data transfer in a program <prog> started by the statement
 SUBMIT <prog> WITH <seltab> IN <table>.
 Note that <table> can be created in the calling program with RANGES (see [Calling Executable Programs \(Reports\) \[Page 1113\]](#)).
- not linked to a database table. This means that
 - they are not passed to logical databases (see [Program-specific Selection Criteria and Logical Databases \[Page 820\]](#)).
 - they cannot be used with the short form of logical expressions (see [Using Selection Tables in Logical Expressions \[Page 859\]](#)).
 - they cannot be used with the variant CHECK SELECT-OPTIONS described in [Using Selection Tables with the CHECK Statement in GET Events \[Page 862\]](#).

You can use these internal tables like true selection tables in the WHERE clause of Open SQL statements and in logical expressions with the IN parameter (see [Using Selection Criteria in Programs \[Page 857\]](#)).

Tables created using the RANGES statement play an important role in passing parameters when you call an executable program from another program using the SUBMIT statement. The program calling the executable program can fill the RANGES table with values which are passed as parameters to the actual selection table in the program called without the selection screen having to be displayed (see [Calling Executable Programs \(Reports\) \[Page 1113\]](#)).

```
RANGES S_CARRID FOR SPFLI-CARRID.
```

RANGES

```
S_CARRID-SIGN = 'I'.  
S_CARRID-OPTION = 'EQ'.  
S_CARRID-LOW = 'LH'.
```

```
APPEND S_CARRID.
```

In this example, the internal table S_CARRID is created with the structure of a selection table and by referring to the column CARRID of the database table SPFLI. The fields S_CARRID-LOW and S_CARRID-HIGH have the same type as CARRID. The header line of the internal table S_CARRID is filled and appended to the table. The selection condition, that is defined in the table, functions as the following logical expression:
SPFLI-CARRID EQ 'LH'

Program-specific Selection Criteria and Logical Databases

If a logical database is attached to your executable program, and you define a selection criterion in the program which is attached to a database table of the logical database, you must distinguish between two cases:

- If you define a selection criterion for a column of a database table which does **not** provide dynamic selections (see [Standard Selection Screens and Logical Databases \[Page 797\]](#)), your self-defined selection criteria do not influence the amount of data read by the logical database. Only after reading can you perform checks within your executable program (report) during a GET event (see [Using Selection Criteria in Programs \[Page 857\]](#)).
- If you define a selection criterion for a column of a database table designated for dynamic selections (see [Standard Selection Screens and Logical Databases \[Page 797\]](#)), the system transfers the values the user enters in the input fields on the selection screen to the logical database. There, they are used as dynamic selections. This kind of selection is more efficient than for database tables not designated for dynamic selections.

By defining a selection criterion for a column of a database table, which is foreseen for dynamic selections, with the SELECT-OPTIONS statement, you enable the system to display the dynamic selection directly on the selection screen. The user must not click *User selection* to let it appear. The result is the same as if the user would enter his selection limits on the screen for dynamic selections.

You can also prevent the system from transferring the selection criteria to the logical database (see [Preventing the Transfer of Selection Criteria to Logical Databases \[Page 829\]](#)).

The contents of internal tables created with the [RANGES \[Page 818\]](#) statement are not transferred to logical databases.

Basic Form of the SELECT-OPTIONS Statement

Basic Form of the SELECT-OPTIONS Statement

To create selection criteria that the report user can fill on the selection screen, use the SELECT-OPTIONS statement as follows:.

Syntax

SELECT-OPTIONS <seltab> FOR <f>.

This statement creates a selection table <seltab>, which is attached to the column <f> of a database table or to an internal field <f>. The database table must be declared in the program with the TABLES statement. The name <seltab> can contain up to eight characters.

The field <f> cannot have data type F. The data type F is not supported on the selection screen.

The selection table <seltab> is filled by the report user on the selection screen. In the program, you can modify the lines of the selection table or append more lines to it.

You can change the text fields to the left of the input fields on the selection screen by using text elements as described in [Selection Texts \[Page 154\]](#).

The following example shows how the selection table is filled with the user inputs on the selection screen:

```
REPORT SAPMZTST.
TABLES SPFLI.
SELECT-OPTIONS AIRLINE FOR SPFLI-CARRID.
LOOP AT AIRLINE.
  WRITE: / 'SIGN:', AIRLINE-SIGN,
        'OPTION:', AIRLINE-OPTION,
        'LOW:', AIRLINE-LOW,
        'HIGH:', AIRLINE-HIGH.
ENDLOOP.
```

After starting SAPMZTST, a selection screen appears as follows:

AIRLINE	<input type="text"/>	To	<input type="text"/>	
---------	----------------------	----	----------------------	---

The name of the selection criterion, two input fields, *FROM* and *TO*, and an arrow icon are displayed. The value which the report user enters into the first input field (*FROM* field), is written into the AIRLINE-LOW component of the selection table. The value which the report user enters into the second input field (*TO* field), is written into the AIRLINE-HIGH component of the selection table.

For example, assume that the report user enters values as shown here:

AIRLINE	AA	To	<input type="text"/>	
---------	----	----	----------------------	---

The output of SAPMZTST appears as follows:

SIGN: I OPTION: EQ LOW: AA HIGH:

If the user leaves the *TO* field blank (single value selection), the default settings for SIGN and OPTION are I and EQ.

Basic Form of the SELECT-OPTIONS Statement

For example, assume that the report user enters values as shown here:

AIRLINE	AA	to	LH	
---------	----	----	----	--

The output of SAPMZTST appears as follows:

SIGN: I OPTION: BT LOW: AA HIGH: LH

If the user enters a value in the *TO* field (interval selection), the default settings for SIGN and OPTION are I and BT.

To set up a more complex selection pattern, the user can click the arrow icon on the right side of the selection screen. A new *Multiple Selection* window appears. Assume that the user fills the fields, as shown here:

Multiple Selection for AIRLINE			
Single value selections			
	AA		
+	AF		
+			
+			
+			
From line		1 of 2	
Ranges			
	DL	to	LH
+	SA	to	UA
+		to	
+		to	
+		to	
From line		1 of 2	
Copy Options Multiple selection..			

After the user saves such an extended selection pattern by clicking *Copy*, the arrow icon on the selection screen turns to green to indicate that the selection is more complex than shown in the *To* and *From* fields.

The purpose of the *Multiple Selection* window is filling more than one line of the selection table.

The output of SAPMZTST appears as follows:


SIGN: I OPTION: EQ LOW: AA HIGH:

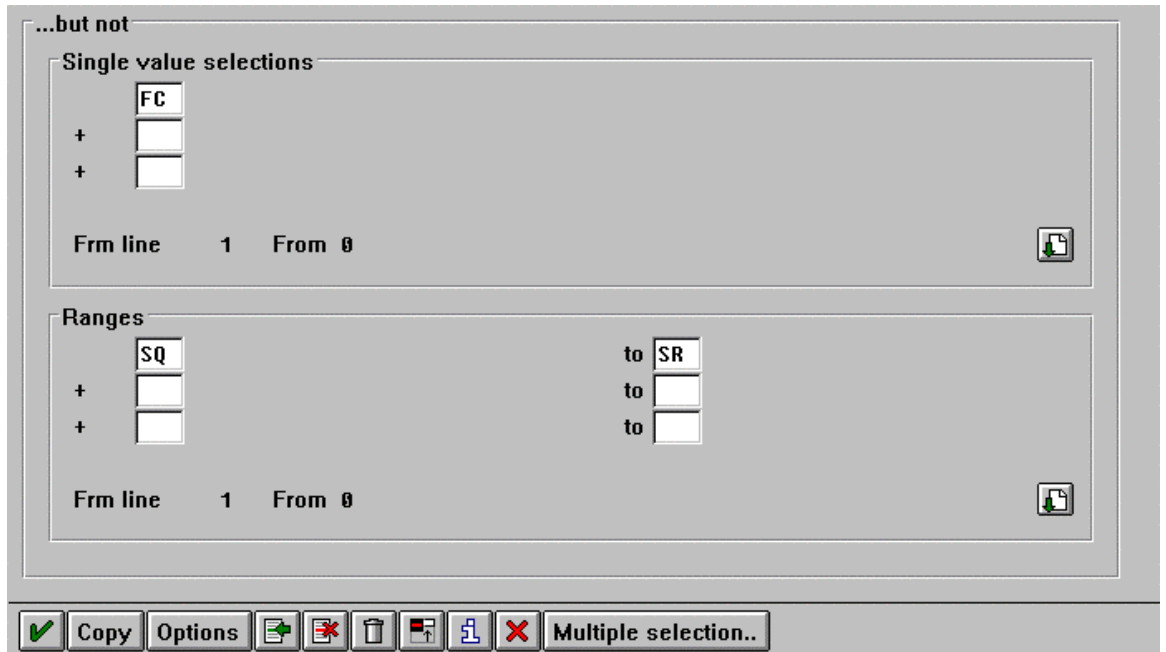
SIGN: I OPTION: EQ LOW: AF HIGH:

SIGN: I OPTION: BT LOW: DL HIGH: LH

SIGN: I OPTION: BT LOW: SA HIGH: UA


Basic Form of the SELECT-OPTIONS Statement

By clicking the *Expand* icon  in the *Multiple Selections* window, the user can enter the selection criteria to be excluded from the resulting set:




The screenshot shows a window titled "...but not" with two main sections: "Single value selections" and "Ranges".







Single value selections:

- Input field: FC
- Buttons: + (two times)
- Footer: Frm line 1 From 0
- Icon: 

Ranges:

- Input field: SQ
- Buttons: + (two times)
- Input field: SR
- Buttons: to (two times)
- Footer: Frm line 1 From 0
- Icon: 

Toolbar:

- Buttons:  Copy Options      Multiple selection..

If the user enters values as above, the output of SAPMZTST appears as follows:

SIGN: E OPTION: EQ LOW: FC HIGH:

SIGN: E OPTION: BT LOW: SQ HIGH: SR

The SIGN fields of the corresponding lines in the selection table contain an E.

To set the SIGN and OPTION values in the lines of the selection table, the user must double-click the input fields of the selection screen or of the *Multiple Selection* window. The *Maintain Selection Options* window appears which, for single value selections, looks as follows:

Basic Form of the SELECT-OPTIONS Statement

The user can choose the selection operator via *Selection Options*. The correspondence between the symbols and the possible operators in the OPTION field is the same as in the table in [Specifying Conditions for Line Selection in the Program \[Page 560\]](#).

The user can choose whether the SIGN field of the selection table should contain I (*Select*) or E (*Exclude from selection*) via *Incl/Excl*.

When the user saves this selection pattern, the corresponding field on the selection screen or in the *Multiple Selection* window appears as follows:



The symbol of the operator in the OPTION field is shown, and the red color indicates that the SIGN field contains E. If the SIGN field contains I, it is green.

With this selection, the output of SAPMZTST appears as follows:

SIGN: E OPTION: GE LOW: AA HIGH:

The *Maintain Selection Options* window for an interval selection criterion functions in a similar way.

Assigning Default Values to Selection Criteria

Assigning Default Values to Selection Criteria

To assign default values to the selection criteria to be displayed on the selection screen, use the DEFAULT option of the SELECT-OPTIONS statement. The syntax is as follows:

Syntax

```
SELECT-OPTIONS <seltab> FOR <f> DEFAULT <g> [TO <h>]....
```

The default values <g> and <h> may be actual values (in single quotes) or the names of fields whose contents should be used as the default values.

The system transports the default values to the selection criteria before the event INITIALIZATION (see [INITIALIZATION \[Page 1213\]](#)). Therefore, the specification of field names instead of literals is only feasible for fields that are already filled when the user starts the program. Such fields are, for example, system fields (see [System-Defined Data Objects \[Page 117\]](#)).

For each SELECT-OPTIONS statement, you can specify one DEFAULT addition. This means that you can fill only the **first line** of the selection table <seltab> with default values. All components of the first line of the selection table can be preset by using the DEFAULT option:

- To set only the LOW field (single value selection), use:
.....DEFAULT <g>.
- To set the LOW and HIGH fields (interval selection), add TO as follows:
.....DEFAULT <g> TO <h>.
- To set the OPTION field (selection operator), add OPTION <op> as follows:
.....DEFAULT <g> [to <h>] OPTION <op>.
 - For single value selection, <op> can be EQ, NE, GE, GT, LE, LT, CP, or NP. The default value is EQ.
 - For interval selection, <op> can be BT or NB. The default value is BT.
- To set the SIGN field (inclusion/exclusion), add SIGN <s>, as follows:
.....DEFAULT <g> [to <h>] [OPTION <op>] SIGN <s>.

The SIGN <s> can be I (inclusion) and E (exclusion). The default value is I.

```
REPORT SAPMZTST.
TABLES SPFLI.
SELECT-OPTIONS AIRLINE FOR SPFLI-CARRID
      DEFAULT 'AA'
      TO 'LH'
      OPTION NB
      SIGN I.
```

After starting SAPMZTST, the selection screen appears as follows:

AIRLINE	 AA	to	LH	
---------	--	----	----	---

Assigning Default Values to Selection Criteria

The symbol before the *FROM* field shows that the AIRLINE-OPTION field contains the operator NB (not between). This symbol is green to show that AIRLINE-SIGN field contains I. The arrow on the right side is not green because only one line of the section table is filled.

Restricting the Selection Table to One Line

Restricting the Selection Table to One Line

To restrict the user's access to the selection table to the first line, use the NO-EXTENSION option of the SELECT-OPTIONS statement. The syntax is as follows:

Syntax

```
SELECT-OPTIONS <seltab> FOR <f>..... NO-EXTENSION.....
```

If you specify this option, the right arrow does not appear on the selection screen and the user cannot access the *Multiple Selection* window.

```
REPORT SAPMZTST.
```

```
TABLES SPFLI.
```

```
SELECT-OPTIONS AIRLINE FOR SPFLI-CARRID NO-EXTENSION.
```

The selection screen appears as follows:

AIRLINE	<input type="text"/>	To	<input type="text"/>
---------	----------------------	----	----------------------

Restricting the Selection Table to Single Value Selection

Restricting the Selection Table to Single Value Selection

To restrict the appearance of a selection criterion on the selection screen to a single value selection, use the NO INTERVALS option of the SELECT-OPTIONS statement. The syntax is as follows:

Syntax

```
SELECT-OPTIONS <seltab> FOR <f>..... NO INTERVALS.....
```

If you specify this option, the *TO* field does not appear on the selection screen. The input on the selection screen is restricted to a single value selection. However, the user can enter interval selections into the *Multiple Selection* window.

```
REPORT SAPMZTST.  
TABLES SPFLI.  
SELECT-OPTIONS AIRLINE FOR SPFLI-CARRID NO INTERVALS.
```

The selection screen appears as follows:



The user can enter only a single value selection directly. However, the user can enter further selections by clicking the arrow on the right side of the screen.

If you add the NO-EXTENSION option as follows:

```
REPORT SAPMZTST.  
TABLES SPFLI.  
SELECT-OPTIONS AIRLINE FOR SPFLI-CARRID NO INTERVALS  
NO-EXTENSION.
```

Then the selection screen appears as follows:



Now, the user can enter only a single value selection.

Preventing the Transfer of Selection Criteria to Logical Databases

Preventing the Transfer of Selection Criteria to Logical Databases

If a logical database is attached to your report program, you can connect a selection criterion to a database table, which is also part of the logical database. If that database table is designated for a dynamic selection (see [Standard Selection Screens and Logical Databases \[Page 797\]](#)), the system transfers the corresponding selection criteria to the logical database (see [Program-specific Selection Criteria and Logical Databases \[Page 820\]](#)). The logical database does not read lines from the database table that do not meet these selection criteria.

If you want the logical database to read these lines anyway, for example, when you use the selection criteria for other purposes than limiting database accesses (see [Using Selection Criteria in Programs \[Page 857\]](#)), use the NO DATABASE SELECTION option of the SELECT-OPTIONS statement as follows:


Syntax

```
SELECT-OPTIONS <seltab> FOR <f>..... NO DATABASE SELECTION.....
```

The logical database F1S is attached to the following report program. The column CONNID of database table SPFLI is designated for dynamic selection (see [Standard Selection Screens and Logical Databases \[Page 797\]](#)).

```
REPORT SAPMZTST.
TABLES SPFLI.
SELECT-OPTIONS CONN FOR SPFLI-CONNID NO DATABASE SELECTION.
GET SPFLI.
  IF SPFLI-CONNID IN CONN.
    WRITE: SPFLI-CARRID, SPFLI-CONNID, 'meets criterion'.
  ELSE.
    WRITE: SPFLI-CARRID, SPFLI-CONNID,
          'does not meet criterion'.
  ENDIF.
```

The first part of the selection screen is defined in the logical database. The last line (CONN) is defined in the report program.

Airline carrier	AA	to	UA	
From	Frankfurt			
To	Berlin			
CONN	2436	to		

If the user enters the above selection criteria, the output appears as follows:

```
LH 2402 does not meet criterion
LH 2436 meets criterion
```

Preventing the Transfer of Selection Criteria to Logical Databases**LH 2462 does not meet criterion**

The following report does not use the NO DATABASE SELECTION OPTION:

```
REPORT SAPMZTST.  
TABLES SPFLI.  
SELECT-OPTIONS CONN FOR SPFLI-CONNID.  
GET SPFLI.  
  IF SPFLI-CONNID IN CONN.  
    WRITE: SPFLI-CARRID, SPFLI-CONNID, 'meets criterion'.  
  ELSE.  
    WRITE: SPFLI-CARRID, SPFLI-CONNID,  
          'does not meet criterion'.  
  ENDIF.
```

With the same selections as above, the output appears as follows:

LH 2436 meets criterion

The result is the same as if you did not use the SELECT-OPTIONS statement, but the user has selected *User selections* in the application toolbar of the selection screen and then entered 2436 there.

Further Options for Selection Criteria

Further Options for Selection Criteria

There are a number of other options you can use with the SELECT-OPTIONS statement. The tasks you can perform with each option and the syntax for each are listed below:

- To suppress display of a selection criterion on the selection screen, use
SELECT-OPTIONS <seltab> FOR <f>... NO-DISPLAY.....
- To enable the selection criterion to accept upper and lower case letters, use
SELECT-OPTIONS <seltab> FOR <f>... LOWER CASE.....
- To make a selection mandatory for the *From* field on the selection screen, use
SELECT-OPTIONS <seltab> FOR <f>... OBLIGATORY.....
- To use default values from the SAP memory for the *From* field, use
SELECT-OPTIONS <seltab> FOR <f>... MEMORY ID <pid>.....
- To assign the fields of a selection criterion to a modification group, use
SELECT-OPTIONS <seltab> FOR <f>... MODIF ID <key>.....
- To assign a matchcode object to the *From* and *To* fields of a selection criterion, use
SELECT-OPTIONS <seltab> FOR <f>... MATCHCODE OBJECT <obj>...

These options have the same syntax and function as the options for the PARAMETERS statement. For an explanation of the options, see [PARAMETERS - Defining Input Fields for Variables \[Page 803\]](#).

SELECTION-SCREEN - Formatting

The selection screen that you define when you use the PARAMETERS or SELECT-OPTIONS statements on their own, has a standard layout in which all parameters appear line by line.

This standard output is not always sufficient. For example, when you define a group of radio buttons, it is clear that this group must be set off against other input fields.

You can use the formatting options of the SELECTION-SCREEN statement to define your own layout for parameters and selection criteria and to place pushbuttons in the application toolbar or on the screen.

You can only format selections which you call yourself from the program. Standard selection screens of executable programs are only displayed if they contain at least one input field defined using the PARAMETERS or SELECT-OPTIONS statements. You cannot therefore place comments, underlining or pushbuttons on a standard selection screen without including at least one input field. User-defined selection screens, on the other hand, can be called even if they do not contain an input field.

The SELECTION-SCREEN statement has the following additions, enabling you to do the following:

[Specifying Blank Lines, Underlines, and Comments \[Page 833\]](#)

[Placing Several Elements On a Single Line \[Page 839\]](#)

[Positioning an Element \[Page 840\]](#)

[Creating Blocks of Elements \[Page 841\]](#)

[Creating Pushbuttons in the Application Toolbar \[Page 842\]](#)

[Creating Pushbuttons on the Selection Screen \[Page 844\]](#)

Specifying Blank Lines, Underlines, and Comments

Specifying Blank Lines, Underlines, and Comments

The options of the SELECTION-SCREEN statement described in the following topics create

[Blank Lines \[Page 834\]](#)

[Underlines \[Page 835\]](#)

[Comments \[Page 836\]](#)

An example is shown in

[Example of Blank Lines, Underlines, and Comments \[Page 837\]](#)

Blank Lines

To produce blank lines on the selection screen, use the SKIP option with the SELECTION-SCREEN statement. The syntax is as follows:

Syntax

SELECTION-SCREEN SKIP [<n>].

This statement generates <n> blank lines, where <n> can have a value between 1 and 9. To produce a single blank line, you can omit <n>.

Underlines

Underlines

To underline a line or part of a line on the selection screen, use the ULINE option with the SELECTION-SCREEN statement. The syntax is as follows:

Syntax

SELECTION-SCREEN ULINE `[[/]<pos(len)>]` `[MODIF ID <key>]`.

This statement creates an underline.

If you do not use the format option `<pos(len)>`, a new line is created under the current line. If you use the format option `<pos(len)>`, the underline begins on position `<pos>` and continues for a length of `<len>` characters on the current line. With several elements on one line, you can also specify `(<len>)` without `<pos>`.

You can request a line feed by using the optional slash (/) (compare the ULINE and WRITE statements in [Creating Simple Lists with the WRITE Statement \[Page 890\]](#)).

For `<pos>`, you can specify a number, POS_LOW, or POS_HIGH. POS_LOW and POS_HIGH are the positions of the *From* and *To* fields as they appear on the selection screen when the SELECT-OPTIONS statement is used.

As with the PARAMETERS statement, you can use the MODIF ID `<key>` option to assign the underline to a modification group `<key>`, which can be used during the AT SELECTION-SCREEN OUTPUT event to modify the screen (see [Assigning Parameters to a Modification Group \[Page 814\]](#)).

Comments

To write text on the selection screen, use the COMMENT option with the SELECTION-SCREEN statement. The syntax is as follows:

Syntax

```
SELECTION-SCREEN COMMENT [/]<pos(len)> <comm> [FOR FIELD <f>]  
[MODIF ID <key>].
```

You must always define the format (starting position and length - only with several elements on one line can <pos> be omitted) when you use this option. For <comm>, you can specify a text symbol (see [Text Symbols \[Page 157\]](#)) or you can specify a field name with a maximum length of eight characters. This character field must not be declared with (for example) the DATA statement, but is generated with length <len> automatically. You must fill this character field dynamically during the INITIALIZATION event (see [INITIALIZATION \[Page 1213\]](#)).

The text <comm> will be displayed, starting in column <pos>, for a length of <len>. If you do not use the slash (/), the comment is written onto the current line; otherwise a new line is created.

To assign a text label to a parameter or a select option, use the FOR FIELD <f> option. <f> can be the name of a parameter or a selection criterion. As a result, if the user requests help on the comment on the selection screen, the help text for the assigned field <f> is displayed.

The MODIF ID <key> option for the selection screen comment is the same as described for the PARAMETERS statement (see [Assigning Parameters to a Modification Group \[Page 814\]](#)).

Example of Blank Lines, Underlines, and Comments

Example of Blank Lines, Underlines, and Comments

This is an example of how to use blank lines, underlines, and comments on the selection screen:

```

SELECTION-SCREEN COMMENT /2(50) TEXT-001 MODIF ID SC1.

SELECTION-SCREEN SKIP 2.
SELECTION-SCREEN COMMENT /10(30) COMM1.
SELECTION-SCREEN ULINE.

PARAMETERS: R1 RADIOBUTTON GROUP RAD1,
             R2 RADIOBUTTON GROUP RAD1,
             R3 RADIOBUTTON GROUP RAD1.

SELECTION-SCREEN ULINE /1(50).
SELECTION-SCREEN COMMENT /10(30) COMM2.
SELECTION-SCREEN ULINE.

PARAMETERS: S1 RADIOBUTTON GROUP RAD2,
             S2 RADIOBUTTON GROUP RAD2,
             S3 RADIOBUTTON GROUP RAD2.

SELECTION-SCREEN ULINE /1(50).

INITIALIZATION.

COMM1 ='Radio Button Group 1'.
COMM2 ='Radio Button Group 2'.

LOOP AT SCREEN.
  IF SCREEN-GROUP1 = 'SC1'.
    SCREEN-INTENSIFIED = '1'.
    MODIFY SCREEN.
  ENDIF.
ENDLOOP.

```

The selection screen appears as follows:

Example for Blank Lines, Underlines, and Comments	
Radio Button Group 1	
R1	<input checked="" type="radio"/>
R2	<input type="radio"/>
R3	<input type="radio"/>
Radio Button Group 2	
S1	<input checked="" type="radio"/>
S2	<input type="radio"/>
S3	<input type="radio"/>

Example of Blank Lines, Underlines, and Comments

The text specified in the text symbol 001 appears highlighted at the top of the selection screen. Two groups of radio buttons are displayed below the text, separated by underlines and described by comments. If there were no slashes (/) in the statements that use the formatted ULINE option, the underlines would overwrite the last line of each radio button group.

Placing Several Elements On a Single Line

Placing Several Elements On a Single Line

To position a set of parameters or comments on a single line on the selection screen, you must declare the elements in a block enclosed by the following two statements:

Syntax

```
SELECTION-SCREEN BEGIN OF LINE.
```

```
...
```

```
SELECTION-SCREEN END OF LINE.
```

Note that the selection text (name of the parameter or text element) is not displayed when you use this option. To display a selection text, you must provide the description by using the SELECTION-SCREEN statement with the COMMENT option.

Also, do not use a slash with the format option <pos(len)>. In this formatting option, you can omit the position specification <pos> within the above statements. The object is then placed in the current position on the line.

```
SELECTION-SCREEN BEGIN OF LINE.  
  SELECTION-SCREEN COMMENT 1(10) TEXT-001.  
  PARAMETERS: P1(3), P2(5), P3(1).  
SELECTION-SCREEN END OF LINE.
```

This example causes the following line to appear on the selection screen:



The line starts with the text symbol 001 and is followed by the input fields for the parameters P1, P2, and P3.

Positioning an Element

To position the next parameter or comment on the selection screen, use the POSITION option with the SELECTION-SCREEN statement. The syntax is as follows:

Syntax

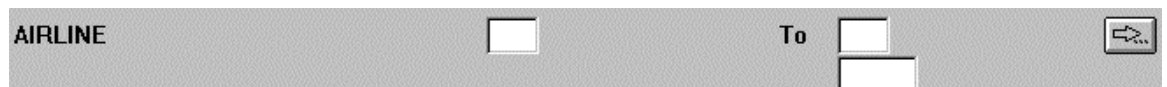
SELECTION-SCREEN POSITION <pos>.

For <pos>, you can specify a number, POS_LOW, or POS_HIGH. POS_LOW and POS_HIGH are the positions of the *From* and *To* fields as they appear on the selection screen when the SELECT-OPTIONS statement is used.

Use the POSITION option only between the BEGIN OF LINE and END OF LINE options.

```
REPORT SAPMZTST.  
TABLES SPFLI.  
SELECT-OPTIONS AIRLINE FOR SPFLI-CARRID.  
SELECTION-SCREEN BEGIN OF LINE.  
  SELECTION-SCREEN POSITION POS_HIGH.  
  PARAMETERS FIELD(5).  
SELECTION-SCREEN END OF LINE.
```

The selection screen appears as follows:



The input field for the parameter FIELD appears under the *TO* field of the selection criterion AIRLINE. For FIELD, no selection text is displayed.

Creating Blocks of Elements

To create a logical block of elements on the selection screen, mark the beginning of the block with the BEGIN OF BLOCK option of the SELECTION-SCREEN statement, then define the individual elements and mark the end of the block with the END OF BLOCK option as shown below:

Syntax

```
SELECTION-SCREEN BEGIN OF BLOCK <block>
                        [WITH FRAME [TITLE <title>]]
                        [NO INTERVALS].
```

```
...
SELECTION-SCREEN END OF BLOCK <block>.
```

You must define a name <block> for each block. You can nest blocks.

If you add the WITH FRAME option, a frame will be drawn around the block. You can nest up to five different blocks with frames.

You can add a title to each frame by using the TITLE option. Like <name> in the COMMENT option, <title> can be a text symbol or a field name with a maximum length of eight characters. This character field must not be declared with (for example) the DATA statement, but is generated with a length that corresponds to the frame width, which is set automatically according to the nesting depth of the frame. You must fill this character field dynamically during the INITIALIZATION event (see [INITIALIZATION \[Page 1213\]](#)).

If you use the NO INTERVALS option, the system processes **all** SELECT-OPTIONS statements in this block as if they had this option (see [Restricting the Selection Table to Single Value Selection \[Page 828\]](#)). If the block has a frame, the width of the frame is drawn smaller and nested blocks automatically inherit the NO INTERVALS option.

```
SELECTION-SCREEN BEGIN OF BLOCK RAD1
                        WITH FRAME TITLE TEXT-002.
PARAMETERS R1 RADIOBUTTON GROUP GR1.
PARAMETERS R2 RADIOBUTTON GROUP GR1.
PARAMETERS R3 RADIOBUTTON GROUP GR1.
SELECTION-SCREEN END OF BLOCK RAD1.
```

On the selection screen, the three radio buttons R1, R2, R3 form a block, which is surrounded by a frame and has the title specified in the text symbol 002, as shown below:



Creating Pushbuttons in the Application Toolbar

You can create up to five pushbuttons in the application toolbar on the selection screen. These buttons are also automatically connected to function keys. The syntax is as follows:

Syntax

SELECTION-SCREEN FUNCTION KEY <i>.

<i> must be between 1 and 5. You must specify the text to appear on the buttons during runtime in the ABAP Dictionary fields SSCRFIELDS-FUNCTXT_0<i>.

You must declare SSCRFIELDS with a TABLES statement.

When the user clicks this button, FC0<i> is entered in the field SSCRFIELDS-UCOMM, which can be checked during the event AT SELECTION-SCREEN (see [AT SELECTION-SCREEN \[Page 1216\]](#)).

```
TABLES SSCRFIELDS.

DATA FLAG.

SELECTION-SCREEN BEGIN OF SCREEN 500 AS WINDOW TITLE TIT.
  SELECTION-SCREEN FUNCTION KEY 1.
  SELECTION-SCREEN FUNCTION KEY 2.
SELECTION-SCREEN END OF SCREEN 500.

AT SELECTION-SCREEN.
  CASE SY-DYNNR.
    WHEN '0500'.
      IF SSCRFIELDS-UCOMM = 'FC01'.
        FLAG = '1'.
      ELSEIF SSCRFIELDS-UCOMM = 'FC02'.
        FLAG = '2'.
      ENDIF.
    ENDCASE.

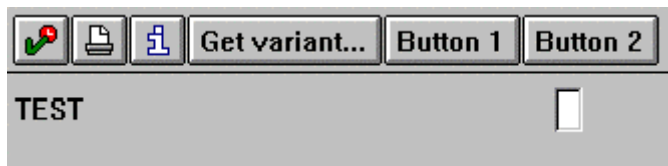
START-OF-SELECTION.

  SSCRFIELDS-FUNCTXT_01 = 'Button 1'.
  SSCRFIELDS-FUNCTXT_02 = 'Button 2'.
  TIT = 'Buttons'.

  CALL SELECTION-SCREEN 500 STARTING AT 10 10
                                ENDING   AT 10 11.

  IF FLAG = '1'.
    WRITE / 'Button 1 was clicked'.
  ELSEIF FLAG = '2'.
    WRITE / 'Button 2 was clicked'.
  ENDIF.
```

This example program defines a user-defined selection screen as a modal dialog box, containing two pushbuttons in the application toolbar with the texts 'Button 1' and 'Button 2'.

Creating Pushbuttons in the Application Toolbar

When the user clicks one of the buttons, the internal field FLAG is set during the event AT SELECTION-SCREEN. The FLAG field can be processed further during the continued flow of the program when the user has chosen *Execute* (see [Controlling the Flow of ABAP Programs by Events \[Page 1208\]](#)).

Creating Pushbuttons on the Selection Screen

To create a pushbutton on the selection screen, you use the PUSHBUTTON option with the SELECTION-SCREEN statement. The syntax is as follows:

Syntax

```
SELECTION SCREEN PUSHBUTTON [/]<pos(len)> <name>
                        USER-COMMAND <ucom> [MODIF ID <key>].
```

The parameters [/]<pos(len)>, <name>, and the MODIF ID option are the same as described for the COMMENT option in [Comments \[Page 836\]](#).

The text specified in <name> is the pushbutton text.

For <ucom>, you must specify a code of up to four characters. When the user clicks the pushbutton on the selection screen, <ucom> is entered in the Dictionary field SSCRFIELDS-UCOMM.

You must declare SSCRFIELDS by using a TABLES statement.

The contents of the SSCRFIELDS-UCOMM field can be checked during the event AT SELECTION-SCREEN (see [AT SELECTION-SCREEN \[Page 1216\]](#)).

The following example has the same effect as the example shown for pushbuttons in the application tool bar (see [Creating Pushbuttons in the Application Toolbar \[Page 842\]](#)). However, according to the *SAP Style Guide*, you are recommended to place pushbuttons in the application toolbar, if possible.

```
TABLES SSCRFIELDS.
DATA FLAG.
PARAMETERS TEST.
SELECTION-SCREEN PUSHBUTTON /20(10) BUT1
                        USER-COMMAND CLI1.
SELECTION-SCREEN PUSHBUTTON /20(10) TEXT-020
                        USER-COMMAND CLI2.

INITIALIZATION.
    BUT1 = 'Button 1'.

AT SELECTION-SCREEN.
    IF SSCRFIELDS-UCOMM = 'CLI1'.
        FLAG = '1'.
    ELSEIF SSCRFIELDS-UCOMM = 'CLI2'.
        FLAG = '2'.
    ENDIF.

START-OF-SELECTION.
    IF FLAG = '1'.
        WRITE / 'Button 1 was clicked'.
    ELSEIF FLAG = '2'.
        WRITE / 'Button 2 was clicked'.
    ENDIF.
```

Creating Pushbuttons on the Selection Screen

If the text symbol TEXT-020 is defined as 'Button 2' (see [Text Symbols \[Page 157\]](#)), this example causes two pushbuttons with the texts 'Button 1' and 'Button 2' to appear on the selection screen, as shown below:



CLI1 and CLI2 are used for <ucom>. When the user clicks one of the buttons, the internal field FLAG is set as defined by the event AT SELECTION-SCREEN. The FLAG field can be processed further during the continued flow of the program (see [Controlling the Flow of ABAP Programs by Events \[Page 1208\]](#)).

Calling Selection Screens

How you call a selection screen from an ABAP program depends on whether you are calling a standard selection screen for an executable program (report) or a user-defined selection screen in a program.

[Calling Standard Selection Screens \[Page 847\]](#)

[Calling User-defined Selection Screens \[Page 848\]](#)

Calling Standard Selection Screens

Calling Standard Selection Screens

The standard selection screen for an executable program consists of

- The selections of any linked logical database
- and all selection screen elements from the declaration part of a program that are not assigned to a user-defined selection screen.

The standard selection screen is called **automatically** as long as you have defined at least one input field for it using the PARAMETERS or SELECT-OPTIONS statement. It is called between the INITIALIZATION and START-OF-SELECTION events (see [Event Blocks for Executable Programs \(Reports\) \[Ext.\]](#)). The PBO and PAI phases of the selection screen generate events, which you can process using the AT SELECTION-SCREEN keyword (see [AT SELECTION-SCREEN \[Page 1216\]](#))

```
REPORT SELSCREENDEF.

TABLES SPFLI.

SELECTION-SCREEN BEGIN OF BLOCK MYSEL WITH FRAME TITLE TIT.
  PARAMETERS: DEPTIME LIKE SPFLI-DEPTIME,
              ARRTIME LIKE SPFLI-ARRTIME.
SELECTION-SCREEN END OF BLOCK MYSEL.

INITIALIZATION.
  TIT = 'Times'.

...
```

If this executable program is linked to logical database F1S, the following selection screen is displayed automatically when you start the program:

The top three blocks are defined in the logical database. The bottom block is defined in the program itself.

Calling User-defined Selection Screens

You can create user-defined selection screens in executable programs (reports), function modules and module pools. You can call them as follows:

[Call From a Program \[Page 849\]](#)

[Call as a Report Transaction \[Page 853\]](#)

[Call as a Dialog Transaction \[Page 855\]](#)

Call From a Program

Call From a Program

Any program in which selection screens may be defined can also call a selection screen using the CALL SELECTION-SCREEN statement.

Syntax

```
CALL SELECTION-SCREEN <numb> [STARTING AT <x1> <y1>]
                                [ENDING AT <x2> <y2>].
```

This statement calls selection screen number <numb>. The selection screen must be defined in the same program, either as the standard selection screen (1000) or as a user-defined selection screen (see [Standard and User-defined Selection Screens \[Page 801\]](#)).

You can call a selection screen as a modal dialog box using the STARTING AT and ENDING AT additions. This is possible even if you have not used the AS WINDOW addition in the definition of the selection screen. However, you are recommended to do so, since warnings and error messages will then also be displayed as modal dialog boxes (see example below).

When it returns from the selection screen to the program, the CALL SELECTION-SCREEN statement sets the return value SY-SUBRC as follows:

- SY-SUBRC = 0 if the user chose *Execute*
- SY-SUBRC = 4 if the user chose *Cancel*

You should always use CALL SELECTION-SCREEN to call selection screens, and not CALL SCREEN. If you use CALL SCREEN, the system will not be able to process the selection screen.

Every selection screen called leads to AT SELECTION-SCREEN events (see [AT SELECTION-SCREEN \[Page 1216\]](#)). The system field SY-DYNNR contains the number of the selection screen which is currently active.

REPORT SELSCREENDEF.

```
SELECTION-SCREEN BEGIN OF BLOCK SEL1 WITH FRAME TITLE TIT1.
  PARAMETERS: CITYFR LIKE SPFLI-CITYFROM,
              CITYTO LIKE SPFLI-CITYTO.
SELECTION-SCREEN END OF BLOCK SEL1.
```

```
SELECTION-SCREEN BEGIN OF SCREEN 500 AS WINDOW.
  SELECTION-SCREEN INCLUDE BLOCKS SEL1.
  SELECTION-SCREEN BEGIN OF BLOCK SEL2
    WITH FRAME TITLE TIT2.
    PARAMETERS: AIRPFR LIKE SPFLI-AIRPFROM,
              AIRPTO LIKE SPFLI-AIRPTO.
  SELECTION-SCREEN END OF BLOCK SEL2.
SELECTION-SCREEN END OF SCREEN 500.
```

```
INITIALIZATION.
  TIT1 = 'Cities'.
```

```
AT SELECTION-SCREEN.
  CASE SY-DYNNR.
    WHEN '0500'.
```

Call From a Program

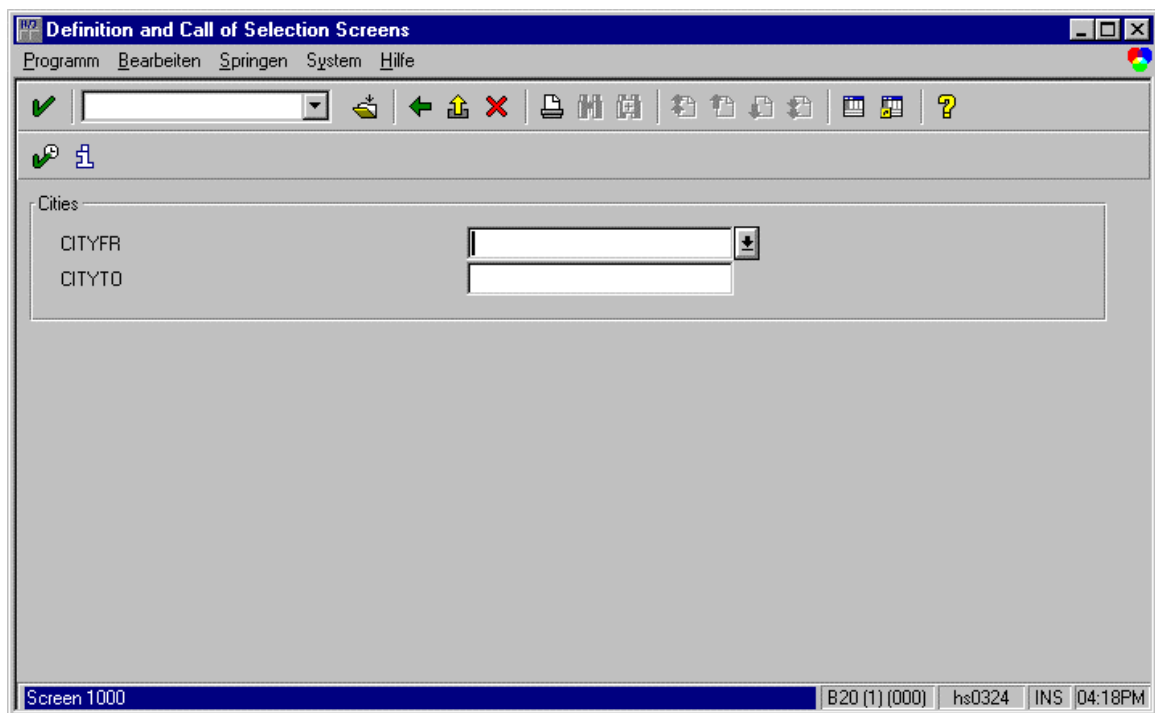
```
MESSAGE W159(AT) WITH 'Screen 500'.  
WHEN '1000'.  
MESSAGE W159(AT) WITH 'Screen 1000'.  
ENDCASE.
```

```
START-OF-SELECTION.  
TIT1 = 'Cities for Airports'.  
TIT2 = 'Airports'.  
CALL SELECTION-SCREEN 500 STARTING AT 10 10.  
TIT1 = 'Cities again'.  
CALL SELECTION-SCREEN 1000 STARTING AT 10 10.
```

The above executable program contains definitions for the standard selection screen and the user-defined screen number 500. Selection screen 500 is defined to be displayed as a modal dialog box. It contains block SEL1 from the standard selection screen. Note the phase in which the titles of the screens are defined. For the purpose of demonstration, the program calls warning messages with message class AT at the AT SELECTION-SCREEN event.

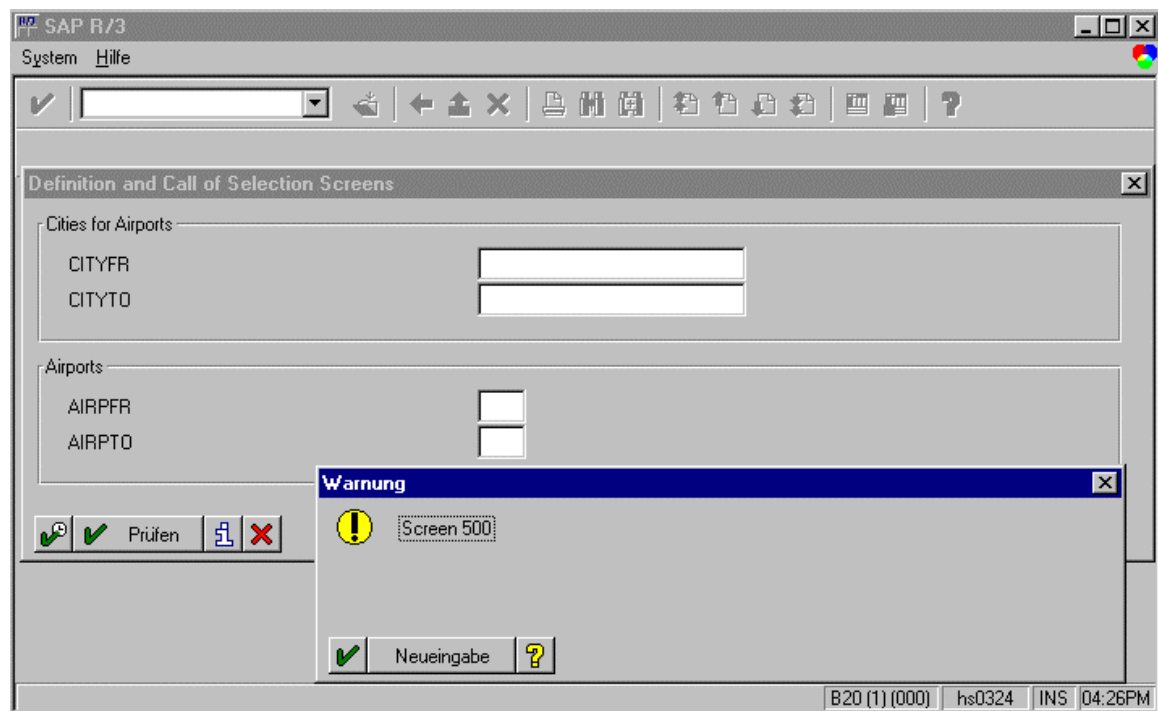
When you start the program, the following sequence of screens is displayed:

1. The standard selection screen. If the user chooses *Execute*, the system displays the warning SCREEN 1000 in the status line.

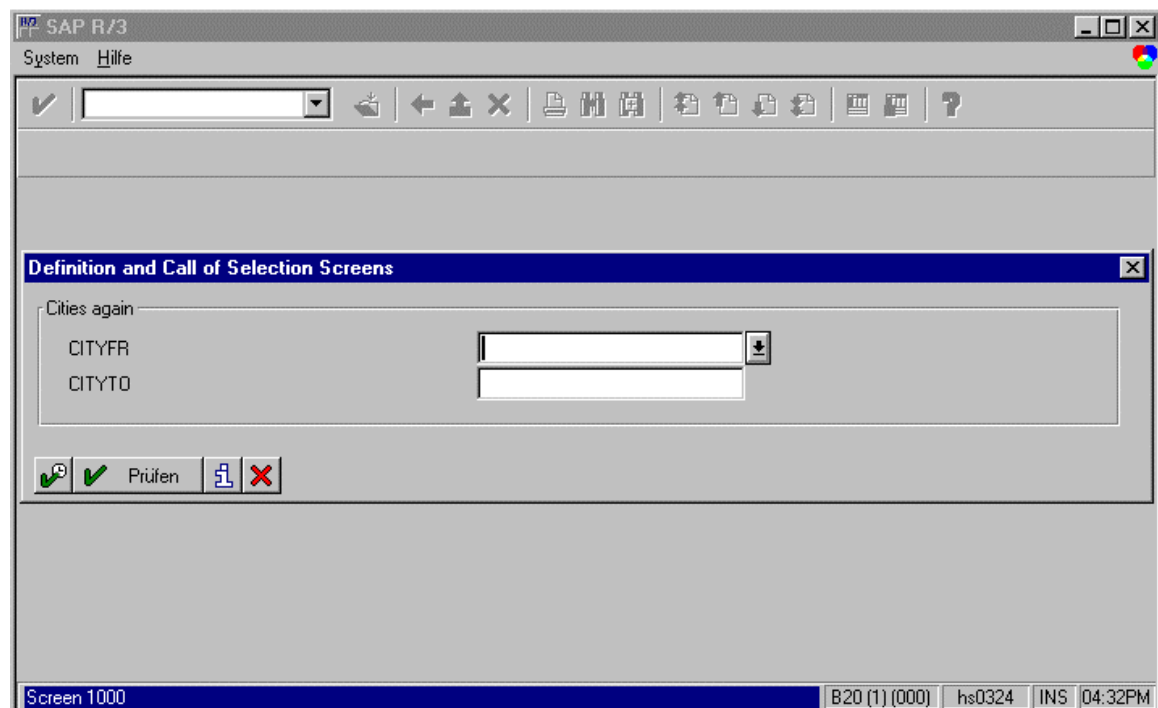


2. Once the user has confirmed the warning by choosing *Enter*, selection screen 500 is called as a modal dialog box. When the user chooses *Execute*, the system displays the warning SCREEN 500, also in a dialog box.

Call From a Program



- When the user has confirmed this warning, the standard selection screen is called again, but this time as a modal dialog box. Since you cannot define the standard selection screen as a modal dialog box, the warning message displayed when the user chooses *Execute* is displayed in the status line and not as a modal dialog box.



The consequence of this is that you can only exit this selection screen using Cancel, since there is no *Enter* function in the dialog box with which to confirm the warning message.

Call as a Report Transaction


Call as a Report Transaction

You can define a transaction code to any executable program (see [Assigning Transaction Codes to Executable Programs \(Reports\) \[Page 81\]](#)). You should choose the type report transaction. If you choose the type dialog transaction, the system will treat the program like a module pool. (For more information about calling selection screens in dialog transactions, see the next section).

When you define a transaction code, you can make one of the selection screens defined in the program into the initial screen. The standard selection screen is proposed as a default, but you can overwrite it.

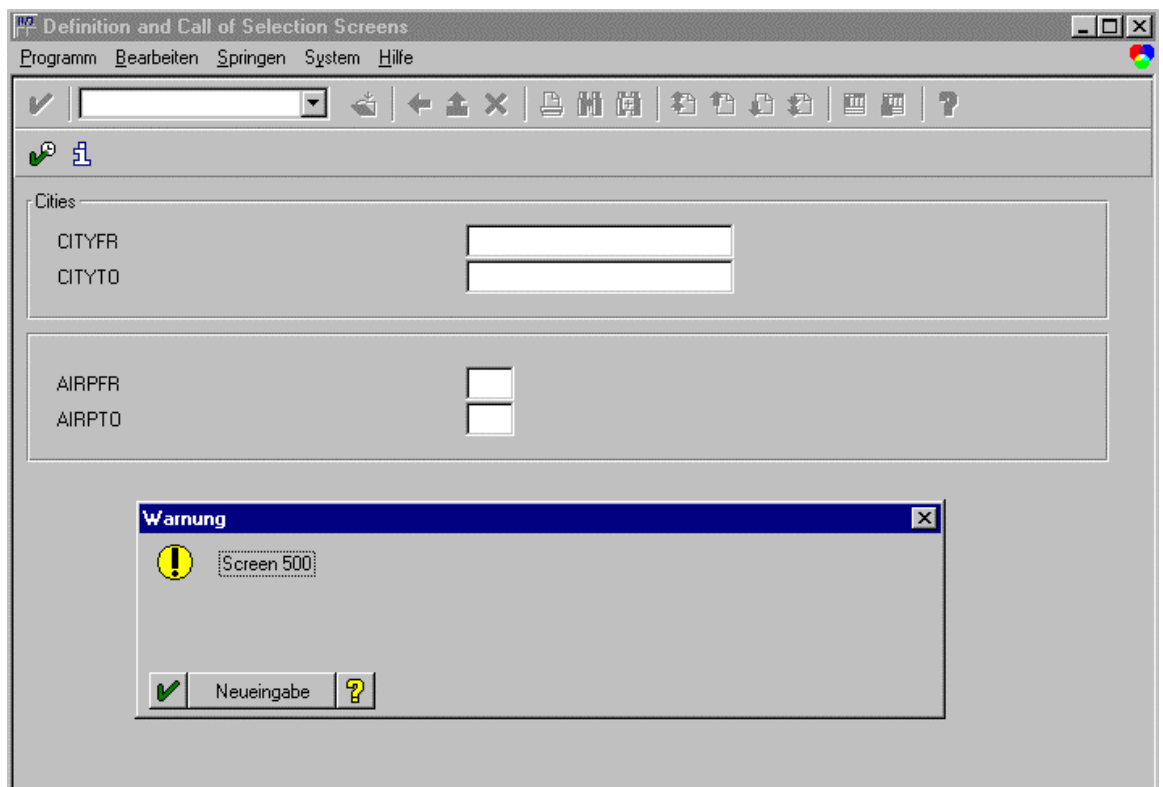
Let us take the sample program SELSCREENDEF from the [Call From a Program \[Page 849\]](#) section and create a report transaction for it with transaction code SELSCREEN500:

Transaktionscode	SELSCREEN500
Entwicklungs-klasse	

Transaktionstext	Calling Selection Screen 500
Programm	SELSCREENDEF
Selektionsbild	500
Start mit Variante	
Berechtigungsobjekt	
 Werte	

When you call the transaction, the executable program is started directly with selection screen 500 as a full screen. However, the warning is still displayed as a dialog box when the user chooses *Execute*, because selection screen 500 was originally defined as a modal dialog box.

Call as a Report Transaction



The sequence of screens then continues in the same way as described from point 2 onwards in the above example [Call From a Program \[Page 849\]](#).

Call as a Dialog Transaction

Call as a Dialog Transaction

You can set selection screens of executable programs and module pools that are started as dialog transactions as the initial screen of the transaction.

When you process the selection screen (AT SELECTION-SCREEN event), you must ensure that the program flow moves onto the correct subsequent screen.

Let us consider the following module pool:

```
*&-----*
*& Modulpool      SAPMSSLS      *
*&-----*
```

INCLUDE MSSLSTOP.

INCLUDE MSSLSEVT.

Include MSSLSTOP contains the following definition of selection screen 500:

```
*&-----*
*& Include MSSLSTOP      *
*&-----*
```

PROGRAM SAPMSSLS.

SELECTION-SCREEN BEGIN OF SCREEN 500 AS WINDOW.

SELECTION-SCREEN BEGIN OF BLOCK SEL1 WITH FRAME.

PARAMETERS: CITYFR LIKE SPFLI-CITYFROM,
CITYTO LIKE SPFLI-CITYTO.

SELECTION-SCREEN END OF BLOCK SEL1.

SELECTION-SCREEN BEGIN OF BLOCK SEL2 WITH FRAME.

PARAMETERS: AIRPFR LIKE SPFLI-AIRPFROM,
AIRPTO LIKE SPFLI-AIRPTO.

SELECTION-SCREEN END OF BLOCK SEL2.

SELECTION-SCREEN END OF SCREEN 500.

Include MSSLSEVT processes the AT SELECTION-SCREEN event:

```
*-----*
* INCLUDE MSSLSEVT      *
*-----*
```

AT SELECTION-SCREEN.

...

LEAVE TO SCREEN 100.

When we create a transaction code for program SAPMSSLS, we enter screen 500 as the initial screen.

Call as a Dialog Transaction

The screenshot shows the configuration for the transaction **SESCREEN_DIALOG**. The fields are as follows:

Transaktionscode	SESCREEN_DIALOG
Entwickungsklasse	\$TMP
<hr/>	
Transaktionstext	Selection Screen as first Dynpro
Programm	SAPMSSL
Dynpronummer	500
Berechtigungsobjekt	<input type="text"/> Werte
<input checked="" type="checkbox"/> Pflege der Standardtransaktionsvariante erlaubt	

Calling transaction `SESCREEN_DIALOG` starts the program by displaying the selection screen.

The screenshot shows the selection screen for the transaction. It has a menu bar (Abgrenzungen, Bearbeiten, Springen, System, Hilfe) and a toolbar. The main area contains two groups of fields:

CITYFR	<input type="text"/>
CITYTO	<input type="text"/>
<hr/>	
AIRPFR	<input type="text"/>
AIRPTO	<input type="text"/>

At the bottom right, the status bar shows: B20 (2) (000) | hs0324 | INS | 06:15PM

You can process the user entries from the selection screen either at the AT `SELECTION-SCREEN` event, or later in the application logic. When the AT `SELECTION-SCREEN` event (PAI of the selection screen) has been processed, the program moves on to the next screen.

Using Selection Criteria in Programs

Using Selection Criteria in Programs

You can use selection criteria, which are entered in selection tables, for three different tasks in your report program:

[Using Selection Tables in the WHERE Clause \[Page 858\]](#)

[Using Selection Tables in Logical Expressions \[Page 859\]](#)

[Using Selection Tables with the CHECK Statement in GET Events \[Page 862\]](#)

Using Selection Tables in the WHERE Clause

To limit the database access of the Open SQL statements SELECT, UPDATE, and DELETE, you use the WHERE clause.

To use selection tables in a WHERE condition, write:

Syntax

..... WHERE <f> IN <seltab>.

<f> is the name of a database field (column of a database table) without a prefix and <seltab> is the selection table which is attached to this field. If you use this WHERE condition with an Open SQL statement (see [Specifying Conditions for Line Selection in the Program \[Page 560\]](#)), the system accesses only those lines of the addressed database table, where the contents of field <f> meets the selection criterion stored in <seltab>.

```
REPORT SAPMZTST.  
TABLES SPFLI.  
SELECT-OPTIONS AIRLINE FOR SPFLI-CARRID.  
SELECT * FROM SPFLI WHERE CARRID IN AIRLINE.  
  WRITE SPFLI-CARRID.  
ENDSELECT.
```

In the SELECT-OPTIONS statement of this example, the selection table AIRLINE is attached to the CARRID column of the database table SPFLI. The WHERE clause of the SELECT statement causes the system to check if the contents of the CARRID column meet the selection criteria stored in AIRLINE.

Assume that the report user enters two lines in the selection table, namely an interval selection and a single value selection, as follows:

SIGN	OPTION	LOW	HIGH
I	BT	DL	UA
E	EQ	LH	

Then, the report output appears as follows:

DL DL SQ UA UA UA

All airlines between "DL" and "UA", except "LH", are selected.

Using Selection Tables in Logical Expressions

Using Selection Tables in Logical Expressions

To control the internal flow of your program, you must program conditions with logical expressions. You can program special logical expressions using selection tables (see [Checking Selection Criteria \[Page 245\]](#)). The syntax is as follows:

Syntax

... <f> IN <seltab>....

The logical expression is true if the contents of the field <f> meet the selection limits stored in the selection table <seltab>. <f> can be any internal field or the column of a database table.

If the selection table <seltab> is attached to <f> with the SELECT-OPTIONS statement, you can use the following short form for the logical expression:

Syntax

... <seltab>....

This short form is not possible for selection tables defined by the RANGES statement.

```
REPORT SAPMZTST.
TABLES SPFLI.
SELECT-OPTIONS AIRLINE FOR SPFLI-CARRID.
WRITE: 'Inside', 'Outside'.
SELECT * FROM SPFLI.
  IF SPFLI-CARRID IN AIRLINE.
    WRITE: / SPFLI-CARRID UNDER 'Inside'.
  ELSE.
    WRITE: / SPFLI-CARRID UNDER 'Outside'.
  ENDIF.
ENDSELECT.
```

Assume that the report user enters two lines in the selection table, namely an interval selection and a single value selection, as follows:

SIGN	OPTION	LOW	HIGH
I	BT	DL	UA
E	EQ	LH	

Then, the report output appears as follows:

Using Selection Tables in Logical Expressions

Inside	Outside
	AA
	AA
DL	
DL	
	LH
	LH
	LH
	LH
	LH
	LH
	LH
	LH
	LH
SQ	
UA	
UA	
UA	

In the SELECT loop, all lines are read from the database table SPFLI. Using the IF statement, the program flow is branched into two statement blocks according to the logical expression. The short form IF AIRLINE is also possible in this program.

TABLES SPFLI.

SELECT-OPTIONS: S_CARRID FOR SPFLI-CARRID,
 S_CITYFR FOR SPFLI-CITYFROM,
 S_CITYTO FOR SPFLI-CITYTO,
 S_CONNID FOR SPFLI-CONNID.

SELECT * FROM SPFLI.

CHECK: S_CARRID,
 S_CITYFR,
 S_CITYTO,
 S_CONNID.

WRITE: / SPFLI-CARRID, SPFLI-CONNID,
 SPFLI-CITYFROM, SPFLI-CITYTO.

ENDSELECT.

After starting the program, the selection screen appears, on which the user might fill the input fields as follows:

S_CARRID	AA	To	UA	
S_CITYFR	FRANKFURT	To	LONDON	
S_CITYTO	BERLIN	To	NEW YORK	
S_CONNID	100	To	2500	

Then, the output appears as follows:

Using Selection Tables in Logical Expressions

LH	0400	FRANKFURT	NEW YORK
LH	0402	FRANKFURT	NEW YORK
LH	2402	FRANKFURT	BERLIN
LH	2436	FRANKFURT	BERLIN
LH	2462	FRANKFURT	BERLIN

In the SELECT loop, the system reads all lines from the database table SPFLI. From these lines, the system writes only those which meet the conditions in the selection tables onto the output screen. Otherwise, the system leaves the loop pass after the CHECK statement. The CHECK statement uses the short forms of the logical expressions. The long forms are:

CHECK: SPFLI-CARRID IN S_CARRID,
SPFLI-CITYFR IN S_CITYFR,
SPFLI-CITYTO IN S_CITYTO,
SPFLI-CONNID IN S_CONNID.

Using Selection Tables with the CHECK Statement in GET Events

You can use the CHECK statement to leave

- loops (see [Terminating a Loop Pass Conditionally \[Page 258\]](#))
- subroutines (see [Terminating Subroutines Conditionally \[Page 476\]](#))
- processing blocks (see [Leaving Processing Blocks Conditionally \[Page 1238\]](#))

Besides the possibility of using selection tables in logical expressions with the CHECK statement (see example in [Using Selection Tables in Logical Expressions \[Page 859\]](#)), ABAP provides a variant of the CHECK statement which you can use only after reading a line of a database table with a logical database.

To check whether the contents of that line meet the selection criteria stored in **all** selection tables that are attached to this database, use the check statement as follows:

Syntax

CHECK SELECT-OPTIONS.

With this statement, you can check the contents of the actual database table (addressed by GET) against **all** selection tables to which it is connected by using different SELECT-OPTIONS statements. This variant of the CHECK statement

- works only with selection criteria that are attached to database tables with the SELECT-OPTIONS statement
- should only be used in the processing blocks of GET events (see [GET <table> \[Page 1226\]](#)).

Since the CHECK statement cannot be used until after a line has been read by the logical database, you should use this variant only if the selections offered by the logical database are not sufficient to meet your requirements and the relevant table is not designated for dynamic selections.

The logical database F1S is attached to the following executable program (report).

REPORT SAPMZTST.

TABLES: SPFLI,SFLIGHT.

SELECT-OPTIONS: MAX FOR SFLIGHT-SEATSMAX,
OCC FOR SFLIGHT-SEATSOCC.

GET SFLIGHT.





WRITE: / SPFLI-CARRID, SPFLI-CONNID.

CHECK SELECT-OPTIONS.

WRITE: SFLIGHT-SEATSMAX, SFLIGHT-SEATSOCC.

For example, the report user fills the selection screen as follows:

Using Selection Tables with the CHECK Statement in GET Events

Carrier ID	AA	To	UA	
From	Frankfurt			
To	Berlin			
Departure date	01/01/95	To	08/31/95	
MAX		To		
OCC		To		

Then, the output appears as follows:

```

LH 2402
LH 2402
LH 2402
LH 2402
LH 2436      280      10
LH 2436      280      20
LH 2436
LH 2436
LH 2462      220      10
LH 2462      220      20
LH 2462
LH 2462

```

The system reads all lines from SFLIGHT that meet the selection criteria of the logical database. If these contents do not meet the self-defined selection criteria MAX and OCC, the system leaves the GET event before writing the contents of SEATSMAX and SEATSOCC onto the screen,.

Pre-Setting Selections Using Variants

This section describes

[What is a Variant? \[Page 865\]](#)

[Creating and Changing Variants \[Page 866\]](#)

[Using Variables with Variants \[Page 875\]](#)

[Running an Executable Program \(Report\) with a Variant \[Page 887\]](#)

What is a Variant?

What is a Variant?

When you start an executable program, ABAP usually offers you input fields for database-specific and report-specific selections on a selection screen. To select a certain set of data, you have to enter the corresponding values (see [Working with Selection Screens \[Page 795\]](#)).

If you want to run the same executable program with the same selections at regular intervals (for example, for monthly sales statistics), you would not want to enter the same values each time. So, ABAP offers you a possibility to combine the desired values for all these selections in one selection set. You can create as many different selection sets as you like for each executable program and they remain assigned only to the program in question. Such a selection set is called a variant.

A variant is an interface to the selection screen. Variants you use online may have different functions than those you use in background processing.

Using Variants Online

Online, starting an executable program via variant saves you work, since you do not have to enter the same selection set again and again each time the selection screen appears. In addition, using a variant minimizes input errors. By creating one variant with optimal values for each application of the program, you can guarantee that the resulting list is produced as precise and as fast as possible. Entering the exact values on the selection screen also reduces the runtime of the program.

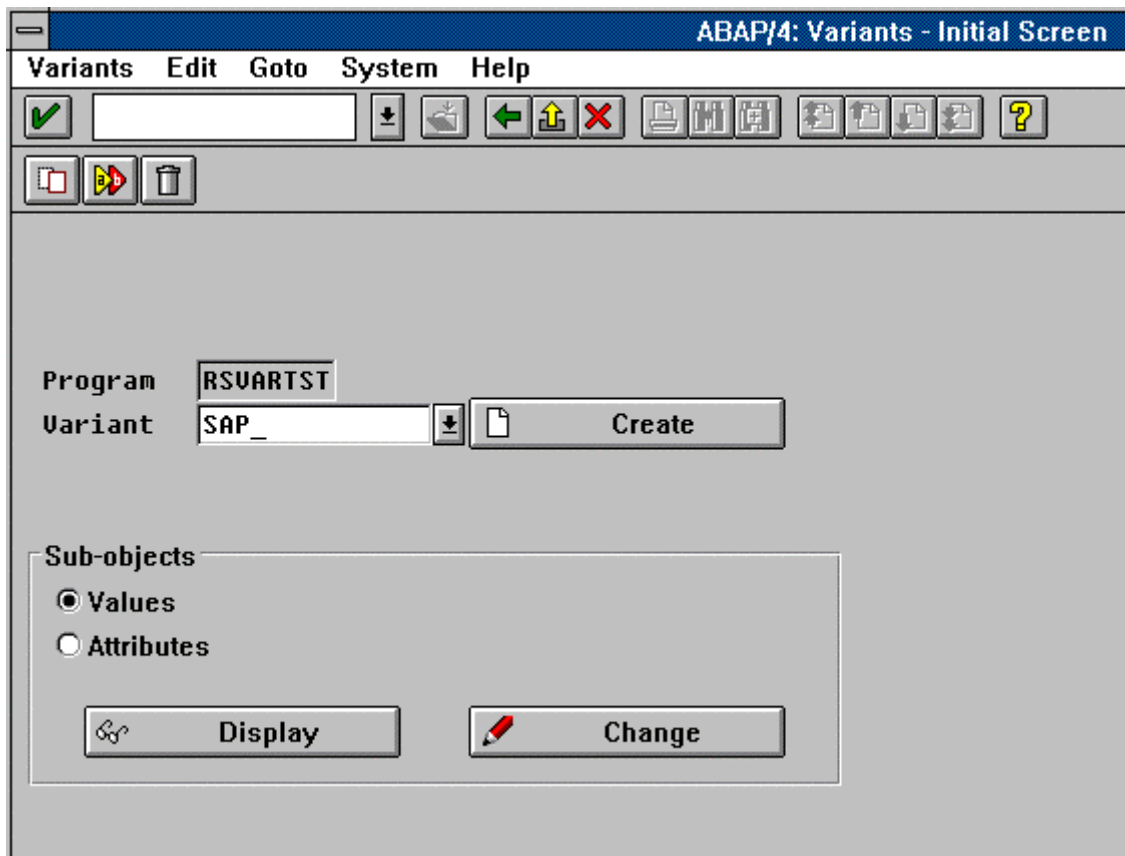
Using Variants in Background Processing

In background processing, a variant is the only possibility you have to pass values for the selections. Therefore, report programs executed in the background must be started via a variant (Exception: SUBMIT... VIA JOB). To prevent you from creating a new variant for each change of values, ABAP offers the possibility to supply a variant with variable values (see [Attributes of a Variant \[Page 870\]](#))

To ensure that an executable program (report) always starts via a variant, you can determine this in the program attributes.

Creating and Changing Variants

To create or change a variant, start from the *ABAP Editor: Initial screen*. Enter the name of the program you want to maintain a variant for, mark *Variants* and choose *Change*. The *ABAP: Variants - Initial Screen* appears. You can now display a list of existing variants for the program, create a new variant, or modify an existing one.



The following topics describe:

[Displaying a List of all Variants \[Page 867\]](#)

[Creating Variants \[Page 869\]](#)

[Attributes of a Variant \[Page 870\]](#)

[Changing Variants \[Page 872\]](#)

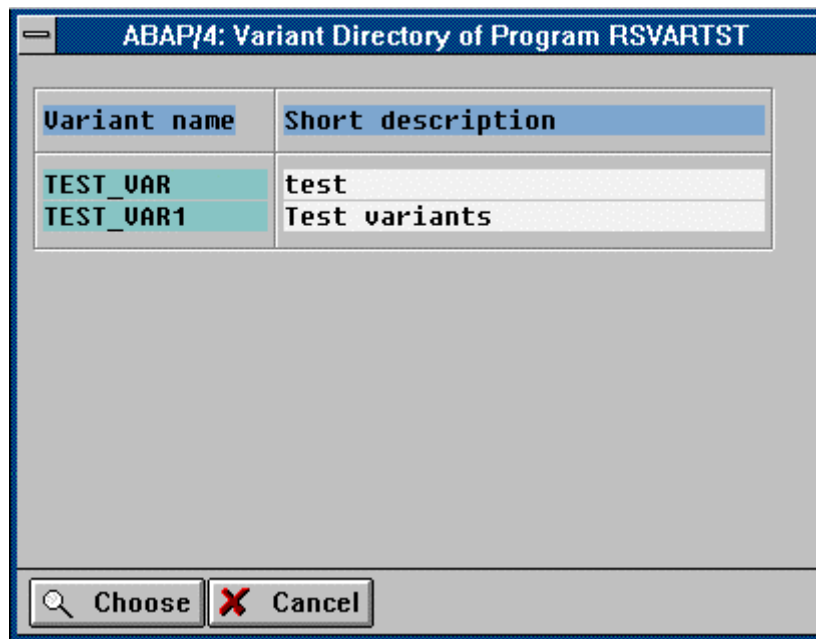
[Deleting Variants \[Page 873\]](#)

[Printing Variants \[Page 874\]](#)

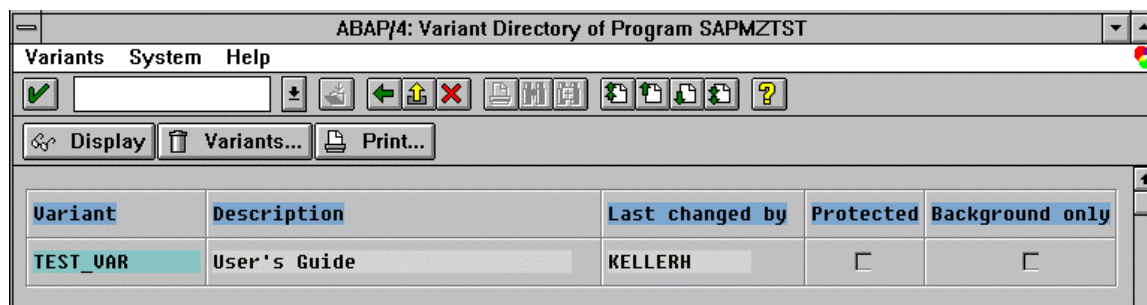
Displaying a List of all Variants

Displaying a List of all Variants

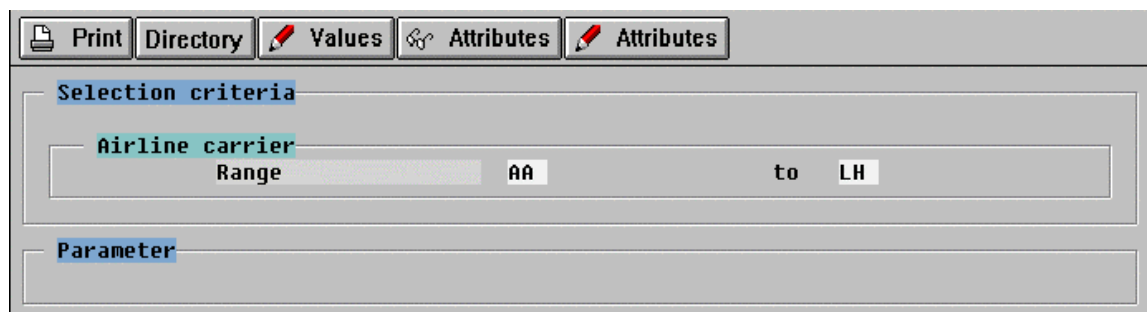
Before you create a new variant, you should check whether there is already a variant for your particular program that you could use or amend slightly. To proceed, click the possible entries button of the variant name field.



Or choose *Variants* → *Directory* on the initial screen:



From this screen, you can display, delete, or print variants. The display of a variant looks for example as follows:



To display the attributes, click the *Display Attributes* pushbutton.

Displaying a List of all Variants

To change a variant from the directory screen, choose either *Change Values* oder *Change Attributes*. You then see one of the screens described in [Creating Variants \[Page 869\]](#).

Creating Variants

Creating Variants

To create a new variant:

1. On the *ABAP Editor: Initial screen*, enter the name of the executable program you want to create a variant for, select *Variants*, and choose *Change*.
2. On the *ABAP: Variants - Initial Screen*, enter the name of the variant.

This name can comprise up to 14 alphanumeric characters. Do not use '&' or '%' in the name, since they result in an error message, and do not begin the name with SAP_, since these variants may be overwritten when updating the system.
3. Choose *Create*.

The selection screen appears.
4. Enter the desired selection values, including multiple selection and dynamic selection.
5. Choose *Continue*.

You then see an overview screen where you can specify attributes for your variant and save it (see [Attributes of a Variant \[Page 870\]](#)).

Note that when you create a new variant, you always have to fill in values as well as attributes. Therefore, you can save a new variant on the second screen only.

Attributes of a Variant

You maintain the attributes of a variant on the following screen:

Field attributes				
Fld name	Type	Protected	Invisible	Variable
NAME	P	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
DATE	P	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

You can enter the following attributes when creating a variant:

- *Description*
Enter a short, meaningful description of the variant. This may be up to 30 characters long.
- *Background only*
Specify whether you want to use the variant in background processing only, or in online environment as well.
- *Protected variant*
Mark this field if you want to protect your variant against being changed by other users.
- *Do not display variant*
Mark this field if you want the variant name to be displayed in the catalog only, but not in the F4 value list.

For the selections you cover in a variant, you can enter the following attributes:

- *Type*
The system displays whether the field is a parameter or a select option.
- *Protected*

Attributes of a Variant

Mark this field for each field on the selection screen you want to protect from being overwritten. Values that you mark this way are displayed to the users, but they cannot change them, that is they are not ready to accept input.

- *Invisible*

If you mark this column, the system will not display the corresponding field on the selection screen the user sees when starting the executable program.

- *Variable*

Mark this column if you want to set the value for this field at runtime. You can do this in the following three ways which are described in detail in [Using Variables with Variants \[Page 875\]](#):

- variable date calculation
- user-specific values
- values from Table TVARV

After you have entered all the attributes, save this setting. When you create a new variant, you must make entries on both the value and the attributes screen and you can save the new variant on the attributes screen only. However, if you want to change either values or attributes of an existing variant, you can save the variant on the corresponding screen.

Changing Variants

To change an existing variant, call the variant as described in [Creating Variants \[Page 869\]](#). Instead of using *Create*, however, choose an existing variant name and *Change*. You can now change the values or the attributes (see [Attributes of a Variant \[Page 870\]](#)).

After entering the changes, save the values or attributes on the corresponding screen.

Deleting Variants

Deleting Variants

To delete a variant, enter the report name and the variant name in the variants initial screen. Then, choose *Variants* → *Delete*.

The system first displays a window where you can confirm or cancel your decision.

It then confirms the action taken with an appropriate message.

To delete several variants in one action, choose *Variants* → *Directory*. Mark the variants you want to delete, and choose *Delete*.

Printing Variants

To print a variant, enter the name of the variant in the variants initial screen, choose the values for display (from the *ABAP: Variants - Initial Screen* or from the *Variants* menu), and click *Print*. Note that you cannot print the values if you are in change mode.

The *Print Screen List* screen appears.

If the print parameters do not agree with the default values displayed on the screen, then specify the parameters that are applicable to your department. You can find out the correct values from your system administrator. Mark the *Print immediately* field.

To print the variant, choose *Print*.

Using Variables with Variants

Using Variables with Variants

If you use variable values, you do not have to create a new variant for each change of the value set.

To supply a selection with a variable value, mark the *Variable* column of the desired selection(s) on the attributes screen and click on *Selection variables*. The following screen appears:

Using variables with variants includes three different options:

- Using variable date calculations (see [Using Variable Date Calculations \[Page 876\]](#)).

If, in a variant, you want to use, for example, the date of the day or the last day of last month, you can use variable date calculation.

- Using user-specific values (see [Using User-Specific Values \[Page 877\]](#)).

To fill certain selections with values that differ from user to user, you use a variant with user-specific values.

- Using variables defined in Table TVARV (see [Using Table TVARV \[Page 882\]](#)).

To fill certain selections with values that change according to the application, you use a variant which takes the variable values from Table TVARV. To avoid having to create a new variant and specify it in every background processing run following each minimal change in the selection data, you can simply assign a variable in Table TVARV to it and then change the values concerned in the table. This is particularly important for the values on the selection screen that are input-protected.

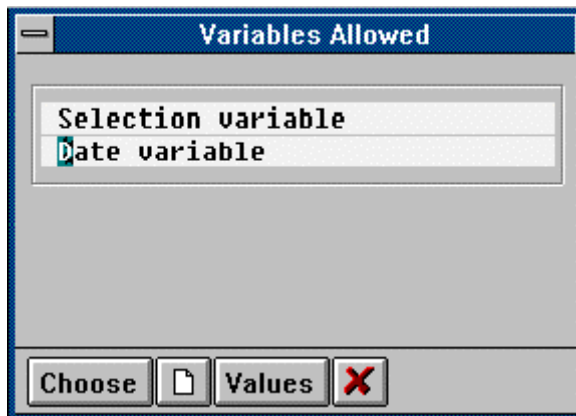
Using Variable Date Calculations

To use variable date calculation in a variant, you must define a date field as selection criterion or as parameter in your executable program (report).

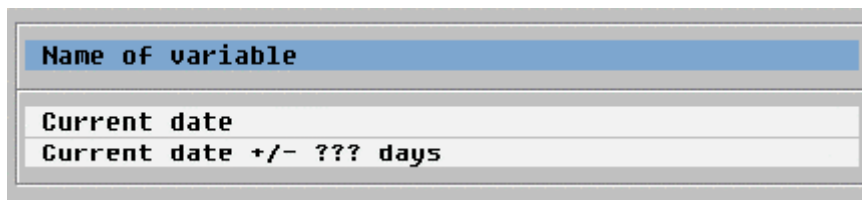
PARAMETERS DATE LIKE SY-DATUM.

You can then assign a variable to this date field.

To assign a variable to a date field, on the selection variables screen click on the possible entries button of the *Variable type* field. Choose *Date variable* in the following dialog box:



You get back to the selection variables screen. Click on the possible entries button of *Variable name* to get a list of proposals for date calculations.



If necessary, enter the desired values. To subtract days, you must enter them in the format of a number with an minus sign in the **last** position (for example 10-).

Note that you cannot maintain the proposals in the list further or add new ones.

Using User-Specific Values

Using User-Specific Values

By using user-specific values, you can cover any user-specific entries into selections with a variant. For each authorized user, these values are stored in certain tables and filled individually into the selections when the variant is executed.

With this method, the user does not have to enter never-changing values, such as the user number or company code, every time he or she starts the executable program (report). On the selection screen, the user must fill in only those fields whose values change from program execution to program execution. Several users can use the same variant.

Creating User-Specific Values

To fill selections with user-specific values, the master records of the users in question must contain the corresponding user parameter with an identity <pid>.

In the executable program (report), you then have to define a parameter or a selection criterion with the option...MEMORY ID <pid> with the corresponding parameter identity (see [Using Default Values from SAP Memory \[Page 812\]](#)).

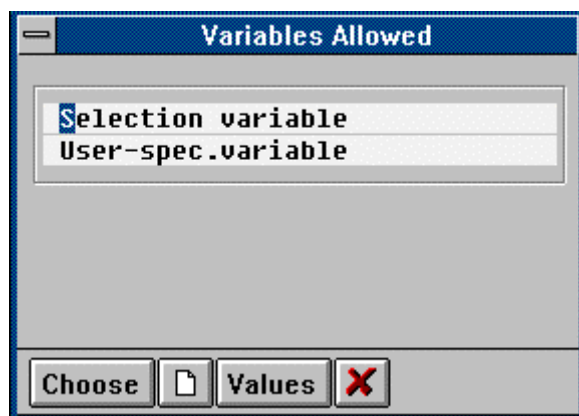
```
DATA: COMPANY_CODE(6).
```

```
...
```

```
SELECT-OPTIONS: CC LIKE COMPANY_CODE MEMORY ID BUK.
```

```
...
```

When creating the variant, on the attributes screen mark *Variable* for the corresponding selection and click *Selection variables*. On the appearing screen, click on the possible entries pushbutton of *Variable type* and choose *User-specific variable* in the following dialog box:.



You get back to the selection variables screen. The system automatically sets the appropriate long text of the parameter as *Variable name*.

At runtime, this user-specific variable receives the value(s) for the current user for the current parameter identity.

To change the values of existing user-specific variables, ABAP provides two ways:

- The programmer can use the function modules VARI_USER_VARS_* in the executable program (see [Changing Values of User-Specific Variables from the Program \[Page 881\]](#)) to change the variables of any user.

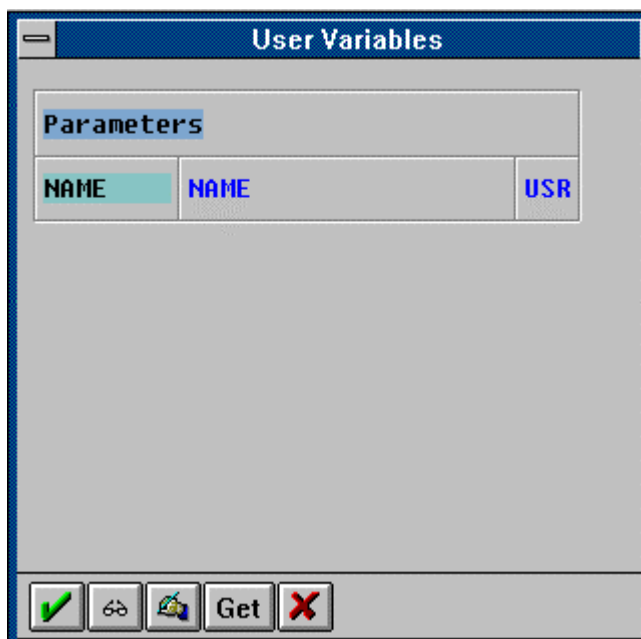
- The user can change the values of his own variables on the selection screen (see [Changing Values of User-Specific Variables Interactively \[Page 879\]](#)).

Changing Values of User-Specific Variables Interactively

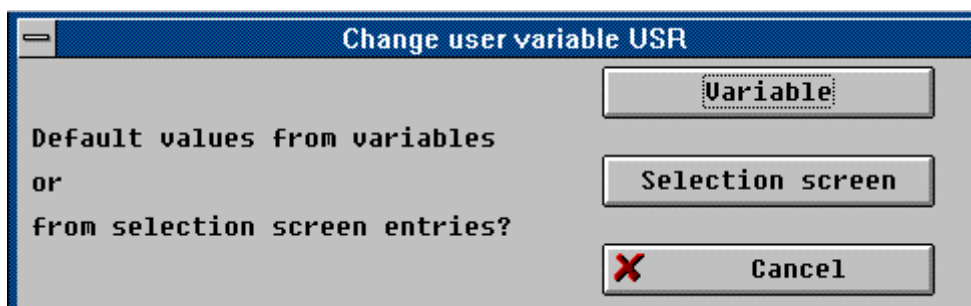
Changing Values of User-Specific Variables Interactively

Any user can change the values of his user-specific variables on the selection screen of the report. To do so, choose *Goto* → *User variables*.

You then see a window containing the user-specific selection criteria and parameters. You can now display or change the attached values.



If you choose *Change*, the system displays another window. Here, you can choose whether to change the values proposed by the variable or the entries on the selection screen.



In both cases, the system displays a window on which you can enter and save the desired values.

Changing Values of User-Specific Variables Interactively

The screenshot shows a dialog box titled "User Variables for ID USR". It contains three main sections: "Settings" with two unchecked checkboxes ("Upper/lower case" and "No check for valid range"); "Parameter value of user variables" with a text field containing "SMITH"; and "Values of user variables for selection criteria" with two empty text fields separated by "to", and a button with a right-pointing arrow. At the bottom, there are three buttons: a green checkmark, a yellow folder icon, and a red "X" icon, with the text "Selection options" between the folder icon and the "X" icon.

Beware that these changes affect all variants that use the corresponding user-specific variables.

Changing Values of User-Specific Variables from the Program

Changing Values of User-Specific Variables from the Program

To change the current values of user-specific variables from the executable program, you can use a set of function modules.

Function module	Function
VARI_USER_VARS_GET	Reads existing variable values
VARI_USER_VARS_SET	Changes existing variable values
VARI_USER_VARS_COPY	Copies existing variable values
VARI_USER_VARS_DELETE	Deletes variable values
VARI_USER_VARS_RENAME	Renames variable values
VARI_USER_VARS_DIALOG	Allows entering variable values interactively

Use *Edit* → *Insert statement* → *CALL FUNCTION* in the ABAP Editor to insert these function modules in your program.

Using Table TVARV

Using values from Table TVARV is especially useful in background processing. Table TVARV saves you the effort of creating a new variant for each minor change of values or of changing an existing variant again and again, since you only have to change the values stored in TVARV. Beware, however, that every change of a value in Table TVARV affects all variants using this particular variable.

The following topics describe

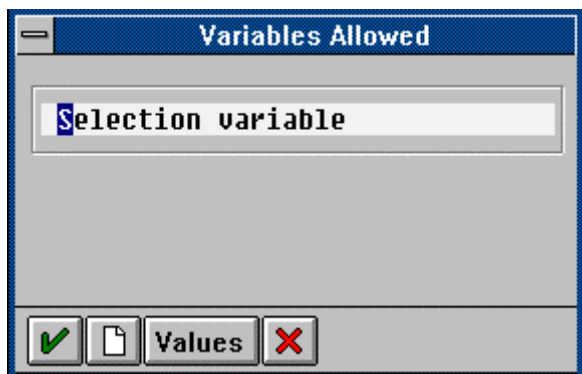
[Creating New Entries in Table TVARV \[Page 883\]](#)

[Changing Entries in Table TVARV \[Page 885\]](#)

Creating New Entries in Table TVARV

Creating New Entries in Table TVARV

To connect a selection to an entry in Table TVARV, on the attributes screen mark *Variable* and choose *Maintain Variable*. On the subsequent screen, choose *Selection variable* as *Variable type*.



Then, enter the name of the variable:

- If you want to use an existing variable and you know the name, enter it directly in the *Variable name* field.
- If you want to use an existing variable and you don't know the name, click the possible entries pushbutton for the *Name* field and choose the desired variable from the list displayed.

To see the values of a variable from this list, mark the variable and click *Values*. The system then branches to the entries in Table TVARV.

- If you want to create a new variable, enter the name and choose *ENTER*.

The system tells you that this variable does not exist and asks you whether to branch to the maintenance screen. If you confirm, the initial table screen appears, on which Table TVARV is preselected.

Click the *Maintain* button.

The table maintenance screen for Table TVARV appears. Enter the name and type of the variable and choose *Create*.

If you want to create a new variable for a select option, you see a blank screen on which you can enter lower and upper limits, option, as well as inclusive and exclusive criteria.

If you want to create a variable for a parameter, a dialog window appears in which you can enter the parameter value:

Creating New Entries in Table TVARV

The screenshot shows a SAP dialog box titled "Maintain Table TVARV: Selection Variables". At the top, there is a toolbar with buttons: a document icon, "Change", "Display", "Directory", a red X icon, and "Delete". Below the toolbar, the text "Variable." is followed by a text input field containing "hktst". Underneath, the "Type" section has two radio buttons: "Parameter" (which is selected) and "Select option". The main area of the dialog box contains the text "Parameter value" followed by a checkbox labeled "Only upper case" which is currently unchecked. Below this is a large empty text input field. At the bottom of the dialog box, there are two buttons: "Save" and "Back".

Save your new variable and return to the variable maintenance screen. You can now enter the next variable name or save the variables attached so far and return to the attributes screen.

Changing Entries in Table TVARV

Changing Entries in Table TVARV

Beware that changes to the values of Table TVARV affect all variants that use these particular variables.

To change values from Table TVARV, choose *System* → *Services* → *Table maintenance*.

Then, enter the name TVARV on the *Table Maintenance* screen and choose *Maintain*.

The maintenance screen for Table TVARV appears, asking you for the name and type of the variable. From this screen, you can display and change values of existing variables, create new variables, copy variable values, or delete them. The *Catalog* button provides you with an overview of existing variables.

Displaying Variable Values

To display the value of a variable, proceed as follows:

- If you know the name and type of the variable, you can call it directly. Enter the name and type and then choose *Display*.
- If you want to view a list of existing variables, choose *Directory*.

This brings you to a selection screen where you can specify which variables among the set of existing variables you want to view in a list. If you make no entry here, the system lists all existing variables.

Now choose *Execute* to obtain the requested list of variables.

Double-click the desired variable to select it for display.

If you select a parameter, the system displays a window containing the current value of the parameter.

If you select a selection criterion, the system displays a new screen containing the values.

Changing Variable Values

To change the value of a variable, proceed as follows:

1. If you know the name and type of the variable, you can call it directly. Enter the name and type and then choose *Change*.

To see a list of existing variables, choose *Directory*.

This brings you to a selection screen on which you can specify which variables among the set of existing variables you want to see in a list. If you make no entry here, the system lists all existing variables.

Double-click the desired variable to select it.

If you select a parameter, the system displays a window containing the current value of the parameter.

If you select a selection criterion, the system displays a new screen containing the values.

2. Change the values by overwriting the old values.
3. Save the new values.

The changed values now appear in Table TVARV. At runtime, the system displays the values in the corresponding selections.

Creating Variable Values

To create a new variable with values in the table directly, specify the name and type of the variable and then choose *Create*.

If you want to create a new selection criterion, you see a blank screen on which you can enter lower and upper limits, option as well as inclusive and exclusive criteria.

If you want to create a parameter, a window appears in which you can enter the parameter value.

Save your new variable.

Do not forget to enter this new variable name into a variant so that you can use it.

Copying Variable Values

To copy a variable, enter the name and type of the source variable and then choose *Copy*.

You then see a window in which you can specify the name of the copy. Save this by choosing *Copy*.

You can now change the new variable as you like. However, do not forget to enter this new variable name into the variant so that you can use it.

You can also copy variables from the list displayed by using the *Directory* option.

Deleting Variables

To delete a variable, enter the name and type of the variable and then choose *Delete*.

The system displays a window in which you can confirm or cancel your decision.

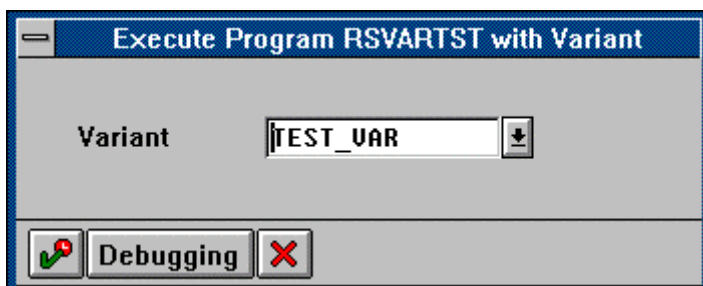
You can also delete variables from the list displayed using the *Directory* option.

Running an Executable Program (Report) with a Variant

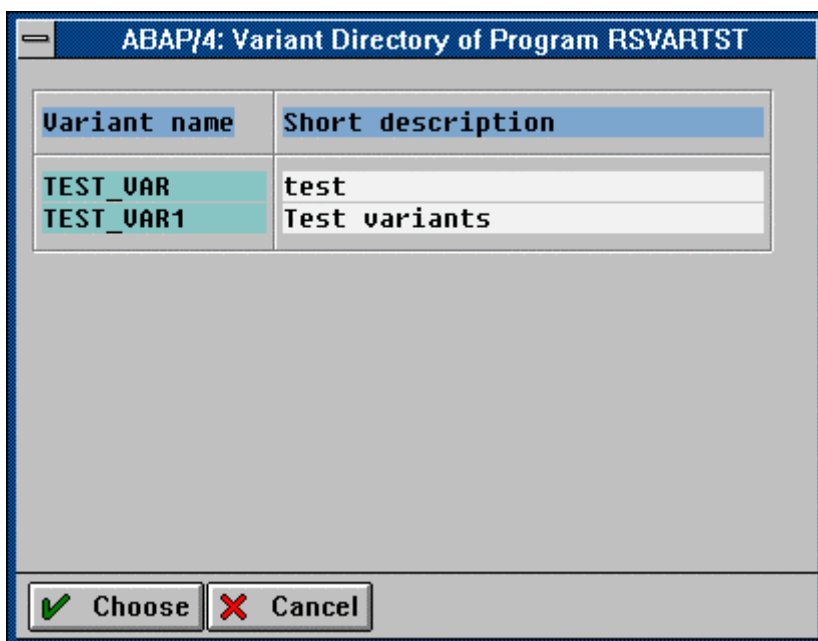
Running an Executable Program (Report) with a Variant

To execute an executable program (report) with a variant, enter the name of the program on the initial screen of the ABAP Editor and choose *Execute with variant*.

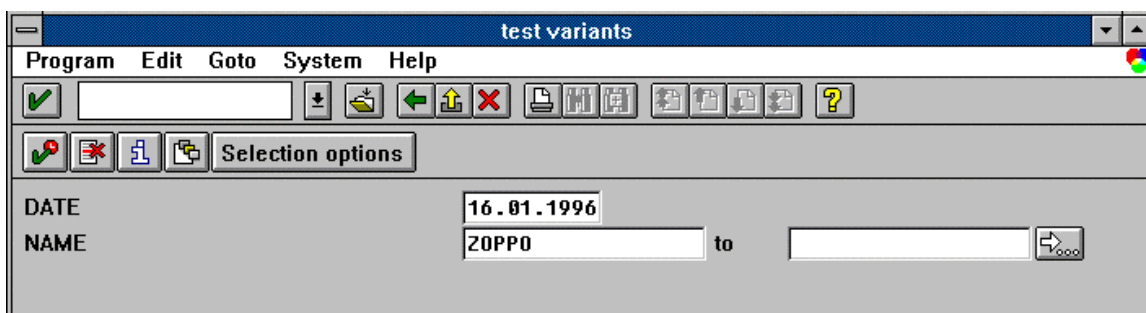
A dialog window appears, on which you must enter the name of the desired variant.



If you don't know the name, click the possible entries pushbutton. The system displays a list of the existing variants. Mark the desired variant and copy it. Choose *Execute* on the dialog window to start the executable program (report) with the chosen variant.



The selection screen of the report appears. Those fields that are covered by the variant, already contain values.



Running an Executable Program (Report) with a Variant

If necessary, fill in the other fields and choose *Execute*. The system displays the desired list.

Lists

Lists

Creating Simple Lists with the WRITE Statement

This section describes how to create simple output lists on the screen. To do this, you use the WRITE statement.

Here, you learn

[The WRITE Statement \[Page 891\]](#)

[Positioning WRITE Output on the Screen \[Page 894\]](#)

[Formatting Options \[Page 896\]](#)

[Outputting Symbols and Icons on the Screen \[Page 898\]](#)

[Lines and Blank Lines on the Output Screen \[Page 899\]](#)

[Outputting Field Contents as Checkboxes \[Page 900\]](#)

[Using WRITE via a Statement Structure \[Page 901\]](#)

When you run executable programs (reports), the system automatically displays the list generated by the program, when the program processing has finished. If you are running a dialog program (module pool), you can send the list to the screen using the LEAVE TO LIST-PROCESSING statement.

ABAP allows you to generate more complex and effective output lists, both on the screen and on paper, than that covered here. The following sections build on this introduction.

The WRITE Statement

The WRITE Statement

The basic ABAP statement for outputting data on the screen is WRITE.

Syntax

WRITE <f>.

This statement outputs the field <f> to the current list in its standard output format. By default, the list is displayed on the screen.

The field <f> can be

- any data object (see [Data Objects \[Page 112\]](#))
- a field symbol or formal parameter (see [Working with Field Symbols \[Page 336\]](#))
- a text symbol (see [Working with Text Elements \[Page 146\]](#))

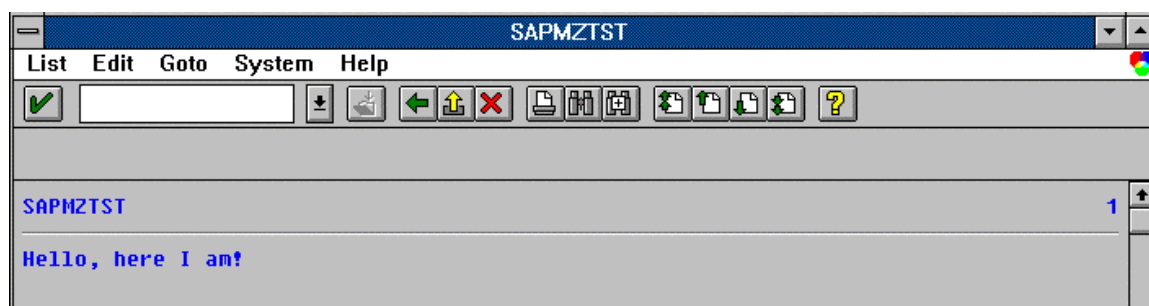
You can print the current output list directly from the output screen by choosing *Print*.

If a selection screen is defined for the program (see [Working with Selection Screens \[Page 795\]](#)), you can choose *Execute and print* on the selection screen. Then, the list is not output to the screen, but sent directly to a printer.

PROGRAM SAPMZTST.

WRITE 'Hello, here I am!'.

When you start this program, the system leaves the current screen (this may be the *ABAP Editor: Initial Screen*) and branches to the output screen as follows:



The output screen has the same name as the title of the program specified in the program attributes (see [Maintain Program Attributes \[Page 74\]](#)).

The first line on the screen contains the list header. By default, the list header is the same as the title of the program. However, you can maintain the list header independently from the program title outside the actual program. For further information about this, see [Working with Text Elements \[Page 146\]](#). The current page number (1) appears on the right.

The list header is followed by one horizontal line and then the output is displayed.

You can choose *Search* to search for specific patterns.

The WRITE Statement

On the screen, the output is normally left-justified. If you use several WRITE statements, the output fields are displayed one after the other, each separated by one column (i.e. one blank). If there is not enough space for an output field on the current line, a new line is started.

```
PROGRAM SAPMTEST.
```

```
TABLES SPFLI.
```

```
.....
```

```
WRITE: 'COMPANY: ', SPFLI-CARRID.
```

Note the use of the colon and the commas (see [Syntax Structure \[Page 91\]](#)).

The program fragment in this example outputs two fields, the literal 'COMPANY: ' and the component CARRID of the table work area SPFLI, to the screen:

```
COMPANY:   AA
```

The format of data fields on the output screen depends on their data type (see [Elementary Data Types - Predefined \[Page 106\]](#)).

Output format of predefined data types

Data type	Output length	Positioning
C	field length	left-justified
D	8	left-justified
F	22	right-justified
I	11	right-justified
N	field length	left-justified
P	2 * field length (+1)	right-justified
T	6	left-justified
X	2 * field length	left-justified

The numeric data types F, I, and P are right-justified and padded with blanks on the left. If there is sufficient space, thousands separators are also output. If a type P field contains decimal places, the default output length is increased by one.

With the data type D, the internal format of a date differs from its output format. When you use the WRITE statement for outputting data, the system automatically outputs dates of type D in the format specified in the user's master record (e.g. DD/MM/YYYY or MM/DD/YYYY).

```
PROGRAM SAPMTEST.
```

```
DATA NUMBER TYPE P VALUE '-1234567.89' DECIMALS 2.
```

```
WRITE: 'Number', NUMBER, 'is packed'.
```

The output appears as follows:

```
Number      1,234,567.89- is packed
```

The WRITE Statement

The field NUMBER has a total length of 13, i.e. 9 digits (including the decimal point), the leading minus sign, and two commas as separators. The output length of the NUMBER field is $2 \cdot 8 + 1 = 17$ because the field length of a type P field is 8. The superfluous positions are filled with four blanks. This means that there are five blanks between the literal 'Number' and the number itself.

Positioning WRITE Output on the Screen

You can position the output of a WRITE statement on the screen by making a format specification before the field name as follows:

Syntax

WRITE AT [/][<pos>][(<len>)] <f>.

where

- the slash '/' denotes a new line,
- <pos> is a number or variable up to three digits long denoting the position on the screen,
- <len> is a number or variable of up to three digits long denoting the output length.

If the format specification contains only direct values (i.e. no variables), you can omit the keyword AT.

```

/
WRITE 'First line.'.
WRITE 'Still first line.'
WRITE / 'Second line.'
WRITE /13 'Third line.'

```

This generates the following output on the screen:

```

First Line. Still first line.
Second line.
          Third line.

```

If you specify a certain position <pos>, the field is always output in that position regardless of whether or not there is enough space available or whether other fields are overwritten.

```

/
DATA: LEN TYPE I VALUE 10,
      POS TYPE I VALUE 11,
      TEXT(10) VALUE '1234567890'
WRITE 'The text ----- appears in the text.'.
WRITE AT POS(LEN) TEXT.

```

This produces the following output on the screen:

```

The text -1234567890- appears in the text.

```

If the output length <len> is too short, fewer characters are displayed. Numeric fields are truncated on the left and prefixed with an asterisk (*). All other fields are truncated on the right, but no indication is given that the field is shorter.

```

/
DATA: NUMBER TYPE I VALUE 1234567890,
      TEXT(10) VALUE 'abcdefghij'.
WRITE: (5) NUMBER, /(5) TEXT.

```

Positioning WRITE Output on the Screen

This produces the following output:

```
*7890
```

```
abcde
```

In the default setting, you cannot create empty lines with the WRITE statement. You learn more about empty lines and how to change the default setting under [Creating Blank Lines \[Page 1012\]](#) in the section 'Creating Lists'.

```
WRITE:   'One',  
        / '   ',  
        / 'Two'.
```

The output looks as followss:

```
One  
Two
```

The System suppresses lines that contain nothing but empty spaces.

Formatting Options

You can use various formatting options with the WRITE statement.

Syntax

WRITE.... <f> <option>.

Formatting options for all data types

Option	Purpose
LEFT-JUSTIFIED	Output is left-justified.
CENTERED	Output is centered.
RIGHT-JUSTIFIED	Output is right-justified.
UNDER <g>	Output starts directly under the field <g>.
NO-GAP	The blank after the field <f> is omitted.
USING EDIT MASK <m>	Specifies a format template <m>.
USING NO EDIT MASK	Deactivates a format template specified in the ABAP Dictionary.
NO-ZERO	If a field contains only zeros, these are replaced by blanks. For type C and N fields, leading zeros are replaced automatically.

Formatting options for numeric fields

Option	Purpose
NO-SIGN	The leading sign is not output.
DECIMALS <d>	<d> defines the number of digits after the decimal point.
EXPONENT <e>	In type F fields, the exponent is defined in <e>.
ROUND <r>	Type P fields are multiplied by 10**(-r) and then rounded.
CURRENCY <c>	Format according to currency <c> in table TCURX.
UNIT <u>	The number of decimal places is fixed according to the unit <u> specified in table T006 for type P fields.

Formatting options for date fields

Option	Purpose
DD/MM/YY	Separators as defined in user's master record
MM/DD/YY	Separators as defined in user's master record
DD/MM/YYYY	Separators as defined in user's master record

Formatting Options

MM/DD/YYYY	Separators as defined in user's master record
DDMMYY	No separators.
MMDDYY	No separators.
YYMMDD	No separators.

For more detailed information on formatting options and the exclusion principles within some of these options, see the keyword documentation of the WRITE statement.

Below are some examples of formatting options. For more examples, see the section [Creating Complex Lists \[Page 938\]](#). The decimal character and thousands separators (period or comma) of numeric fields are defined in the user's master record

ABAP code	Screen output
DATA: G(5) VALUE 'Hello', F(5) VALUE 'Dolly'. WRITE: G, F. WRITE: /10 G, / F UNDER G. WRITE: / G NO-GAP, F.	Hello Dolly Hello Dolly HelloDolly
DATA TIME TYPE T VALUE '154633'. WRITE: TIME, /(8) TIME USING EDIT MASK '__:__:__'.	154633 15:46:33
WRITE: '000123', / '000123' NO-ZERO.	000123 123
DATA FLOAT TYPE F VALUE '123456789.0'. WRITE FLOAT EXPONENT 3.	123456.789E+03
DATA PACK TYPE P VALUE '123.456' DECIMALS 3. WRITE PACK DECIMALS 2. WRITE: / PACK ROUND -2, / PACK ROUND -1, / PACK ROUND 1, / PACK ROUND 2.	123.46 12,345.600 1,234.560 12.346 1.235
WRITE: SY-DATUM, / SY-DATUM YYMMDD.	06/27/1995 950627

Apart from the formatting options shown in the above tables, you can also use the formatting options of the FORMAT statement. These options allow you to specify the intensity and color of your output. For further information about this, see [The FORMAT Statement \[Page 996\]](#).

Outputting Symbols and Icons on the Screen

You can output symbols or R/3 icons on the screen by using the following syntax:

Syntax

```
WRITE <symbol-name> AS SYMBOL.
```

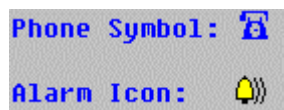
```
WRITE <icon-name> AS ICON.
```

The names of symbols and icons (<symbol-name> and <icon-name>) are system-defined constants which are specified in the include programs <SYMBOL> and <ICON> (the angle brackets are part of the name). The includes also contain a short description of the symbols and icons. The easiest way to output symbols and icons is to use a statement structure (see the example in [Using WRITE via a Statement Structure \[Page 901\]](#))

To make symbols and icons available to your program, you must import the appropriate include or the more comprehensive include <LIST> in your program. For further information about importing include programs, see [Using Include Programs \[Page 443\]](#).

```
INCLUDE <SYMBOL>.  
INCLUDE <ICON>.  
WRITE: / 'Phone Symbol:', SYM_PHONE AS SYMBOL.  
SKIP.  
WRITE: / 'Alarm Icon: ', ICON_ALARM AS ICON.
```

This produces the following output:



```
Phone Symbol: ☎  
Alarm Icon: 🔔
```

You can replace both the above INCLUDE statements with one single INCLUDE statement

```
INCLUDE <LIST>.
```

Lines and Blank Lines on the Output Screen

Horizontal lines

You can generate horizontal lines on the output screen by using the following syntax:

Syntax

```
ULINE [AT [/][<pos>][(<len>)]].
```

This is equivalent to

```
WRITE [AT [/][<pos>][(<len>)]] SY-ULINE.
```

The format specifications after AT are exactly the same as the format specifications described for the WRITE statement in [Positioning WRITE Output on the Screen \[Page 894\]](#).

If there are no format specifications, the system starts a new line and fills it with a horizontal line. Otherwise, horizontal lines are output as specified.

Another way of generating horizontal lines is to type the appropriate number of hyphens in a WRITE statement as follows:

```
WRITE [AT [/][<pos>][(<len>)]] '-----...'
```

Vertical lines

You generate vertical lines on the output screen by using the following syntax:

Syntax

```
WRITE [AT [/][<pos>]] SY-VLINE.
```

or

```
WRITE [AT [/][<pos>]] '|'
```

Blank lines

You can generate blank lines on the screen by using the following syntax:

Syntax

```
SKIP [<n>].
```

Starting on the current line, this statement generates <n> blank lines on the output screen. If no value is specified for <n>, one blank line is output. In the standard setting, you cannot create empty lines with the WRITE statement alone.

To position the output on a specific line on the screen use:

Syntax

```
SKIP TO LINE <n>.
```

This statement allows you to move the output position upwards or downwards.

For more information and examples, see [Creating Complex Lists \[Page 938\]](#).

Outputting Field Contents as Checkboxes

You can output the first character of a field as a checkbox on the output screen by using the following syntax:

Syntax

WRITE <f> AS CHECKBOX.


If the first character of the field <f> is an "X", the checkbox is displayed filled. If the first character is SPACE, the checkbox is displayed blank.

The checkboxes that are created by this statement are input enabled by default. With other words, the user can fill them or make them empty by mouse clicks. How you can make output fields input enabled or disabled is explained under [Enabling Fields for Input \[Page 1003\]](#). Input enabled fields are essentially important in interactive lists that allow a dialog with the user (see [Interactive Lists \[Page 1030\]](#)).

```
DATA: FLAG1    VALUE ' ',
      FLAG2    VALUE 'X',
      FLAG3(5) VALUE 'Xenon'.

WRITE: / 'Flag 1 ', FLAG1 AS CHECKBOX,
       / 'Flag 2 ', FLAG2 AS CHECKBOX,
       / 'Flag 3 ', FLAG3 AS CHECKBOX.
```

This produces the following output list:



The screenshot shows a list of three items, each with a label and a checkbox. The labels are 'Flag 1', 'Flag 2', and 'Flag 3'. The checkboxes are represented by small squares. The first checkbox is empty, while the second and third checkboxes are filled with an 'X'.

Flag 1	<input type="checkbox"/>
Flag 2	<input checked="" type="checkbox"/>
Flag 3	<input checked="" type="checkbox"/>

For FLAG2 and FLAG3, the checkboxes are filled because the first character of these fields is "X". The user can change the contents of the checkboxes by mouse clicks.

Using WRITE via a Statement Structure

Using WRITE via a Statement Structure


The R/3 System includes a useful facility for trying out all options and output formats of the WRITE statement and inserting them into your program. To proceed, choose *Edit → Insert Statement...* in the ABAP Editor and then select WRITE in the relevant dialog box (see [Inserting Ready-Made Keyword Structures \[Page 101\]](#)):

WRITE

When you have confirmed your selection with Enter, you see the following screen:

The screenshot shows the 'Assemble a WRITE Statement' dialog box. It features a title bar and a menu bar with 'System' and 'Help'. Below the menu bar is a toolbar with icons for Copy, Display, Initialize, and others. The main area is divided into sections: 'Output from' with radio buttons for Fld, Symbol, Icon, Line, and Checkbox, each with a corresponding text field; 'Output to' with a radio button for Struct. and a 'Select components' button; and a section for formatting options including checkboxes for 'to new line', 'without trailing blank', 'under fld', 'Color', 'Intensified display', 'Inverse display', 'Display as input field', and 'Alignment', along with radio buttons for 'from col.', 'in length', and alignment options (Left-just., Centered, Right-just.). A 'More formatting options' button is at the bottom right.

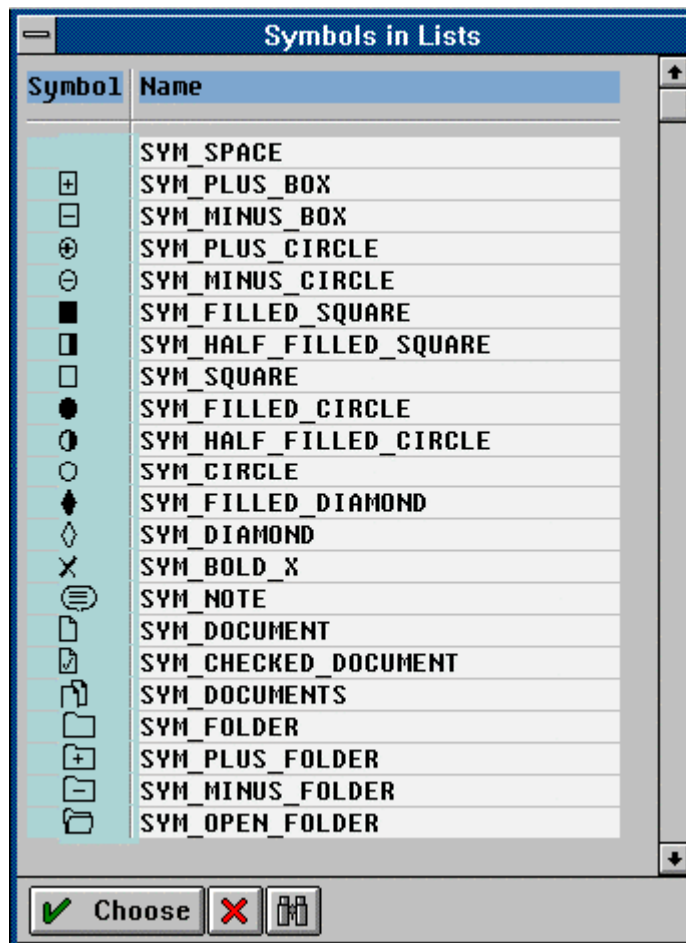
On this screen, you can

- determine the output format of an internal field by entering its name or a literal in the field *Fld*. You then choose the formatting options on this screen or on another screen which you can access by selecting *More formatting options* .
- generate the WRITE statements for symbols, icons, lines, and checkboxes simply by selecting the appropriate fields.

Using WRITE via a Statement Structure

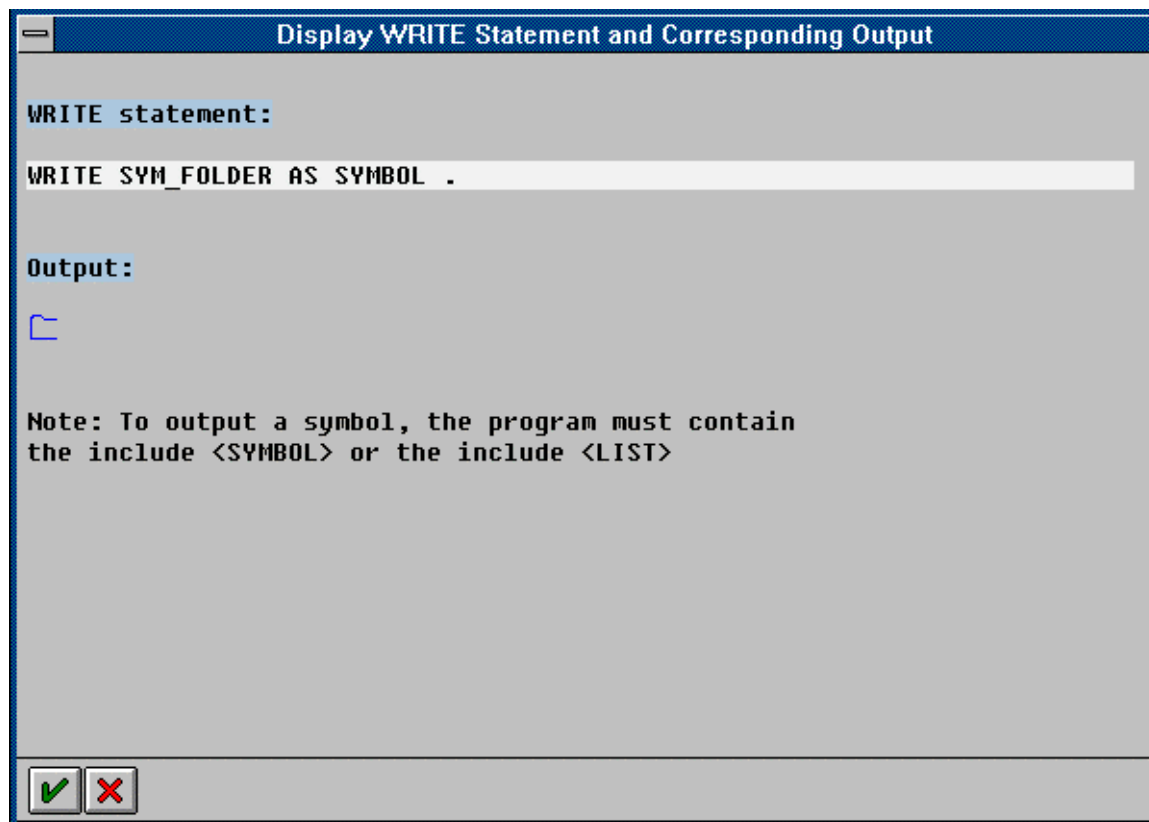
- generate the WRITE statements for components of structures defined in the ABAP-Dictionary. This is useful, for example, after executing a SELECT statement (see [Reading Data from Database Tables \[Page 542\]](#)).

On the screen *Assemble a WRITE-Statement*, select the radio button *Symbol* and then *Display*. You then see the following dialog box:



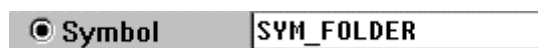
Here, you can choose a symbol, e.g. SYM_FOLDER. The next dialog box displays the relevant WRITE statement and the resulting output on the output screen:

Using WRITE via a Statement Structure



Also displayed is a note informing you that you need an include program in your program (see [Outputting Symbols and Icons on the Screen \[Page 898\]](#)).

After pressing *Continue*, you then see that the field *Symbol* on the *Assemble a WRITE Statement* screen now contains a value:



If you now choose *Execute*, the following text is inserted into your program:

```
WRITE SYM_FOLDER AS SYMBOL.
```

On the *Assemble a WRITE Statement* screen, select the radio button *Structure* and enter the following in the appropriate input field:



Then, choose *Select components*. On the next screen, you can select the components of the ABAP-Dictionary structure SFLIGHT you want to output with WRITE, e.g.:

Using WRITE via a Statement Structure

Fields of table SFLIGHT			
Sel. Key Field name			
<input type="checkbox"/>	X	MANDT	Client
<input checked="" type="checkbox"/>	X	CARRID	Airline carrier Id
<input checked="" type="checkbox"/>	X	CONNID	Flight connection Id
<input checked="" type="checkbox"/>	X	FLDATE	Flight date
<input checked="" type="checkbox"/>		PRICE	Ticket price
<input type="checkbox"/>		CURRENCY	Local currency of airline
<input checked="" type="checkbox"/>		PLANETYPE	Plane type
<input type="checkbox"/>		SEATSMAX	Maximum capacity
<input checked="" type="checkbox"/>		SEATSOCC	Occupied seats
<input type="checkbox"/>		PAYMENTSUM	Total of current bookings

If you adopt this selection, the following WRITE statement is inserted into your program:

```
WRITE: SFLIGHT-CARRID,  
      SFLIGHT-CONNID,  
      SFLIGHT-FLDATE,  
      SFLIGHT-PRICE,  
      SFLIGHT-PLANETYPE,  
      SFLIGHT-SEATSOCC.
```


Formatting Data

Formatting Data

Data often needs to be formatted before a list can be created. This means that any program which reads and presents data must sort data, calculate totals, count items in lists, and so on. You can read the data to be refined from database tables or from sequential files, or you can create generic data in the program.

The topic below contains an example for refining data:

[Example for Refined Data \[Page 906\]](#)

The current section deals with refining the data of a dataset after having created the dataset. The refining process is independent of the process of retrieving data. In the first step, you create a dataset; in the second step, you refine the dataset.

Exceptions of this rule are the processes of creating your own datasets generically or of directly accessing databases using SQL statements. Sometimes, data can be sufficiently refined during its retrieval. For an example, see the SQL report that refines data during retrieval in

[Refining Data During Reading \[Page 909\]](#)

When using logical databases to access database tables, when reading data from sequential files, or if the options of Open SQL are not comprehensive enough, the retrieved data often appears in a sequence and structure you need to refine. To refine this data at a later time, you store it during retrieval in condensed form in a temporary dataset.

Creating and Refining Datasets

You use temporary datasets to provide selected data for refining after the retrieval. ABAP offers two methods of creating datasets in the storage: internal tables and extract datasets. Your choice depends on the task you want to accomplish.

Internal Tables

Use internal tables if you want the datasets to map the underlying data structures as closely as possible, and if you want to access individual data directly. For examples demonstrating how to use internal tables for refining data, see:

[Refining Data Using Internal Tables \[Page 911\]](#)

Extract Datasets

An extract is a sequential dataset you can create with a program. Use extracts if you want to process large amounts of data as a whole several times. The topic below describes how to create extract datasets, how to fill them with data, and, finally, how to refine this data.

[Refining Data Using Extract Datasets \[Page 916\]](#)

Example for Refined Data

For many report evaluations, the sequence in which you want to process data may differ from the sequence in which it is stored. Since the results of read operations reflect the sequence in which data is stored, you must re-sort the entire data material you selected into the desired sequence.

A typical result of refining data in the context of a flight reservation application is the creation of a list, designed to contain booking information for each flight number. The flight connections are to be sorted by departure city, the flights by date, and the customer data by class and smoker/non-smoker. For each flight, the total number of passengers and the overall luggage weight are to be determined.

A section of the resulting list may look like this:

Example for Refined Data

LH 2407 from BERLIN to FRANKFURT			
Date: 1996/01/09	Book-ID	Smoker	Class
	00000002		C
	00000011		C
	00000013		C
	00000014		C
	00000025		C
	00000007	X	C
	00000022	X	C
	00000006		F
	00000010		F
	00000016		F
	00000027		F
	00000030		F
	00000003	X	F
	00000019	X	F
	00000023	X	F
	00000001		Y
	00000004		Y
	00000005		Y
	00000009		Y
	00000012		Y
	00000015		Y
	00000017		Y
	00000020		Y
	00000021		Y
	00000026		Y
	00000029		Y
	00000031		Y
	00000032		Y
	00000008	X	Y
	00000018	X	Y
	00000024	X	Y
	00000028	X	Y
Number of bookings: 32			
Total luggage weight: 622 KG			
Date: 1996/01/10	Book-ID	Smoker	Class
	00000007		C
	00000011		C

You find four different programs that create exactly this list under:

[Refining Data During Reading \[Page 909\]](#)

[Refining Data Using Flat Internal Tables \[Page 912\]](#)

[Refining Data Using Nested Internal Tables \[Page 914\]](#)

[Example for Refining Data Using Extract Datasets \[Page 936\]](#)

Refining Data During Reading

Refining Data During Reading

The most direct method of refining data is to use the corresponding options of the SELECT statement (see [Reading Data from Database Tables \[Page 542\]](#)).

The example program below refines the data from tables SPFLI, SFLIGHT, and SBOOK as specified in the [Example for Refined Data \[Page 906\]](#).

```
REPORT SAPMZTST.

DATA: SUM TYPE I, CNT TYPE I.

TABLES: SPFLI, SFLIGHT, SBOOK.

SELECT * FROM SPFLI ORDER BY CITYFROM CITYTO CONNID.
SKIP.
WRITE: / SPFLI-CARRID,
       SPFLI-CONNID,
       'from', (15) SPFLI-CITYFROM,
       'to', (15) SPFLI-CITYTO.
ULINE.
SELECT * FROM SFLIGHT WHERE CARRID = SPFLI-CARRID
       AND CONNID = SPFLI-CONNID
       ORDER BY FLDATE.
SKIP.
WRITE: / 'Date:', SFLIGHT-FLDATE.
WRITE: 20 'Book-ID', 40 'Smoker', 50 'Class'.
ULINE.
SUM = 0.
CNT = 0.
SELECT * FROM SBOOK WHERE CARRID = SFLIGHT-CARRID
       AND CONNID = SFLIGHT-CONNID
       AND FLDATE = SFLIGHT-FLDATE
       ORDER BY CLASS SMOKER BOOKID.
WRITE: / SBOOK-BOOKID UNDER 'Book-ID',
       SBOOK-SMOKER UNDER 'Smoker',
       SBOOK-CLASS UNDER 'Class'.
SUM = SUM + SBOOK-LUGGWEIGHT.
CNT = CNT + 1.
ENDSELECT.
ULINE.
WRITE: 'Number of bookings: ', (3) CNT,
       / 'Total luggage weight:', (3) SUM, SBOOK-WUNIT.
ENDSELECT.
ENDSELECT.
```

This program uses the ORDER BY clause of the SELECT statement, which is described in [Specifying the Order of Lines \[Page 568\]](#). For other options of the SELECT statement that can be used for refining data, see [Selecting and Processing Data from Specific Columns \[Page 546\]](#).

With this method of refining data, you must program any database access yourself. In addition, you must program a selection screen, which offers to the user the possibility of restricting the set of data to be read (see [Working with Selection Screens \[Page 795\]](#)).

When reading data using logical databases, you need not program the database accesses yourself. The system automatically creates the selection screens. However, you have no influence on the sequence in which the data is read. The following topics describe how to sort such data records using temporary datasets.

Refining Data Using Internal Tables

When storing data in internal tables, you need one internal table for each database you read. This internal table contains some or all columns of the database table. It is up to you whether you create an internal table with a flat structure for each database table or if you create, for example, internal tables with nested structures. For tables with a flat structure, you must work with keys. In nested internal tables, you can store the desired data from the database tables according to their hierarchy in the logical database. You need no keys here.

For working with large amounts of data, storing the data in internal tables and refining them from there has the following disadvantages:

When dividing large amounts of data among several internal tables, the runtime required for processing the internal tables may be considerable, since the system processes all the tables one after the other. And the system stores internal table in an uncompressed form, which means that large amounts of data require much storage space. The system may even have to roll out the dataset into a paging area, thus slowing down the processing.

[Creating and Processing Internal Tables \[Page 260\]](#) describes in detail how to use internal tables.

The topics below contain two examples for refining data using internal tables:

[Refining Data Using Flat Internal Tables \[Page 912\]](#)

[Refining Data Using Nested Internal Tables \[Page 914\]](#)

Refining Data Using Flat Internal Tables

The following example shows how to refine data using flat internal tables.

```

The program is connected to the logical database F1S.
REPORT SAPMZTST.
DATA: SUM TYPE I, CNT TYPE I.
TABLES: SPFLI, SFLIGHT, SBOOK.
DATA: SORT_SPFLI LIKE SPFLI OCCURS 100 WITH HEADER LINE,
      SORT_SFLIGHT LIKE SFLIGHT OCCURS 100 WITH HEADER LINE,
      SORT_SBOOK LIKE SBOOK OCCURS 100 WITH HEADER LINE.
START-OF-SELECTION.
GET SPFLI.
APPEND SPFLI TO SORT_SPFLI.
GET SFLIGHT.
APPEND SFLIGHT TO SORT_SFLIGHT.
GET SBOOK.
APPEND SBOOK TO SORT_SBOOK.
END-OF-SELECTION.
SORT: SORT_SPFLI BY CITYFROM CITYTO CONNID,
      SORT_SFLIGHT BY FLDATE,
      SORT_SBOOK BY CLASS SMOKER BOOKID.
LOOP AT SORT_SPFLI.
  SKIP.
  WRITE: / SORT_SPFLI-CARRID,
         SORT_SPFLI-CONNID,
         'from', (15) SORT_SPFLI-CITYFROM,
         'to', (15) SORT_SPFLI-CITYTO.
  ULINE.
  LOOP AT SORT_SFLIGHT WHERE CARRID = SORT_SPFLI-CARRID
    AND CONNID = SORT_SPFLI-CONNID.
    SKIP.
    WRITE: / 'Date:', SORT_SFLIGHT-FLDATE.
    WRITE: 20 'Book-ID', 40 'Smoker', 50 'Class'.
    ULINE.
    SUM = 0.
    CNT = 0.
    LOOP AT SORT_SBOOK WHERE CARRID = SORT_SFLIGHT-CARRID
      AND CONNID = SORT_SFLIGHT-CONNID
      AND FLDATE = SORT_SFLIGHT-FLDATE.
      WRITE: / SORT_SBOOK-BOOKID UNDER 'Book-ID',
             SORT_SBOOK-SMOKER UNDER 'Smoker',
             SORT_SBOOK-CLASS UNDER 'Class'.
      SUM = SUM + SORT_SBOOK-LUGGWEIGHT.
      CNT = CNT + 1.
    ENDLOOP.
  ULINE.

```

Refining Data Using Flat Internal Tables

```
WRITE: 'Number of bookings: ', (3) CNT,  
      / 'Total luggage weight:',  
      (3) SUM, SORT_SBOOK-WUNIT.  
ENDLOOP.  
ENDLOOP.
```

This program creates the list displayed in the [Example for Refined Data \[Page 906\]](#).

The GET events that retrieve the data are clearly separated from the sorting process. The structure and the contents of the database tables are taken over completely by three internal tables. After the sorting process using the SORT statement (see [Sorting Internal Tables \[Page 319\]](#)), the system displays the contents of the tables. The loop structure corresponds exactly to the structure of the SELECT loops in the example in [Refining Data During Reading \[Page 909\]](#).

Refining Data Using Nested Internal Tables

The example below shows how to sort data using nested internal tables.

```
The program is connected to the logical database F1S.
REPORT SAPMZTST.
DATA: SUM TYPE I, CNT TYPE I.
TABLES: SPFLI, SFLIGHT, SBOOK.
DATA: BEGIN OF SORT_SBOOK,
      BOOKID LIKE SBOOK-BOOKID,
      SMOKER LIKE SBOOK-SMOKER,
      CLASS LIKE SBOOK-CLASS,
      LUGGWEIGHT LIKE SBOOK-LUGGWEIGHT,
      WUNIT LIKE SBOOK-WUNIT,
      END OF SORT_SBOOK.
DATA: BEGIN OF SORT_SFLIGHT,
      FLDATE LIKE SFLIGHT-FLDATE,
      SBOOK LIKE SORT_SBOOK OCCURS 100,
      END OF SORT_SFLIGHT.
DATA: BEGIN OF SORT_SPFLI OCCURS 100,
      CARRID LIKE SPFLI-CARRID,
      CONNID LIKE SPFLI-CONNID,
      CITYFROM LIKE SPFLI-CITYFROM,
      CITYTO LIKE SPFLI-CITYTO,
      SFLIGHT LIKE SORT_SFLIGHT OCCURS 100,
      END OF SORT_SPFLI.
START-OF-SELECTION.
GET SPFLI.
REFRESH SORT_SPFLI-SFLIGHT.
GET SFLIGHT.
REFRESH SORT_SFLIGHT-SBOOK.
GET SBOOK.
MOVE-CORRESPONDING SBOOK TO SORT_SBOOK.
APPEND SORT_SBOOK TO SORT_SFLIGHT-SBOOK.
GET SFLIGHT LATE.
MOVE-CORRESPONDING SFLIGHT TO SORT_SFLIGHT.
APPEND SORT_SFLIGHT TO SORT_SPFLI-SFLIGHT.
GET SPFLI LATE.
MOVE-CORRESPONDING SPFLI TO SORT_SPFLI.
APPEND SORT_SPFLI.
END-OF-SELECTION.
SORT SORT_SPFLI BY CITYFROM CITYTO CONNID.
LOOP AT SORT_SPFLI.
  SKIP.
  WRITE: / SORT_SPFLI-CARRID,
```

Refining Data Using Nested Internal Tables

```
      SORT SPFLI-CONNID,  
        'from', (15) SORT SPFLI-CITYFROM,  
        'to', (15) SORT SPFLI-CITYTO.  
    ULINE.  
    SORT SORT_SPFLI-SFLIGHT BY FLDATE.  
    LOOP AT SORT_SPFLI-SFLIGHT INTO SORT_SFLIGHT.  
      SKIP.  
      WRITE: / 'Date:', SORT_SFLIGHT-FLDATE.  
      WRITE: 20 'Book-ID', 40 'Smoker', 50 'Class'.  
      ULINE.  
      SORT SORT_SFLIGHT-SBOOK BY CLASS SMOKER BOOKID.  
      SUM = 0.  
      CNT = 0.  
      LOOP AT SORT_SFLIGHT-SBOOK INTO SORT_SBOOK.  
        WRITE: / SORT_SBOOK-BOOKID UNDER 'Book-ID',  
              SORT_SBOOK-SMOKER UNDER 'Smoker',  
              SORT_SBOOK-CLASS UNDER 'Class'.  
        SUM = SUM + SORT_SBOOK-LUGGWEIGHT.  
        CNT = CNT + 1.  
      ENDLOOP.  
      ULINE.  
      WRITE: 'Number of bookings: ', (3) CNT,  
            / 'Total luggage weight:',  
            (3) SUM, SORT_SBOOK-WUNIT.  
    ENDLOOP.  
  ENDLOOP.
```

This program creates the list displayed in the [Example for Refined Data \[Page 906\]](#).

During the GET events, the system reads the data into the three-level table SORT_SPFLI which contains the substructure SFLIGHT and the sub-substructure SBOOK. The sorting process takes place on the individual nesting levels.

This way of programming does not require key relations between the internal tables (no WHERE conditions), but it is more complex than with flat internal tables. And the increased internal maintenance effort has a negative effect on the storage space required as well as on the runtime behavior.

Refining Data Using Extract Datasets

Since internal tables have fixed line structures, they are not suited to handle data sets with varying structures. For this purpose, ABAP offers the possibility to create so-called extract datasets. An extract is a sequential dataset in the storage area of the program. This means that you can access its data only within a loop. You cannot access its individual lines via an index as for internal tables. During the report's runtime, the system can create exactly one extract dataset. As for internal tables, the size of the extract data set is principally unlimited, since the system rolls it out if necessary.

An extract dataset consists of a sequence of records of a pre-defined structure. However, the structure need not be identical for all records. In one extract dataset, you can store records of different length and structure one after the other. You need not create an individual dataset for each different structure you want to store. This fact reduces the maintenance effort considerably.

In contrast to internal tables, the system partly compresses extract datasets when storing them. This reduces the storage space required. In addition, you need not specify the structure of an extract dataset at the beginning of the program, but you can determine it dynamically during the flow of the program.

The following topics explain about

[Creating and Filling Extract Datasets \[Page 917\]](#)

[Processing Extract Datasets \[Page 922\]](#)

For an example showing how to use extract datasets to refine data, see

[Example for Refining Data Using Extract Datasets \[Page 936\]](#)

Creating and Filling Extract Datasets

To create and fill an extract dataset in your program, carry out the following three steps:

1. Define the record types you want to use in the extract dataset as field groups.

[Defining Extract Records as Field Groups \[Page 918\]](#)

2. Define the structure of each record type by assigning fields.

[Assigning Fields to a Field Group \[Page 919\]](#)

3. Extract the desired data into the dataset.

[Extracting a Dataset \[Page 920\]](#)

Defining Extract Records as Field Groups

An extract dataset consists of a sequence of records. These records may have different structures. All records with the same structure form a record type. You must define each record type of an extract dataset as a field group, using the FIELD-GROUPS statement.

Syntax

FIELD-GROUPS <fg>.

This statement defines a field group <fg>. A field group combines several fields under one name. See [Assigning Fields to a Field Group \[Page 919\]](#) to find out how to define the individual fields of a field group. For reasons of readability, define field groups at the beginning of the program, following the declaration section.

A field group does not reserve storage space for the fields, but contains pointers to existing fields. When filling the extract dataset with records, these pointers determine the contents of the stored records.

You can define a special field group HEADER:

Syntax

FIELD-GROUPS HEADER.

When filling the extract dataset, the system automatically prefixes any other field groups with this field group. This means, in the extract dataset, a record of the field group <fg> always contains the fields of the HEADER field group first. When sorting the extract dataset, the system uses these fields as the default sort key.

```
FIELD-GROUPS: HEADER, FLIGHT_INFO, FLIGHT_DATE.
```

This statement chain creates three field groups.

Assigning Fields to a Field Group

Assigning Fields to a Field Group

To determine which fields to include into a field group, use the INSERT statement:

Syntax

```
INSERT <f1>... <fn> INTO <fg>.
```

This statement defines the fields of field group <fg>. Before you can assign fields to a field group, you must define the field group <fg> using the FIELD-GROUPS statement. As fields <f_i>, you can use global data objects only. You cannot assign a local data object defined in a subroutine or function module to a field group.

The INSERT statement, just as the FIELD-GROUPS statement, neither reserves storage space nor transfers values. You use the INSERT statement to create pointers to the fields <f_i> in the field group <fg>, thus defining the structures of the extract records. To fill the extract records, see [Extracting a Dataset \[Page 920\]](#).

While executing the program, you can assign fields to a field group up to the point when you use this field group for the first time to fill an extract record. From this point on, the structure of the record is fixed and may no longer be changed. In short, as long as you have not used a field group yet, you can dynamically make any assignments to it.

Since the special field group HEADER is part of **each** extract record, you can no longer change this field group after filling the first extract record.

A field may occur in several field groups; however, this means unnecessary data redundancy within the extract dataset.

You must not necessarily fill a field group with fields. If you define the HEADER field group, the extract records are filled with the fields from HEADER only.

```
TABLES: SPFLI, SFLIGHT.
```

```
FIELD-GROUPS: HEADER, FLIGHT_INFO, FLIGHT_DATE.
```

```
INSERT: SPFLI-CARRID SPFLI-CONNID SFLIGHT-FLDATE  
        INTO HEADER,  
        SPFLI-CITYFROM SPFLI-CITYTO  
        INTO FLIGHT_INFO.
```

This example creates three field groups. The INSERT statement fills two of the field groups with fields.

Extracting a Dataset

To create the actual extract dataset, use the `EXTRACT` statement:

Syntax

`EXTRACT <fg>.`

With the first `EXTRACT` statement of a program, the system creates the extract dataset and adds the first extract record. With each subsequent `EXTRACT` statement, the system adds another extract record to the extract dataset.

Each extract record contains, if specified, the fields of the field group `HEADER` as sort key, followed by exactly those fields that are contained in the field group `<fg>`. During the extraction process, the system fills the extract record with the current contents of the corresponding fields.

As soon as the system has processed the first `EXTRACT` statement for a field group `<fg>`, the structure of the corresponding extract record in the extract dataset is fixed. You can no longer insert new fields into the field groups `<fg>` and `HEADER`. If you try to modify one of the field groups afterwards and use it in another `EXTRACT` statement, a runtime error occurs.

By processing `EXTRACT` statements several times using different field groups, you fill the extract dataset with records of different length and structure. Since you can modify field groups dynamically up to their first usage in an `EXTRACT` statement, extract datasets provide the advantage that you need not determine the structure at the beginning of the program.

The following program is connected to the logical database `F1S`.

```
REPORT SAPMZTST.

TABLES: SPFLI, SFLIGHT.

FIELD-GROUPS: HEADER, FLIGHT_INFO, FLIGHT_DATE.

INSERT: SPFLI-CARRID SPFLI-CONNID SFLIGHT-FLDATE
        INTO HEADER,
        SPFLI-CITYFROM SPFLI-CITYTO
        INTO FLIGHT_INFO.

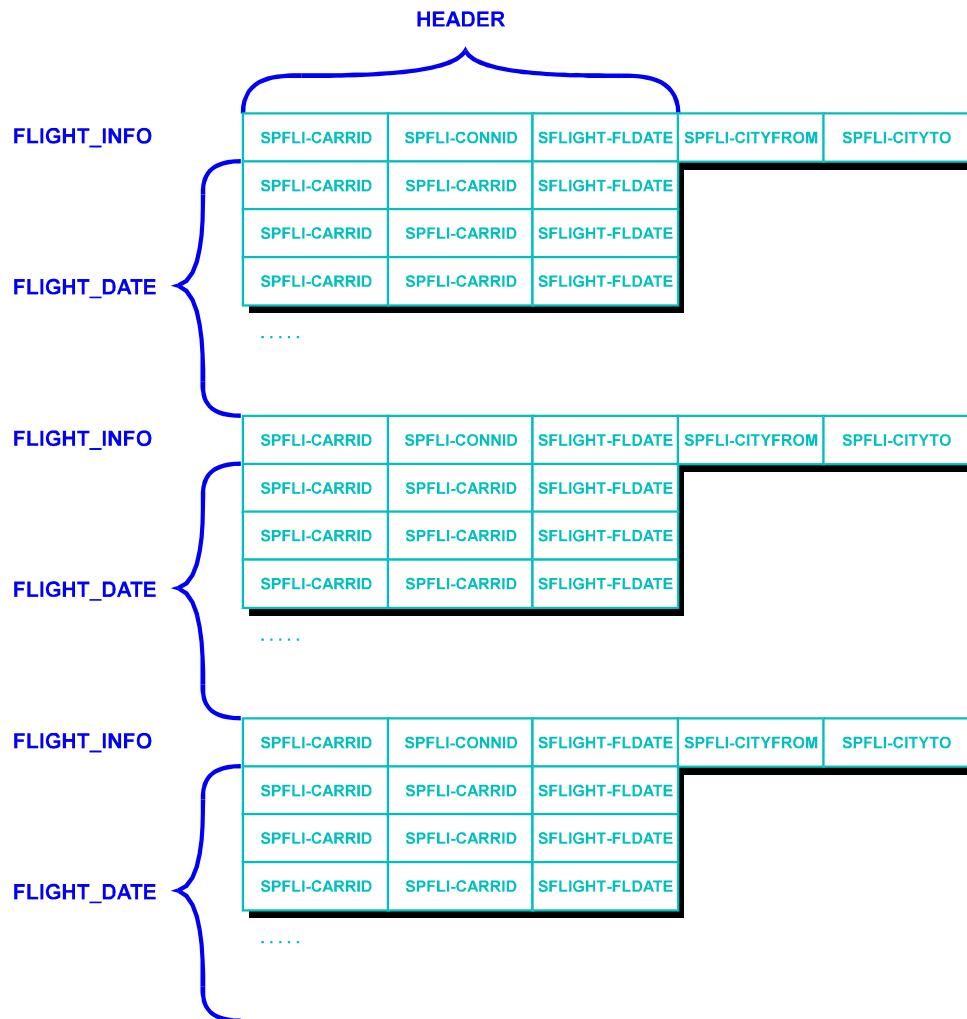
START-OF-SELECTION.

GET SPFLI.
EXTRACT FLIGHT_INFO.

GET SFLIGHT.
EXTRACT FLIGHT_DATE.
```

This example creates three field groups. The `INSERT` statement assigns fields to two of the field groups. During the `GET` events, the system fills the extract dataset with two different record types. The records of the field group `FLIGHT_INFO` consist of five fields: `SPFLI-CARRID`, `SPFLI-CONNID`, `SFLIGHT-FLDATE`, `SPFLI-CITYFROM`, and `SPFLI-CITYTO`. The first three fields belong to the prefixed field group `HEADER`. The records of the field group `FLIGHT_DATE` consist only of the three fields of field group `HEADER`. The following figure shows the structure of the extract dataset:

Extracting a Dataset



Processing Extract Datasets

The following topics describe how to process filled extract datasets.

You must fill the extract dataset of a program with all data desired before you start to process it. After the first processing operation, you can no longer extract any records into the dataset.

How to Process Extract Datasets

[Reading Extract Datasets \[Page 930\]](#)

[Sorting Extract Datasets \[Page 933\]](#)

[Processing Control Levels \[Page 923\]](#)

[Calculating Numbers and Totals \[Page 927\]](#)

Processing Control Levels

Processing Control Levels

By using the SORT statement to sort the extract dataset, you define a control level structure. The control level structure of an extract dataset corresponds to the sequence of the fields in the HEADER field group. After sorting, you can use the AT statement within a LOOP-ENDLOOP loop to program statement blocks that the system processes only at a control break, that is, when the control level changes.

Syntax

```
AT NEW <f> | AT END OF <f>.
```

```
...
```

```
ENDAT.
```

A control break occurs and the system processes the statement block within AT-ENDAT, if the field <f> or a higher-level field of the sort key in the current extract record contains a value different from that in the previous record (for AT NEW) or in the subsequent record (for AT END) of the extract dataset. Field <f> must be part of the HEADER field group.

If the extract dataset is not sorted, the system ignores the statement block within AT-ENDAT.

When determining control breaks, the system ignores any field contents of <f> that were undefined during the extraction process (nulls).

The system processes all AT...ENDAT processing blocks of a loop one after the other. Therefore, beware of their sequence. Within the control level logic, keep to the sorting sequence. This sequence must not necessarily be the sequence of the fields in the HEADER field group, but can also be the one determined in the SORT statement.

If you have sorted an extract dataset by the fields <f1>, <f2>, ..., the processing of the control levels should be written between the other control statements as follows:

```
LOOP.
```

```
AT FIRST.... ENDAT.
```

```
AT NEW <f1>..... ENDAT.
```

```
AT NEW <f2>..... ENDAT.
```

```
...
```

```
AT <fi>..... ENDAT.
```

```
<single line processing without control statement>
```

```
...
```

```
AT END OF <f2>.... ENDAT.
```

```
AT END OF <f1>.... ENDAT.
```

```
AT LAST..... ENDAT.
```

```
ENDLOOP.
```

You need not to use all of the statement blocks shown here, but only those you need in your program.

```
REPORT SAPMZTST.
```

```
DATA: T1(4), T2 TYPE I.
```

```
FIELD-GROUPS: HEADER, TEST.
```

```
INSERT T2 T1 INTO HEADER.
```

```
T1 ='AABB'. T2 = 1. EXTRACT TEST.  
T1 ='BBCC'. T2 = 2. EXTRACT TEST.  
T1 ='AAAA'. T2 = 2. EXTRACT TEST.  
T1 ='AABB'. T2 = 1. EXTRACT TEST.  
T1 ='BBBB'. T2 = 2. EXTRACT TEST.  
T1 ='BBCC'. T2 = 2. EXTRACT TEST.  
T1 ='AAAA'. T2 = 1. EXTRACT TEST.  
T1 ='BBBB'. T2 = 1. EXTRACT TEST.  
T1 ='AAAA'. T2 = 3. EXTRACT TEST.  
T1 ='AABB'. T2 = 1. EXTRACT TEST.
```

```
SORT BY T1 T2.
```

```
LOOP.
```

```
AT FIRST.
```

```
WRITE 'Start of LOOP'.
```

```
ULINE.
```

```
ENDAT.
```

```
AT NEW T1.
```

```
WRITE / ' New T1:'.
```

```
ENDAT.
```

```
AT NEW T2.
```

```
WRITE / ' New T2:'.
```

```
ENDAT.
```

```
WRITE: /14 T1, T2.
```

```
AT END OF T2.
```

```
WRITE / 'End of T2'.
```

```
ENDAT.
```

```
AT END OF T1.
```

```
WRITE / 'End of T1'.
```

```
ENDAT.
```

```
AT LAST.
```

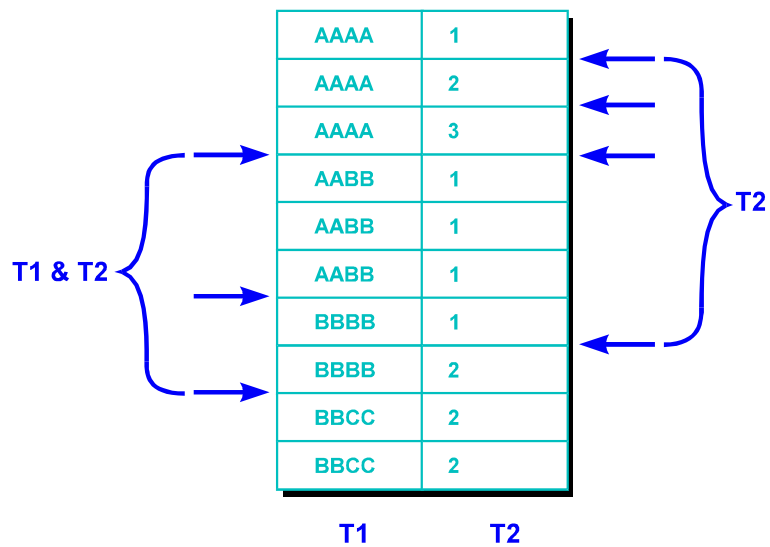
```
ULINE.
```

```
ENDAT.
```

```
ENDLOOP.
```

This program creates a sample extract that contains the fields of the HEADER field group only. After the sorting process, the extract dataset has several control breaks for the control levels T1 and T2, which are indicated in the following figure:

Processing Control Levels



In the LOOP-ENDLOOP loop, the system outputs the contents of the dataset and the encountered control breaks as follows:

Start of LOOP			
New T1:			
New T2:			
	AAAA	1	
End of T2			
New T2:			
	AAAA	2	
End of T2			
New T2:			
	AAAA	3	
End of T2			
End of T1			
New T1:			
New T2:			
	AABB	1	
	AABB	1	
	AABB	1	
End of T2			
End of T1			
New T1:			
New T2:			
	BBBB	1	
End of T2			
New T2:			
	BBBB	2	
End of T2			
End of T1			
New T1:			
New T2:			
	BBCC	2	
	BBCC	2	
End of T2			
End of T1			

Calculating Numbers and Totals

Calculating Numbers and Totals

When reading a sorted extract dataset using LOOP-ENDLOOP, you can access two automatically generated fields CNT(<f>) and SUM(<g>) that provide you with the numbers of different values or with totals on numeric fields. The system fills these fields at the end of a control level (before a control break, see [Processing Control Levels \[Page 923\]](#)) and after reading the last record of the dataset as follows:

- CNT(<f>)
If <f> is a non-numeric field of the HEADER field group and the system sorted the extract dataset by <f>, CNT(<f>) contains the number of different values <f> assumed within the control level or within the entire dataset, respectively.
- SUM(<g>)
If <g> is a numeric field of the extract dataset, SUM (<g>) contains the total of the values of <g> within the control level or the entire dataset, respectively.

You can access these fields either within the processing blocks following AT END OF, before a control break occurs (see [Processing Control Levels \[Page 923\]](#)), or in the processing block following AT LAST, after reading the entire dataset (see [Reading Extract Datasets \[Page 930\]](#)).

Access the CNT(<f>) and SUM(<g>) fields only after sorting the dataset.
Otherwise, a runtime error can occur.

```
REPORT SAPMZTST.
DATA: T1(4), T2 TYPE I.
FIELD-GROUPS: HEADER, TEST.
INSERT T2 T1 INTO HEADER.

T1 ='AABB'. T2 = 1. EXTRACT TEST.
T1 ='BBCC'. T2 = 2. EXTRACT TEST.
T1 ='AAAA'. T2 = 2. EXTRACT TEST.
T1 ='AABB'. T2 = 1. EXTRACT TEST.
T1 ='BBBB'. T2 = 2. EXTRACT TEST.
T1 ='BBCC'. T2 = 2. EXTRACT TEST.
T1 ='AAAA'. T2 = 1. EXTRACT TEST.
T1 ='BBBB'. T2 = 1. EXTRACT TEST.
T1 ='AAAA'. T2 = 3. EXTRACT TEST.
T1 ='AABB'. T2 = 1. EXTRACT TEST.

SORT BY T1 T2.

LOOP.

WRITE: /20 T1, T2.

AT END OF T2.
  ULINE.
  WRITE: 'Sum:', 20 SUM(T2).
  ULINE.
ENDAT.
```

```

AT END OF T1.
  WRITE: 'Different values:', (6) CNT(T1).
  ULINE.
  ENDAT.

AT LAST.
  ULINE.
  WRITE: 'Sum:', 20 SUM(T2),
        / 'Different values:', (6) CNT(T1).
  ENDAT.

ENDLOOP.

```

This program creates a sample extract, containing the fields of the HEADER field group only. After sorting, the system outputs the contents of the dataset, the number of the different T1 fields, and the totals of the T2 fields at the end of each control level and at the end of the loop:

Calculating Numbers and Totals

AAAA	1
Sum:	1
AAAA	2
Sum:	2
AAAA	3
Sum:	3
Different values:	1
AABB	1
AABB	1
AABB	1
Sum:	3
Different values:	1
BBBB	1
Sum:	1
BBBB	2
Sum:	2
Different values:	1
BBCC	2
BBCC	2
Sum:	4
Different values:	1
Sum:	16
Different values:	4

Reading Extract Datasets

As you do with internal tables, you can also read the data of an extract dataset using a loop:

Syntax

```
LOOP.  
...  
[AT FIRST | AT <fgi> [WITH <fgj>] | AT LAST.  
...  
ENDAT.]  
...  
ENDLOOP.
```

The statements LOOP-ENDLOOP terminate the creation of the extract dataset of a program and execute a loop on all records of the dataset. During each loop pass, the system reads one extract record and places the values of the extracted data directly back into the source fields. You can execute several loops in succession.

In contrast to internal tables, for extract datasets you need no special work area as an interface (see [Choosing a Table Type \[Page 267\]](#)). For each record read, you can process the read data in the loop's statement block using their original field names.

In contrast to the LOOP AT-ENDLOOP loop on internal tables, you cannot nest the LOOP-ENDLOOP loop on extracts, otherwise producing a runtime error.

In the statement block of the loop as well as after processing the loop, no further EXTRACT statements are allowed. They would produce a runtime error.

Loop Control

If you want to execute some statements for certain records of the dataset only, use the control statements AT and ENDAT.

The system processes the statement blocks between the control statements for the different options of AT as follows:

- AT FIRST

The system executes the statement block once for the first record of the dataset.

- AT <fg_i> [WITH <fg_j>]

The system processes the statement block, if the record type of the currently read extract record was defined using the field group <fg_i>. When using the WITH <fg_j> option, in the extract dataset, the currently read record of field group <fg_i> must be immediately followed by a record of field group <fg_j>.

- AT LAST

The system executes the statement block once for the last record of the dataset.

For information on how to use the control statements AT and ENDAT for processing control levels, see [Processing Control Levels \[Page 923\]](#).

Reading Extract Datasets

For information on how to terminate a loop before it is actually finished, see [Terminating Loops \[Page 256\]](#).

The program below is connected to the logical database F1S.

```
REPORT SAPMZTST.
TABLES: SPFLI, SFLIGHT.
FIELD-GROUPS: HEADER, FLIGHT_INFO, FLIGHT_DATE.
INSERT: SPFLI-CARRID SPFLI-CONNID SFLIGHT-FLDATE
        INTO HEADER,
        SPFLI-CITYFROM SPFLI-CITYTO
        INTO FLIGHT_INFO.

START-OF-SELECTION.

GET SPFLI.
EXTRACT FLIGHT_INFO.

GET SFLIGHT.
EXTRACT FLIGHT_DATE.

END-OF-SELECTION.

LOOP.
  AT FIRST.
    WRITE / 'Start of LOOP'.
    ULINE.
  ENDAT.
  AT FLIGHT_INFO WITH FLIGHT_DATE.
    WRITE: / 'Info:',
            SPFLI-CARRID, SPFLI-CONNID, SFLIGHT-FLDATE,
            SPFLI-CITYFROM, SPFLI-CITYTO.
  ENDAT.
  AT FLIGHT_DATE.
    WRITE: / 'Date:',
            SPFLI-CARRID, SPFLI-CONNID, SFLIGHT-FLDATE.
  ENDAT.
  AT LAST.
    ULINE.
    WRITE / 'End of LOOP'.
  ENDAT.
ENDLOOP.
```

Creating and filling the extract dataset corresponds to the examples in [Creating and Filling Extract Datasets \[Page 917\]](#). At the END-OF-SELECTION event, data retrieval is over. The system reads the dataset once in the LOOP-ENDLOOP loop.

The control statements AT FIRST and AT LAST instruct the system to write one line and one underscore line in the list, once at the beginning of the loop and once at the loop's end.

The control statement AT <fg_i> tells the system to output the fields corresponding to each of the two record types. Due to the WITH FLIGHT_DATE option, the system outputs the records of field group FLIGHT_INFO only, if at least one

record of field group FLIGHT_DATE follows; that is, if the logical database passed at least one date for a flight.

The beginning of the output list looks like this:

```

Start of LOOP

Info: AA 0017 #####/##/## ATLANTA SAN FRANCISCO
Date: AA 0017 1996/01/09
Date: AA 0017 1996/01/10
Date: AA 0017 1996/01/11
Date: AA 0017 1996/01/12
Date: AA 0017 1996/01/13
Date: AA 0017 1996/01/14
Date: AA 0017 1996/01/15
Date: AA 0017 1996/01/16
Info: AA 0026 #####/##/## FRANKFURT NEW YORK
Date: AA 0026 1996/01/09
Date: AA 0026 1996/01/10
Date: AA 0026 1996/01/11
Date: AA 0026 1996/01/12
Date: AA 0026 1996/01/13
Date: AA 0026 1996/01/14
Date: AA 0026 1996/01/15
Date: AA 0026 1996/01/16
Info: AA 0064 #####/##/## SAN FRANCISCO NEW YORK
Date: AA 0064 1996/01/09
Date: AA 0064 1996/01/10
Date: AA 0064 1996/01/11
Date: AA 0064 1996/01/12
Date: AA 0064 1996/01/13
Date: AA 0064 1996/01/14
Date: AA 0064 1996/01/15
Date: AA 0064 1996/01/16
Date: AA 0064 1996/08/13
Info: DL 1699 #####/##/## NEW YORK SAN FRANCISCO
Date: DL 1699 1996/01/09
Date: DL 1699 1996/01/10

```

That the contents of field SFLIGHT-FLDATE in the HEADER area of the FLIGHT_INFO record type is not defined is due to the fact that the logical database at the end of a hierarchy level fills all fields of this level with nulls. This feature is important for sorting and for processing control levels of extract datasets.

Sorting Extract Datasets

Sorting Extract Datasets

As you do with internal tables, you can also sort extract datasets using the SORT statement:

Syntax

```
SORT [<order>][AS TEXT]
  [BY <f1> [<order>][AS TEXT]... <fn> [<order>][AS TEXT]].
```

The SORT statement terminates the creation of the extract dataset of a program and, at the same time, sorts its records. Without the BY option, the system sorts the dataset by the key specified in the HEADER field group.

To define a different sort key, use the BY option. The system then sorts the dataset according to the specified components <f₁>... <f_n>. These components must either be fields of the HEADER field group or field groups containing nothing but fields from the HEADER field group. The number of key fields is limited to 50. If you specify more than one key field, the system sorts the records first by <f₁>, then by <f₂>, and so on. The system uses the options you specify before BY as a default for all fields specified behind BY. The options that you specify behind individual fields overwrite for these fields the options specified before BY.

You can specify the sorting sequence by using DESCENDING or ASCENDING in the <order> option. The default is ascending. For text fields, use the AS TEXT option to specify the sorting method. This option, as for sorting internal tables, causes alphabetical sorting (see [Sorting Internal Tables \[Page 319\]](#)). If you want to sort an extract dataset alphabetically more than once, you should include an alphabetic-sort field into the sort key instead of the text field for performance reasons. To fill this field, use the CONVERT statement (see [Converting into a Sortable Format \[Page 207\]](#)).

If you prefix BY with AS TEXT, the option only applies to type C fields of the sort key. If you place AS TEXT behind a field, the field must be of type C. If you place AS TEXT behind a field group, the option only applies to type C fields of this group.

This sorting process is not stable, that is, the old sequence of records with the same sort key must not necessarily be kept.

If the main storage space available is not enough for sorting the data, the system writes data to an external help file during the sorting process. This help file is determined by the SAP profile parameter DIR_SORTTMP.

The SORT statement treats all fields of the sort key that are undefined during the extraction as nulls. The system sorts nulls always in front of the fields with defined values, regardless of the specified sorting sequence.

After processing the SORT statement, no further EXTRACT statements are allowed. They would produce a runtime error.

In your program, you can sort an extract dataset as often as you want, using different keys. The only prerequisite is, that all fields by which you want to sort, are contained in the HEADER during the extraction process. You must not use the SORT statement within a LOOP-ENDLOOP loop. However, you can sort and read the extract dataset one after the other in any sequence.

Each sorting process executed on the extract dataset using the SORT statement defines a control level structure and is required for subsequent processing of control levels (see [Processing Control Levels \[Page 923\]](#)).

The program below is connected to the logical database F1S.

```
REPORT SAPMZTST.

TABLES: SPFLI, SFLIGHT.

FIELD-GROUPS: HEADER, FLIGHT_INFO, FLIGHT_DATE.

INSERT: SPFLI-CARRID SPFLI-CONNID SFLIGHT-FLDATE
       INTO HEADER,
       SPFLI-CITYFROM SPFLI-CITYTO
       INTO FLIGHT_INFO.

START-OF-SELECTION.

GET SPFLI.
EXTRACT FLIGHT_INFO.

GET SFLIGHT.
EXTRACT FLIGHT_DATE.

END-OF-SELECTION.

SORT DESCENDING.

LOOP.
  AT FIRST.
    WRITE / 'Start of LOOP'.
    ULINE.
  ENDAT.
  AT FLIGHT_INFO WITH FLIGHT_DATE.
    WRITE: / 'Info:',
           SPFLI-CARRID, SPFLI-CONNID, SFLIGHT-FLDATE,
           SPFLI-CITYFROM, SPFLI-CITYTO.
  ENDAT.
  AT FLIGHT_DATE.
    WRITE: / 'Date:',
           SPFLI-CARRID, SPFLI-CONNID, SFLIGHT-FLDATE.
  ENDAT.
  AT LAST.
    ULINE.
    WRITE / 'End of LOOP'.
  ENDAT.
ENDLOOP.
```

Except the SORT DESCENDING statement, this example is identical with the example in [Reading Extract Datasets \[Page 930\]](#). The SORT statement tells the system to sort the extract dataset in descending order by the three fields of the HEADER field group, before reading it using the LOOP-ENDLOOP loop. The end of the list looks as follows:

Sorting Extract Datasets

```

DL 1699 1996/01/10
Date: DL 1699 1996/01/09
Info: AA 0064 #####/##/## SAN FRANCISCO NEW YORK
Date: AA 0064 1996/08/13
Date: AA 0064 1996/01/16
Date: AA 0064 1996/01/15
Date: AA 0064 1996/01/14
Date: AA 0064 1996/01/13
Date: AA 0064 1996/01/12
Date: AA 0064 1996/01/11
Date: AA 0064 1996/01/10
Date: AA 0064 1996/01/09
Info: AA 0026 #####/##/## FRANKFURT NEW YORK
Date: AA 0026 1996/01/16
Date: AA 0026 1996/01/15
Date: AA 0026 1996/01/14
Date: AA 0026 1996/01/13
Date: AA 0026 1996/01/12
Date: AA 0026 1996/01/11
Date: AA 0026 1996/01/10
Date: AA 0026 1996/01/09
Info: AA 0017 #####/##/## ATLANTA SAN FRANCISCO
Date: AA 0017 1996/01/16
Date: AA 0017 1996/01/15
Date: AA 0017 1996/01/14
Date: AA 0017 1996/01/13
Date: AA 0017 1996/01/12
Date: AA 0017 1996/01/11
Date: AA 0017 1996/01/10
Date: AA 0017 1996/01/09

End of LOOP

```

Note that the system places the records containing undefined characters in the SFLIGHT-FLDATE field in front of the other records during the sorting process. This is done to preserve the hierarchy of the data from the logical database, independent of the sorting sequence.

Example for Refining Data Using Extract Datasets

The following example refines the data of the tables SPFLI, SFLIGHT, and SBOOK as described in the [Example for Refined Data \[Page 906\]](#).

```

The program below is connected to the logical database F1S.
REPORT SAPMZTST.
TABLES: SPFLI, SFLIGHT, SBOOK.
FIELD-GROUPS: HEADER, FLIGHT_INFO, FLIGHT_BOOKING.
INSERT:
  SPFLI-CITYFROM SPFLI-CITYTO
  SPFLI-CONNID SFLIGHT-FLDATE
  SBOOK-CLASS SBOOK-SMOKER SBOOK-BOOKID INTO HEADER,
  SPFLI-CARRID          INTO FLIGHT_INFO,
  SBOOK-LUGGWEIGHT SBOOK-WUNIT  INTO FLIGHT_BOOKING.
START-OF-SELECTION.
GET SPFLI.
EXTRACT FLIGHT_INFO.
GET SFLIGHT.
GET SBOOK.
EXTRACT FLIGHT_BOOKING.
END-OF-SELECTION.
SORT.
LOOP.
  AT FLIGHT_INFO.
    SKIP.
    WRITE: / SPFLI-CARRID,
           SPFLI-CONNID,
           'from', (15) SPFLI-CITYFROM,
           'to', (15) SPFLI-CITYTO.
    ULINE.
  ENDAT.
  AT NEW SFLIGHT-FLDATE.
    SKIP.
    WRITE: / 'Date:', SFLIGHT-FLDATE.
    WRITE: 20 'Book-ID', 40 'Smoker', 50 'Class'.
    ULINE.
  ENDAT.
  AT FLIGHT_BOOKING.
    WRITE: / SBOOK-BOOKID UNDER 'Book-ID',
           SBOOK-SMOKER UNDER 'Smoker',
           SBOOK-CLASS UNDER 'Class'.
  ENDAT.
AT END OF SFLIGHT-FLDATE.
  ULINE.
  WRITE: 'Number of bookings: ', (3) CNT(SBOOK-BOOKID),
```

Example for Refining Data Using Extract Datasets

```
      / 'Total luggage weight:',  
        SUM(SBOOK-LUGGWEIGHT), SBOOK-WUNIT.  
    ENDAT.  
  ENDLOOP.
```

The system creates three field groups and fills them with several fields. During the GET events, it fills the extract dataset due to the EXTRACT statements. Note that for GET SFLIGHT no EXTRACT statement occurs, since the desired field SFLIGHT_FLDATE is part of the HEADER field group and thus automatically extracted for each subordinate event GET SBOOK.

After retrieving the data, the system terminates the creation of the dataset due to the SORT statement and sorts the dataset by the HEADER sort key. In the LOOP-ENDLOOP loop, it writes the sorted extract dataset to the output list, thereby executing several AT-ENDAT blocks and using the fields CNT(...) and SUM(...).

Creating Complex Lists

Lists are the output medium for structured, formatted data from ABAP programs. Each program can produce up to 21 lists: one basic list and 20 secondary lists. The basic list is the standard screen of an executable program (report). You can display the basic list in a transaction using the LEAVE TO LIST-PROCESSING statement. This section deals with creating lists in general. That means, most of the statements described here apply to basic as well as to secondary lists. By default, the system transfers the output of a program to the basic list. In most cases, the basic list is the only list of a program. For this reason, the examples in this section mainly deal with the basic list. For information on how to program secondary lists, see [Interactive Lists \[Page 1030\]](#).

From within your ABAP program, you can either output a list on the screen or send it to the SAP spool system. By default, the list is displayed on the screen. All examples in this section use the default. For information on how to print lists, see [Printing Lists \[Page 1126\]](#).

The basic ABAP statement for writing data to lists is the WRITE statement. Other output statements are ULINE and SKIP. For details concerning these three statements, see [Creating Simple Lists with the WRITE Statement \[Page 890\]](#).

The following topics describe the structure of a list and the options you have to layout a list when creating it:

[The Standard List \[Page 939\]](#)

[The Self-Defined List \[Page 953\]](#)

[Lists with Several Pages \[Page 962\]](#)

[Laying Out List Pages \[Page 982\]](#)

The Standard List

The Standard List

If your ABAP program makes use only of the WRITE, SKIP, and ULINE output statements and does not contain any of the editing statements described later on in this section, the system transfers the output to a standard list. In an executable program, the standard list appears on the screen automatically after the data selection is completed.

The topics below describe

[Example for a Standard List \[Page 940\]](#)

[Structure of the Standard List \[Page 942\]](#)

[User Interface of the Standard List \[Page 946\]](#)

Example for a Standard List

The output screen below shows a standard list:

ID	Departure from Departure Time	Arrival at Arrival Time	Time of Flight
0017	NEW YORK 13:30:00	SAN FRANCISCO 16:31:00	06:01:00
0026	FRANKFURT 08:30:00	NEW YORK 09:50:00	08:20:00
0064	SAN FRANCISCO 09:00:00	NEW YORK 17:21:00	05:21:00
0400	FRANKFURT 10:10:00	NEW YORK 11:34:00	08:24:00

SAP *** SAP *** SAP *** SAP *** SAP *** SAP
Flight Information System
International Connections

To create this standard list, use the following sample program.

```

REPORT SAPMZTST.

TABLES SPFLI.

SKIP.
ULINE AT /(62).

SELECT * FROM SPFLI WHERE CONNID GE 0017
      AND CONNID LE 0400.
WRITE: / SY-VLINE, SPFLI-CONNID, SY-VLINE,
      (15) SPFLI-CITYFROM, 26 SY-VLINE,
      31 SPFLI-CITYTO, 51 SY-VLINE, 62 SY-VLINE,
      / SY-VLINE, 8 SY-VLINE,
      SPFLI-DEPTIME UNDER SPFLI-CITYFROM, 26 SY-VLINE,
      SPFLI-ARRTIME UNDER SPFLI-CITYTO, 51 SY-VLINE,
      SPFLI-FLTIME, SY-VLINE.
ULINE AT /(62).
ENDSELECT.

WRITE: /10 'SAP *** SAP *** SAP *** SAP *** SAP *** SAP',
      /19(43) 'Flight Information System',
      /19(43) 'International Connections'.

```

Example for a Standard List

The SELECT statement reads selected lines from the database table SPFLI. Within the SELECT loop, the WRITE, SKIP, and ULINE statements output fields of the table work area SPFLI as well as horizontal and vertical lines to the list. For information on all WRITE options used, see [Creating Simple Lists with the WRITE Statement \[Page 890\]](#). [Creating and Changing List and Column Headers \[Page 152\]](#) explains how to create the list and column headings.

For more information on the structure of this standard list, see [Structure of the Standard List \[Page 942\]](#).

Structure of the Standard List

The following topics provide information on the structure of the standard list.

The standard list consists of

[Standard Page Header \[Page 943\]](#)

[Standard Page \[Page 944\]](#)

Informations about the width of the standard list can be found in

[Width of the Standard List \[Page 945\]](#)

Standard Page Header

Standard Page Header

The standard page header consists at least of a two-line standard heading. The first line of the standard heading contains the list header and the page number. The second line is made up by a horizontal line. When executing the program, the list header is stored in the system field SY-TITLE. If necessary, you can include up to four lines of column headers and another horizontal line into the standard heading.

Example for Standard List				1
ID	Departure from	Arrival at	Time of	
	Departure Time	Arrival Time	Flight	

[Creating and Changing List and Column Headers \[Page 152\]](#) describes how to maintain list and column headers. In addition, you can adapt these headers from within the user interface of the standard list after the list is displayed (see [Modifying List and Column Headers \[Page 952\]](#)).

The width of the standard page header is automatically adapted to the window width.

If the user scrolls vertically through the list, the standard page header remains visible. Only the list beneath the header is scrolled.

If the user scrolls horizontally through the list, list header and page number remain visible.

Standard Page

Beneath the page header, the output data appear. The standard list consists of one single page of dynamic length (internal limit: 60,000 lines). The output length is determined by the current list size.

0017	NEW YORK 13:30:00	SAN FRANCISCO 16:31:00	06:01:00
0026	FRANKFURT 08:30:00	NEW YORK 09:50:00	08:20:00
0064	SAN FRANCISCO 09:00:00	NEW YORK 17:21:00	05:21:00
0400	FRANKFURT 10:10:00	NEW YORK 11:34:00	08:24:00
SAP *** SAP *** SAP *** SAP *** SAP *** SAP			

The output screen includes a vertical scrollbar that allows the user to scroll through lists whose pages are longer than the window.

Width of the Standard List

Width of the Standard List

The width of the standard list depends on the window width in the moment the program is executed. If the user's window size is less than or equal to the standard window size, the width of the standard page conforms to the standard window width. The user may have to scroll to view all parts of the list. If the user's window size exceeds the standard window width, the width of the standard list conforms to the window width selected. In short, the standard list is at least as wide as the standard window. The width of the standard window depends on the operating system.

The output screen includes a horizontal scrollbar that allows the user to scroll through lists wider than the window.

User Interface of the Standard List

The output screen of the standard list contains the standard menu bar and the standard toolbar of the R/3 system.



To scroll through the standard list, the system offers the scrollbars and the functions *First page*, *Previous page*, *Next page*, and *Last page*. To find certain patterns in lists, the user can choose *Edit → Find...*

The user can use the following list-specific functions:

[Printing the Output List \[Page 947\]](#)

[Saving a List \[Page 948\]](#)

[Modifying List and Column Headers \[Page 952\]](#)

Printing the Output List

Printing the Output List

To print the list displayed on the screen, the user chooses *List* → *Print*.

The printed standard page header differs from the displayed standard page header: it additionally contains the current date.

Printing the standard list created in [Example for a Standard List \[Page 940\]](#) results in:

```

12.01.1996          Example for Standard List          1
-----
ID  Departure from  Arrival at  Time of
   Departure Time  Arrival Time  Flight
-----
+-----+-----+-----+-----+
| 0017 | NEW YORK | SAN FRANCISCO | |
|      | 13:30:00 | 16:31:00      | 06:01:00 |
+-----+-----+-----+-----+
| 0064 | SAN FRANCISCO | NEW YORK | |
|      | 09:00:00 | 17:21:00      | 05:21:00 |
+-----+-----+-----+-----+
| 0400 | FRANKFURT | NEW YORK | |
|      | 10:10:00 | 11:34:00      | 08:24:00 |
+-----+-----+-----+-----+
| 0026 | FRANKFURT | NEW YORK | |
|      | 08:30:00 | 09:50:00      | 08:20:00 |
+-----+-----+-----+-----+
SAP *** SAP *** SAP *** SAP *** SAP *** SAP
Flight Information System
International Connections

```

Use this printing method only if you need a hardcopy of the screen list for testing purposes. For detailed information on how to print lists, see [Printing Lists \[Page 1126\]](#).

Saving a List

To save the displayed list, the user chooses *List* → *Save*. The following options appear:

[Saving the List in SAPoffice \[Page 949\]](#)

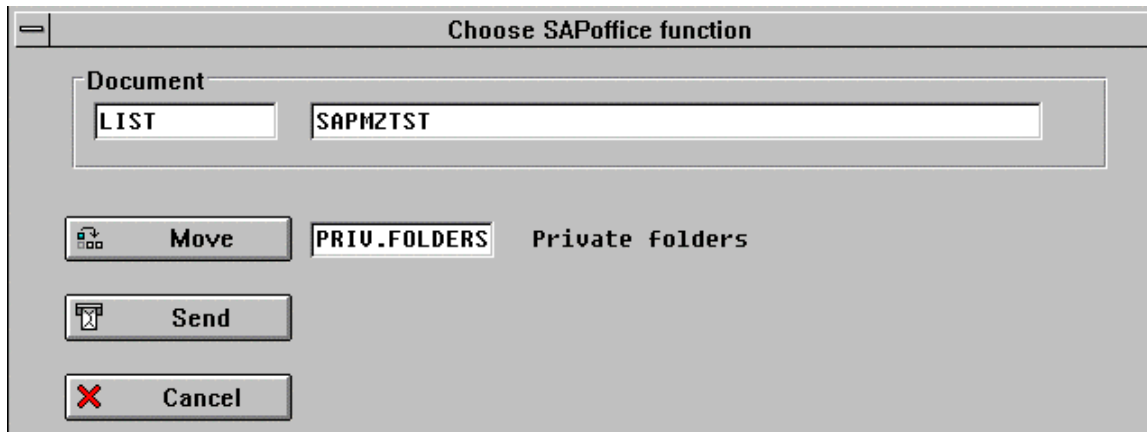
[Saving the List in the Reporting Tree \[Page 950\]](#)

[Saving the List as Local File on the Presentation Server \[Page 951\]](#)

Saving the List in SAPoffice

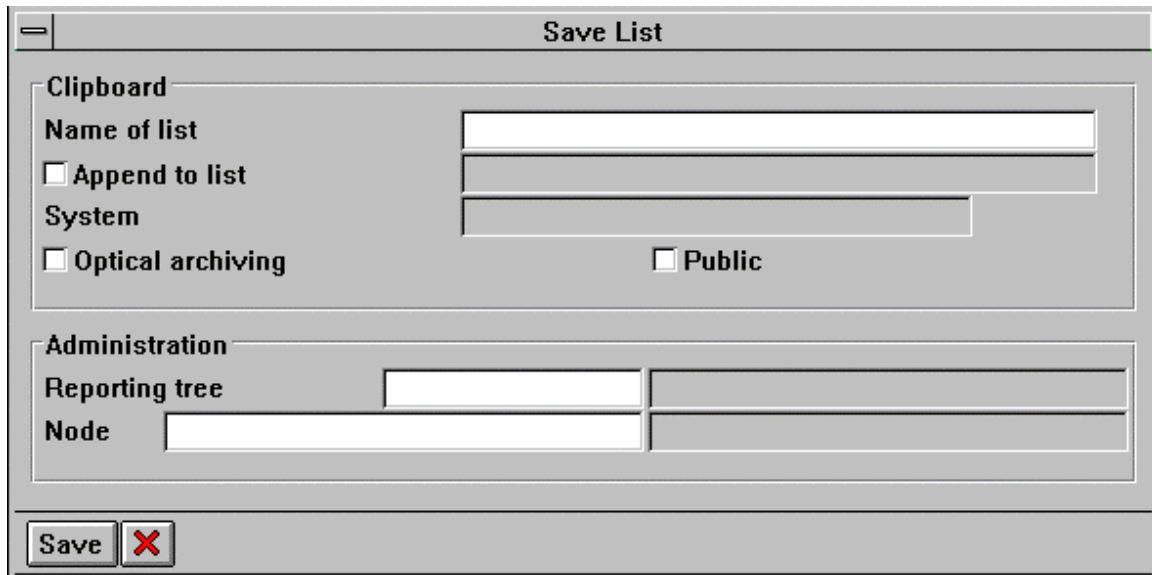
Saving the List in SAPoffice

When choosing *List* → *Save* → *Office*, a dialog asks the user to either store the displayed list in the user's Office folder or to send it to another user.



Saving the List in the Reporting Tree

When choosing *List* → *Save* → *Reporting tree*, a dialog asks the user to save the displayed list in the appropriate branch of the reporting tree.



The dialog box is titled "Save List". It contains two main sections: "Clipboard" and "Administration".

Clipboard section:

- Name of list:** A text input field.
- Append to list:** A checkbox.
- System:** A text input field.
- Optical archiving:** A checkbox.
- Public:** A checkbox.

Administration section:

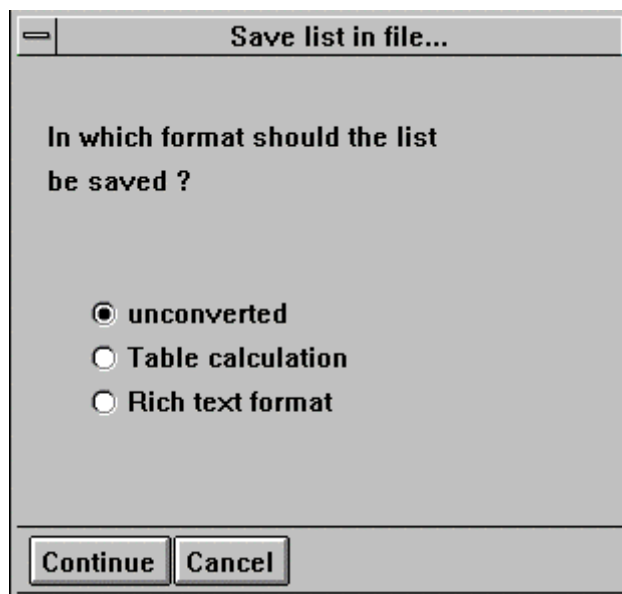
- Reporting tree:** A text input field.
- Node:** A text input field.

At the bottom of the dialog, there are two buttons: "Save" and a button with a red "X" icon.

Saving the List as Local File on the Presentation Server

Saving the List as Local File on the Presentation Server

When choosing *List* → *Save* → *File*, a dialog asks the user to store the displayed list as a local file, offering several format options.



The format options are:

- *unconverted*: the system stores the file as text file.
- *Table calculation*: the system inserts tabs between the columns.
- *Rich text format*: the system stores the data formatted for text processing.

If the user stores the standard list created in [Example for a Standard List \[Page 940\]](#) in the *Rich text format* and re-displays it using a text processing program that can read this format (for example, MS WORD), the list looks like this:

1996/03/05		Example for Standard List		1
ID	Departure from Departure Time	Arrival at Arrival Time	Time of Flight	
0017	NEW YORK 13:30:00	SAN FRANCISCO 16:31:00	06:01:00	
0064	SAN FRANCISCO 09:00:00	NEW YORK 17:21:00	05:21:00	
0400	FRANKFURT 10:10:00	NEW YORK 11:34:00	08:24:00	
0026	FRANKFURT 08:30:00	NEW YORK 09:50:00	08:20:00	
SAP *** SAP *** SAP *** SAP *** SAP *** SAP				
Flight Information System				
International Connections				

Modifying List and Column Headers

Usually, you create and modify list and column headers as text elements (see [Creating and Changing List and Column Headers \[Page 152\]](#)). However, you can also modify these headers while displaying the list on the screen. To do so, choose *System* → *List* → *List header*. The lines of the page header now accept input:

Example for Standard List			
ID	Departure from	Arrival at	Time of
	Departure Time	Arrival Time	Flight
0017	NEW YORK 13:30:00	SAN FRANCISCO 16:31:00	06:01:00

Use this function, for example, to position the column headers exactly above the columns of the displayed list.

Save your changes. The system stores the modified headers as text elements of the program in the text pool of the current logon language. For information on text elements, see [Working with Text Elements \[Page 146\]](#).

The Self-Defined List

You can modify the structure of the standard list and create lists of an individual structure. Use the options of the REPORT statement as well as the events TOP-OF-PAGE and END-OF-PAGE. The PROGRAM statement is equivalent to the REPORT statement and has the same options.

You have the following possibilities for modifications:

[Individual Page Header \[Page 954\]](#)

[Determining the List Width \[Page 956\]](#)

[Determining the Page Length \[Page 958\]](#)

[Defining a Page Footer \[Page 960\]](#)

If your list consists of several pages, you can structure each page differently. For information on how to do this, see [Lists with Several Pages \[Page 962\]](#).

Individual Page Header

To layout a page header individually, you must define it in the processing block following the event keyword TOP-OF-PAGE:

Syntax

```
TOP-OF-PAGE.  
WRITE:....
```

The TOP-OF-PAGE event occurs as soon as the system starts processing a new page of a list. The system processes the statements following TOP-OF-PAGE before outputting the first line on a new page. For information on events and processing blocks, see [Controlling the Flow of ABAP Programs by Events \[Page 1208\]](#).

Remember to end the processing block following TOP-OF-PAGE by using an appropriate event keyword, such as START-OF-SELECTION, if you want to start the actual list processing afterwards (see [Defining Processing Blocks \[Page 1209\]](#)).

The self-defined page header appears beneath the standard page header. If you want to suppress the standard page header, use the NO STANDARD PAGE HEADING option of the REPORT statement:

Syntax

```
REPORT <rep> NO STANDARD PAGE HEADING.
```

When using this statement, the system does not display a standard page header on the pages of a list of program <rep>. If you defined an individual page header using TOP-OF-PAGE, the system displays it.

During the event TOP-OF-PAGE, you can also fill the system fields SY-TVAR0 to SY-TVAR9 with values that should replace eventual placeholders &0 to &9 in the standard page header (see [Creating and Changing List and Column Headers \[Page 152\]](#)).

When scrolling vertically, the self-defined page header remains visible as does the standard page header. However, the self-defined page header consists of normal list lines and therefore does not adapt automatically to the window width.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING.  
TOP-OF-PAGE.  
WRITE: SY-TITLE, 40 'Page', SY-PAGNO.  
ULINE.  
WRITE: / 'SAP AG', 29 'Walldorf', ',SY-DATUM',  
       / 'Neurottstr. 16', / '69190 Walldorf/Baden'.  
ULINE.  
START-OF-SELECTION.
```

Individual Page Header

```
DO 5 TIMES.  
WRITE / SY-INDEX.  
ENDDO.
```

This program does not use the standard page header, but one that is self-defined following TOP-OF-PAGE. It is necessary to specify the event keyword START-OF-SELECTION to implicitly end the TOP-OF-PAGE processing block. The output appears as follows:

SAPMZTST	Page	1
SAP AG	Walldorf,	16.01.1996
Neurottstr. 16		
69190 Walldorf/Baden		
1		
2		
3		
4		
5		

The self-defined page header consists of six lines. The program title comes from the SY-TITLE system field, the page number from SY-PAGNO. The self-defined page header is not as wide as the list.

Determining the List Width

To determine the width of the output list, use the LINE-SIZE option of the REPORT statement.

Syntax

REPORT <rep> LINE-SIZE <width>.

This statement determines the width of the output list of program <rep> as <width> characters. If you set <width> to 0, the system uses the width of the standard list (see [Width of the Standard List \[Page 945\]](#)).

A line can be up to 255 characters long. However, if you intend to print the list, beware that most printers cannot print lists wider than 132 characters. If you want to print the list directly while creating it, the page width must comply with one of the existing print formats. Otherwise, the system will not print the list (see [Print Parameters \[Page 1130\]](#)). Make sure not to choose a list width exceeding 132 characters, unless you create the list for display only.

While creating the list, the system field SY-LINSZ contains the current line width. To adapt the list width to the current window width, see [Lists with Several Pages \[Page 962\]](#).

Horizontal lines you create using the ULINE statement (without the AT option) automatically adapt to the self-defined list width.

```
REPORT SAPMZTST LINE-SIZE 40.
WRITE: 'SY-LINSZ:', SY-LINSZ.
ULINE.
DO 20 TIMES.
  WRITE SY-INDEX.
ENDDO.
```

This program creates the following output:

Defining the width			1
SY-LINSZ: 40			
1	2	3	
4	5	6	
7	8	9	
10	11	12	
13	14	15	
16	17	18	
19	20		

The example uses the standard page header. If you replace the LINE-SIZE value 40 by 60, the output looks as follows:

Determining the List Width

Defining the width					1
SY-LINSZ: 60					
1	2	3	4	5	
6	7	8	9	10	
11	12	13	14	15	
16	17	18	19	20	

The standard page header and the underscores automatically conform to the list width.

Determining the Page Length

To determine the page length of an output list, use the LINE-COUNT option of the REPORT statement.

Syntax

REPORT <rep> LINE-COUNT <length>[(<n>)].

This statement determines the page length of the output list of program <rep> as <length> lines. If you specify the optional number <n>, the system reserves <n> lines of the page length for the page footer. Those lines of the page footer that are not filled at the END-OF-PAGE event, appear as blank lines (see [Defining a Page Footer \[Page 960\]](#)).

If you set <length> to zero, the system uses the standard page length (see [Standard Page \[Page 944\]](#)). To adapt the page length to the current window size, see [Lists with Several Pages \[Page 962\]](#). While creating the list, the system field SY-LINCT contains the current number of lines per page (that is <length>, or 0 for the standard page length).

Beware that the length of the page header is part of <length>. Thus, for the list itself you can use only <length> minus page header length minus <n> lines. If <length> is less than the page header length, a runtime error occurs.

If during list processing the system reaches the end of the area provided for the actual list, it outputs the page footer, if any, inserts some space, and starts a new page. The space inserted belongs to the list background and is not a line of the list. The SY-PAGNO system field always contains the current page number.

When determining a page length, keep the following items in mind:

- For screen output, use the standard page length to avoid page breaks in the middle of the screen.
- For printing lists, set the page length according to the printer requirements. Write your program in a way that it produces reasonable output for any page length. If you choose a page length that is not covered by one of the existing print formats, you may no longer be able to directly print a list while creating it. For more information on print formats, see [Printing Lists \[Page 1130\]](#).
- Use fixed length specifications for form-like lists of a specified page layout only. Before coding a program for such a list, check whether you can use predefined SAPscript forms. For information on forms, see the documentation [BC - Style- and Layout Set Maintenance \[Ext.\]](#).

The following program is meant to demonstrate the usage of the LINE-COUNT option. Therefore, the different pages of the list appear on one screen page.

```
REPORT SAPMZTST LINE-SIZE 40 LINE-COUNT 4.
```

```
WRITE: 'SY-LINCT:', SY-LINCT.  
SKIP.
```

Determining the Page Length

```
DO 6 TIMES.  
  WRITE / SY-INDEX.  
ENDDO.
```

This program determines the page length to be four lines. It uses the standard page header. Suppose, the standard page header consists of the two-line list header. The output then may look as follows:

Page	1
SY-LINCT:	4
Page	2
1	
2	
Page	3
3	
4	
Page	4
5	
6	

The list consists of four pages of four lines each. Each page is made up of the page header and two lines of the actual list. Note the space at the end of each page.

Defining a Page Footer

To define a page footer, use the END-OF-PAGE event. This event occurs if, while processing a list page, the system reaches the lines reserved for the page footer, or if the RESERVE statement triggers a page break (see [Conditional Page Break- Defining a Block of Lines \[Page 966\]](#)). Fill the lines of the page footer in the processing block following the event keyword END-OF-PAGE:

Syntax

```
END-OF-PAGE.  
WRITE:....
```

The system only processes the processing block following END-OF-PAGE if you reserve lines for the footer in the LINE-COUNT option of the REPORT statement (see [Determining the Page Length \[Page 958\]](#)).

Remember to end the processing block following END-OF-PAGE by using an appropriate event keyword, such as START-OF-SELECTION, if you want to start processing the actual list afterwards (see [Defining Processing Blocks \[Page 1209\]](#)).

```
REPORT SAPMZTST LINE-SIZE 40 LINE-COUNT 6(2)  
      NO STANDARD PAGE HEADING.  
  
TOP-OF-PAGE.  
  
WRITE: 'Page with Header and Footer'.  
      ULINE AT /(27).  
  
END-OF-PAGE.  
  
      ULINE.  
      WRITE: /30 'Page', SY-PAGNO.  
  
      START-OF-SELECTION.  
  
      DO 6 TIMES.  
        WRITE / SY-INDEX.  
      ENDDO.
```

This program consists of three processing blocks. The standard page header is turned off. The page length is set to six lines, reserving two of them for the page footer.

Defining a Page Footer

Page with Header and Footer	
1	
2	
Page 1	
Page with Header and Footer	
3	
4	
Page 2	
Page with Header and Footer	
5	
6	
Page 3	

The list consists of three pages of six lines each. Each page is made up of the self-defined two-line page header, of two lines of actual list, and of a two-line page footer. The current page number displayed in the page footer comes from the SY-PAGNO system field.

Lists with Several Pages

If in your program you write more lines to the output page of a list than are defined in the LINE-COUNT option of the REPORT statement, the system automatically creates a new page (see [Determining the Page Length \[Page 958\]](#)). Each new page contains the page header as well as the page footer defined for the program (if any).

Apart from automatic page breaks, you can use the NEW-PAGE and RESERVE statements to code page breaks explicitly. The options of the NEW-PAGE statement allow you to layout each page individually. You also need NEW-PAGE to print lists from within the program (see [Printing Lists \[Page 1146\]](#)).

The following topics explain

[Programming Page Breaks \[Page 963\]](#)

[Standard Page Headers of Individual Pages \[Page 968\]](#)

[Page Lengths of Individual Pages \[Page 970\]](#)

[Page Width of List Levels \[Page 974\]](#)

[Scrolling from within the Program \[Page 975\]](#)

Programming Page Breaks

To program unconditional page breaks, use the NEW-PAGE statement.

[Unconditional Page Break \[Page 964\]](#)

To program page breaks depending on the number of empty lines left on a page, use the RESERVE statement.

[Conditional Page Break- Defining a Block of Lines \[Page 966\]](#)

Unconditional Page Break

To trigger a page break during list processing, use the basic form of the NEW-PAGE statement:

Syntax

NEW-PAGE.

This statement

- ends the current page. All other output appears on a new page.
- only starts a new page if output is written to the current page as well as to the new page after NEW-PAGE. The system then increases the SY-PAGNO system field by one. You cannot produce empty pages.
- does not trigger the END-OF-PAGE event. This means that the system does not output a page footer even if one is defined.

```
REPORT SAPMZTST LINE-SIZE 40.  
TOP-OF-PAGE.  
WRITE: 'TOP-OF-PAGE', SY-PAGNO.  
ULINE AT /(17).  
START-OF-SELECTION.  
DO 2 TIMES.  
  WRITE / 'Loop:'.  
DO 3 TIMES.  
  WRITE / SY-INDEX.  
ENDDO.  
NEW-PAGE.  
ENDDO.
```

This sample program uses both the standard page header whose list header 'Standard Page Header' is defined as text element, and a self-defined page header. Both page headers appear on each page.

Unconditional Page Break

Standard Page Header	1
TOP-OF-PAGE	1
Loop:	
1	
2	
3	
Standard Page Header	2
TOP-OF-PAGE	2
Loop:	
1	
2	
3	

The DO loop encounters the NEW-PAGE statement twice, but executes a page break only once. After the second NEW-PAGE statement, no output is available.

Conditional Page Break- Defining a Block of Lines

To execute a page break under the condition that less than a certain number of lines is left on a page, use the RESERVE statement:

Syntax

RESERVE <n> LINES.

This statement triggers a page break if less than <n> free lines are left on the current list page between the last output and the page footer. <n> can be a variable. Before starting a new page, the system processes the END-OF-PAGE event. RESERVE only takes effect if output is written to the subsequent page. No blank pages are created.

The RESERVE statement thus defines a block of lines that must be output as a whole. To find out which additional practical effects a block of lines may have, see [Positioning Output in the First Line of a Line Block \[Page 992\]](#).

```
REPORT SAPMZTST LINE-SIZE 40 LINE-COUNT 8(2).
END-OF-PAGE.
ULINE.
START-OF-SELECTION.
DO 4 TIMES.
  WRITE / SY-INDEX.
ENDDO.
DO 2 TIMES.
  WRITE / SY-INDEX.
ENDDO.
RESERVE 3 LINES.
WRITE: / 'LINE 1',
      / 'LINE 2',
      / 'LINE 3'.
```

The list header of the standard page header of this sample program is defined as 'Standard Page Header'. The REPORT statement determines the page length to be eight lines. Two of them are used for the standard page header, another two are reserved for the page footer. The page footer consists of a horizontal line and a blank line. Thus, for outputting the actual list, four lines per page remain. The first DO loop fills these four lines. Then the END-OF-PAGE event occurs, after which the system automatically starts a new page. After the second DO loop, the RESERVE statement triggers the END-OF-PAGE event and a page break, since the number of free lines left on the page is less than three. The output looks like this:

Conditional Page Break- Defining a Block of Lines

Standard Page Header	1
1	
2	
3	
4	
Standard Page Header	2
1	
2	
Standard Page Header	3
LINE 1	
LINE 2	
LINE 3	

The three lines on page 3 form a block of lines.

Standard Page Headers of Individual Pages

The standard page header consists of list and column headers (see [Standard Page Header \[Page 943\]](#)). To influence the representation of these individual components of the standard page header, use the following options of the NEW-PAGE statement:

Syntax

NEW-PAGE [NO-TITLE|WITH-TITLE] [NO-HEADING|WITH-HEADING].

Use the NO-TITLE or WITH-TITLE option to suppress or display the standard header on all pages to come. The default for basic lists is WITH-TITLE, for secondary lists NO-TITLE.

Use the NO-HEADING or WITH-HEADING option to suppress or display the column headers on all pages to come. The default for basic lists is WITH-HEADING, for secondary lists NO-HEADING.

For information on basic and secondary lists, see [Interactive Lists \[Page 1030\]](#).

Even if you suppress the standard page header by using the NO STANDARD PAGE HEADING option of the REPORT statement, you can activate the display of the individual components using WITH-TITLE and WITH-HEADING.

The NEW-PAGE statement does not affect the display of a page header that you defined at the event TOP-OF-PAGE, since this event is processed on the new page (see [Individual Page Header \[Page 954\]](#)).

```
REPORT SAPMZTST LINE-SIZE 40.  
WRITE: 'Page', SY-PAGNO.
```

```
NEW-PAGE NO-TITLE.  
WRITE: 'Page', SY-PAGNO.
```

```
NEW-PAGE NO-HEADING.  
WRITE: 'Page', SY-PAGNO.
```

```
NEW-PAGE WITH-TITLE.  
WRITE: 'Page', SY-PAGNO.
```

```
NEW-PAGE WITH-HEADING.  
WRITE: 'Page', SY-PAGNO.
```

This program creates five pages with different page headers. The list header text element is defined as 'Standard Page Header', the column header as 'Column'.

Standard Page Headers of Individual Pages

Standard Page Header		1
Column		
Page	1	
Column		
Page	2	
Page	3	
Standard Page Header		4
Page	4	
Standard Page Header		5
Column		
Page	5	

Pages 1 and 5 contain the complete standard page header. On page 2, the list header is missing. On page 3, the entire page header is suppressed. On page 4, the column header is left out.

Page Lengths of Individual Pages

To determine the page length of each page individually, use the NEW-PAGE statement:

Syntax

NEW-PAGE LINE-COUNT <length>.

This statement determines the page length of the subsequent pages as <length>. <length> can be a variable. If you set <length> to zero, the system uses the standard page length (see [Standard Page \[Page 944\]](#)). The page header is part of the page and thus of the page length.

You cannot use NEW-PAGE to create or change a page footer. A page footer defined in the REPORT statement (see [Determining the Page Length \[Page 958\]](#)) is kept as such, independent of a NEW-PAGE statement.

For the actual list output, <length> minus page header length is available.

When using the LINE-COUNT option of the NEW-PAGE statement, refer to the notes in [Determining the Page Length \[Page 958\]](#).

To adapt the page length to the current window length, set <length> to SY-SROWS. The SY-SROWS system field contains the number of lines of the current window.

```
REPORT SAPMZTST LINE-SIZE 40 LINE-COUNT 0(1).
END-OF-PAGE.
ULINE.
START-OF-SELECTION.
NEW-PAGE LINE-COUNT 5.
DO 4 TIMES.
  WRITE / SY-INDEX.
ENDDO.

WRITE: / 'Next Loop:'.

NEW-PAGE LINE-COUNT 6.
DO 6 TIMES.
  WRITE / SY-INDEX.
ENDDO.
```

This program creates five pages of different lengths. The list header text element is defined as 'Standard Page Header'. The REPORT statement reserves one line of each page for the page footer. The page footer is defined at the END-OF-PAGE event as a horizontal line. The first NEW-PAGE statement sets the page length to 5, the second to 6.

Page Lengths of Individual Pages

Standard Page Header	1
1	
2	
Standard Page Header	2
3	
4	
Standard Page Header	3
Next Loop:	
Standard Page Header	4
1	
2	
3	
Standard Page Header	5
4	
5	
6	

The first NEW-PAGE statement does not start a new page, since no output was written to the list before. The standard page header uses two lines of each page for the list header. The page footer uses one line. For the first DO loop, two lines per page can be used to WRITE output. All page breaks within the DO loop occur **automatically** as soon as the list processing reaches the page footer. The system then displays the page footer. The second NEW-PAGE creates a page break from page 3 to 4. Here, the END-OF-PAGE event is not processed. For the second DO loop, three lines per page can be used to WRITE output. Page breaks again occur **automatically**. The page footer appears.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING
      LINE-SIZE 40 LINE-COUNT 0(2).
```

```
TOP-OF-PAGE.
```

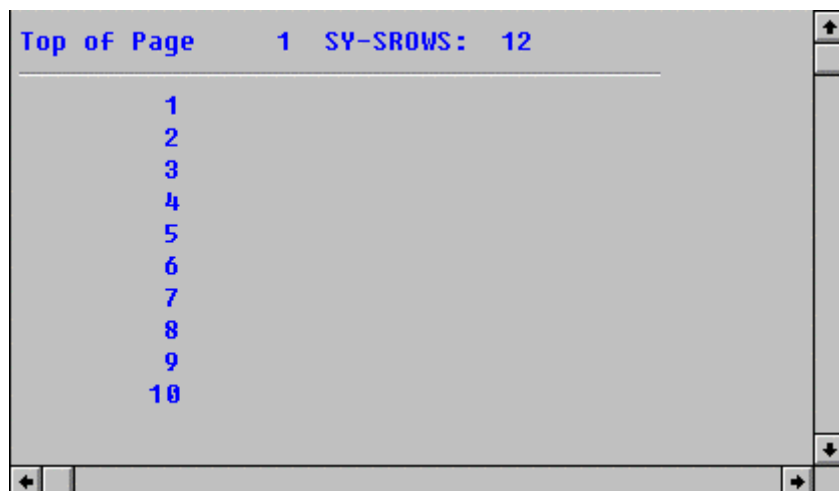
```
WRITE: 'Top of Page', SY-PAGNO,
      'SY-SROWS:', SY-SROWS.
```

```
ULINE.
```

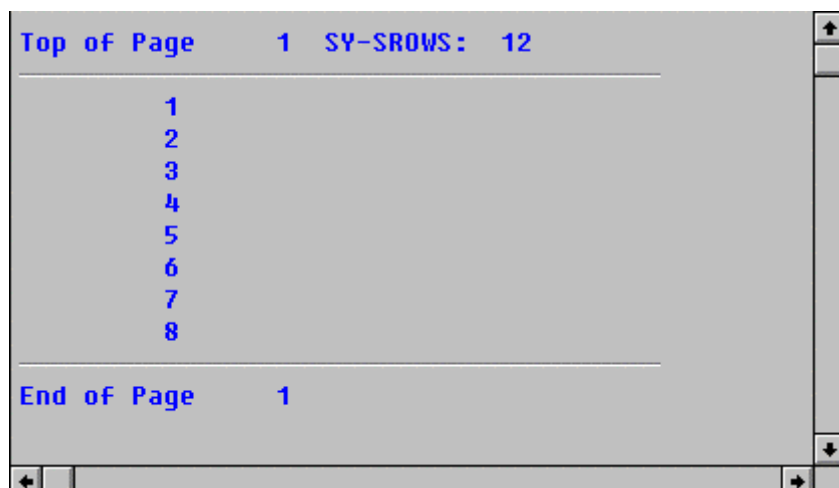
```
END-OF-PAGE.
```

```
ULINE.  
WRITE: 'End of Page', SY-PAGNO.  
  
START-OF-SELECTION.  
  
* NEW-PAGE LINE-COUNT SY-SROWS.  
  
DO 100 TIMES.  
  WRITE / SY-INDEX.  
ENDDO.
```

This program creates one single endless page, since the NEW-PAGE statement is marked as comment:



The system displays as many lines as possible in the current window, which has a length of 12 lines. In the figure above, the 12 lines are made up of two self-defined header lines and 10 lines of the actual list. When scrolling vertically, the page header remains visible. If you remove the asterisk in front of the NEW-PAGE statement and keep the current window length, the output looks as follows:



The list is now separated into several pages where, according to SY-SROWS, each page is 12 lines long. Of these 12 lines, two are reserved for the page header and two for the footer. In this list, the user can scroll explicitly using *Next page* (for example, to page 11):

Page Lengths of Individual Pages

Top of Page	11	SY-SROWS:	12
81			
82			
83			
84			
85			
86			
87			
88			
End of Page	11		

Page Width of List Levels

You **cannot** change the width of individual pages within a list level. You can only change the width of **all** pages of a new list level. To do so, use the NEW-PAGE statement:

Syntax

NEW-PAGE LINE-SIZE <width>.

All list levels starting from the new page have a width of <width> instead of the one specified in the REPORT statement. If you set <width> to zero, the system uses the width of the standard list (see [Width of the Standard List \[Page 945\]](#)).

If you set <width> to SY-SCOLS, you can adapt the width of the new list level to the window width, even if the window is smaller than the standard window. The SY-SCOLS system field contains the number of characters of a line of the current window.

Within a list level, that is, if the next page is not the beginning of a new list level, the system ignores the LINE-SIZE option.

For information on how to create new list levels, see [Interactive Lists \[Page 1030\]](#).

Scrolling from within the Program

Scrolling from within the Program

From within the program, you can scroll through lists vertically and horizontally. Use the SCROLL keyword. Scrolling from within the program makes sense, for example, if you want to scroll to certain pages as a reaction to user input.

The SCROLL statement takes effect on completed lists only. If you use this statement before the first output statement of a list, it does not affect this list. If you use SCROLL after the first output statement of a list, it affects the entire list, including any output statements to come.

After each SCROLL statement, you can query SY-SUBRC to see whether the system succeeded. SY-SUBRC is 0 if the system successfully scrolled, and 4 if scrolling was not possible, because it would exceed the list boundaries. If you are working with several list levels, SY-SUBRC may also be 8, indicating that the list level you specified does not exist (see [Scrolling through Interactive Lists \[Page 1098\]](#)).

The SCROLL statement allows you

Vertical Scrolling

[Scrolling Window by Window \[Page 976\]](#)

[Scrolling by Pages \[Page 977\]](#)

Horizontal Scrolling

[Scrolling to the List's Margins \[Page 979\]](#)

[Scrolling by Columns \[Page 980\]](#)

Scrolling Window by Window

To scroll through a list vertically by the size of the current window and independent of the page length, use this statement:

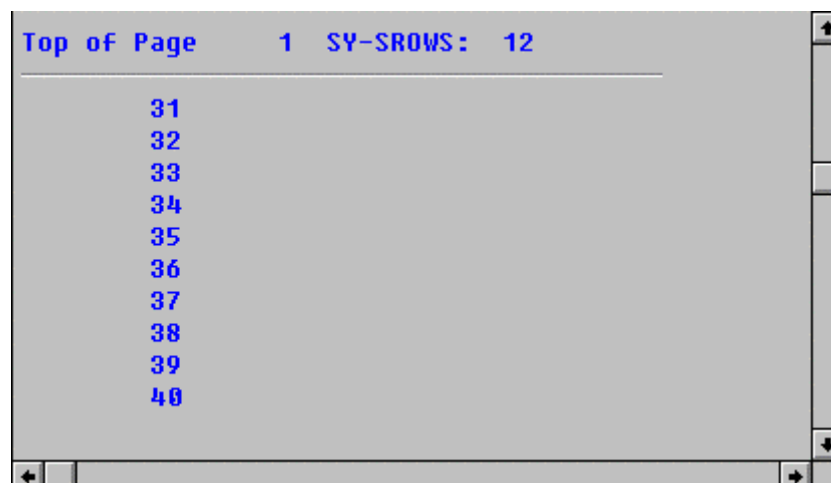
Syntax

SCROLL LIST FORWARD|BACKWARD [INDEX <idx>].

Without the INDEX option, the statement scrolls forward or backward through the current list by the size of the current window. When using the INDEX option, the system scrolls through the list on list level <idx>. For more information on scrolling in list levels, see [Scrolling through Interactive Lists \[Page 1098\]](#).

```
REPORT SAPMZTST NO STANDARD PAGE HEADING LINE-SIZE 40.  
TOP-OF-PAGE.  
WRITE: 'Top of Page', SY-PAGNO, 'SY-SROWS:', SY-SROWS.  
ULINE.  
START-OF-SELECTION.  
DO 100 TIMES.  
  WRITE / SY-INDEX.  
ENDDO.  
DO 3 TIMES.  
  SCROLL LIST FORWARD.  
ENDDO.
```

This executable program (report) creates a list of one endless page. Within the DO loop, the system executes the SCROLL statement three times. If the current window has a length of 12 lines (stored in SY-SROWS), the output of the program appears as below:



Note that the actual list is scrolled by SY-SROWS minus the number of header lines. The user can continue scrolling into both directions.

Scrolling by Pages

Scrolling by Pages

To scroll a list by pages, that is, scroll vertically depending on the page length, the SCROLL statement offers several options.

Scrolling to Certain Pages

To scroll to certain pages, use the TO option of the SCROLL statement:

Syntax

```
SCROLL LIST TO FIRST PAGE | LAST PAGE | PAGE <pag>
  [INDEX <idx>] [LINE <lin>].
```

Without the INDEX option, the statement scrolls the current list to the first, to the last, or to the page numbered <pag>. With the INDEX option, the system scrolls the list of list level <idx>. For information on list levels, see [Interactive Lists \[Page 1030\]](#).

With the LINE option, the system displays the page to which it scrolls starting from line <lin> of the actual list. It does not count the page header lines.

Scrolling by a Certain Number of Pages

To scroll a list by a certain number of pages, use the following options of the SCROLL statement:

Syntax

```
SCROLL LIST FORWARD | BACKWARD <n> PAGES [INDEX <idx>].
```

Without the INDEX option, the statement scrolls forward or backward <n> pages. The INDEX option refers to a certain list level, as mentioned above.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING
  LINE-SIZE 40 LINE-COUNT 8(2).
```

```
DATA: PAG TYPE I VALUE 15,
      LIN TYPE I VALUE 4.
```

```
TOP-OF-PAGE.
```

```
WRITE: 'Top of Page', SY-PAGNO.
ULINE.
```

```
END-OF-PAGE.
```

```
ULINE.
WRITE: 'End of Page', SY-PAGNO.
```

```
START-OF-SELECTION.
```

```
DO 100 TIMES.
  DO 4 TIMES.
    WRITE / SY-INDEX.
  ENDDO.
ENDDO.
```

```
SCROLL LIST TO PAGE PAG LINE LIN.
```

This program creates a list of 100 pages with 8 lines per page. On each page, four lines are used for page header and page footer. Due to the SCROLL statement, the output of the program appears as follows:

Top of Page	15	
	4	
End of Page	15	
Top of Page	16	
	1	
	2	
	3	
	4	
End of Page	16	
Top of Page	17	
	1	
	2	
	3	
	4	
End of Page	17	
Top of Page	18	
	1	

The list display starts at page 15. Due to the LINE option, the first three lines of the actual list are scrolled beneath the page header.

Scrolling to the List's Margins

Scrolling to the List's Margins

To scroll horizontally to the left or right margin of a list, use the following options of the SCROLL statement:

Syntax

SCROLL LIST LEFT | RIGHT [INDEX <idx>].

Without the INDEX option, the statement scrolls to the left or right margin of the current list. With the INDEX option, the system scrolls the list of list level <idx>. For information on list levels, see [Interactive Lists \[Page 1030\]](#).

```
REPORT SAPMZTST NO STANDARD PAGE HEADING LINE-SIZE 200.
```

```
TOP-OF-PAGE.
```

```
WRITE: AT 161 'Top of Page', SY-PAGNO,  
       'SY-SCOLS:', SY-SCOLS.
```

```
ULINE.
```

```
START-OF-SELECTION.
```

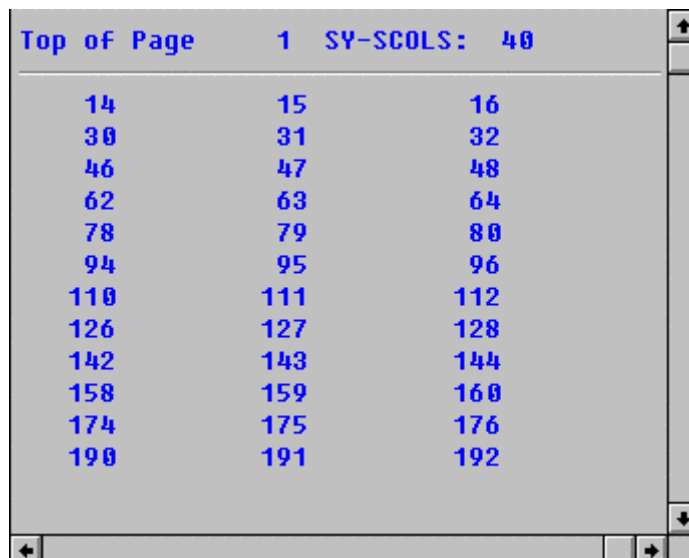
```
DO 200 TIMES.
```

```
  WRITE SY-INDEX.
```

```
ENDDO.
```

```
SCROLL LIST RIGHT.
```

This program creates a list of one page with a width of 200. If the current window width (stored in SY-SCOLS) equals 40, the output of the program looks as follows:



Top of Page 1 SY-SCOLS: 40		
14	15	16
30	31	32
46	47	48
62	63	64
78	79	80
94	95	96
110	111	112
126	127	128
142	143	144
158	159	160
174	175	176
190	191	192

The list display is scrolled to the right margin. The user can now use the scroll bar to scroll to the left.

Scrolling by Columns

To scroll a list horizontally by columns, the SCROLL statement offers several options. A column in this case means one character of the list line.

Scrolling to Certain Columns

To scroll to certain columns, use the TO COLUMN option of the SCROLL statement:

Syntax

SCROLL LIST TO COLUMN <col> [INDEX <idx>].

Without the INDEX option, the system displays the current list starting from column <col>. With the INDEX option, the system scrolls the list at list level <idx>. For information on list levels, see [Interactive Lists \[Page 1030\]](#).

Scrolling by a Certain Number of Columns

To scroll a list by a certain number of columns, use the following option of the SCROLL statement:

Syntax

SCROLL LIST LEFT | RIGHT BY <n> PLACES [INDEX <idx>].

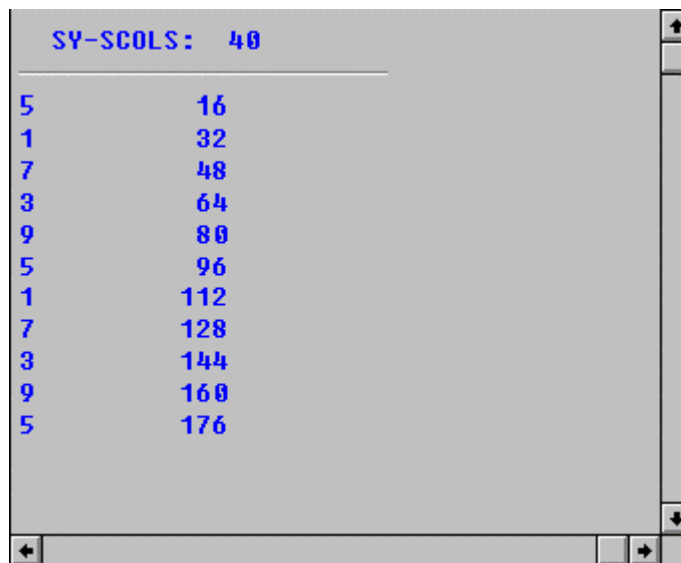
Without the INDEX option, the system scrolls the current list to the left or right by <n> columns. The INDEX option refers to a specified list level, as mentioned above.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING LINE-SIZE 200.
TOP-OF-PAGE.
WRITE: AT 161 'Top of Page', SY-PAGNO,
       'SY-SCOLS:', SY-SCOLS.
ULINE.
START-OF-SELECTION.
DO 200 TIMES.
  WRITE SY-INDEX.
ENDDO.

SCROLL LIST TO COLUMN 178.
```

This program creates a list of one page with a width of 200. If the current window width (stored in SY-SCOLS) equals 40, the output of the program looks as follows:

Scrolling by Columns



SY-SCOLS: 40	
5	16
1	32
7	48
3	64
9	80
5	96
1	112
7	128
3	144
9	160
5	176

The list is displayed starting from column 178. The user can now scroll to the left of the list.

Laying Out List Pages

The layout of a list page determines how well organized, and therefore easy to read, a list appears. Not the amount of information gathered on one page is important, but the way the information is presented. The human eye can cope much better with small blocks of data. And it is equally important that columns or lines containing a new block of information are separated visually from the preceding blocks. When laying out a list page, you should use several blanks or vertical lines to separate individual columns. Before outputting lines that contain a new item of information, draw a blank or underscore line.

The following topics explain the possibilities ABAP offers for laying out list pages.

[Positioning the Output \[Page 983\]](#)

[Formatting Output \[Page 995\]](#)

[Special Output Formats \[Page 1007\]](#)

[Creating Blank Lines \[Page 1012\]](#)

[Drawing Lines, Frames, and Grids \[Page 1014\]](#)

[Determining Which Part of a Page to Scroll Horizontally \[Page 1025\]](#)

Positioning the Output

You can position the output of WRITE and ULINE statements anywhere on the current page. The WRITE, SKIP, or ULINE statements following a position specification may overwrite existing output. For the current output position, refer to the system fields

- SY-COLNO (for the current column)
- SY-LINNO (for the current line)

You can use these system fields to navigate on the page.

ABAP offers a number of keywords to change the absolute as well as the relative output positions. See the following topics:

[Absolute Positioning \[Page 984\]](#)

[Relative Positioning \[Page 989\]](#)

SAP intends to allow only read access to the system fields SY-COLNO and SY-LINNO. Therefore, to position your output, **only** use the statements described in these topics. **Do not** position output by directly assigning values to these system fields. In that case, SAP cannot guarantee the contents of the system fields, since such an assignment does not trigger a plausibility check. Even though today it is possible to assign a column number to SY-COLNO that is outside the page, it doesn't make sense to do so.

Absolute Positioning

After specifying an absolute position, the subsequent output is written to the screen starting at fixed lines and columns.

[Horizontal Positioning \[Page 985\]](#)

[Vertical Positioning \[Page 986\]](#)

[Positioning Output Beneath the Page Header \[Page 987\]](#)

[Example for Absolute Positioning \[Page 988\]](#)

Horizontal Positioning

Horizontal Positioning

To specify a horizontal output position, ABAP offers two ways:

The AT option of the WRITE and ULINE statements (see [Positioning WRITE Output on the Screen \[Page 894\]](#)) and the POSITION statement. The POSITION statement's syntax is:

Syntax

POSITION <col>.

This statement sets the horizontal output position and the SY-COLNO system field to <col>. If <col> lies outside the page, the subsequent output statements are ignored.

The system writes an output following the POSITION statement or a WRITE statement formatted using AT to the specified position, regardless of whether there is enough space. That part of the output that does not fit onto the line, is truncated. Other WRITE output then starts on the next line.

Vertical Positioning

Specify vertical output positions like this:

Syntax

SKIP TO LINE <n>.

This statement sets the vertical output position and the SY-LINNO system field to <lin>. If <lin> does not lie between 1 and the page length, the system ignores the statement.

When using LINE, the system also counts page header and page footer lines.
Make sure that you do not unintentionally overwrite header or footer lines.

Positioning Output Beneath the Page Header

Positioning Output Beneath the Page Header

To position output on the first line following the entire page header, use the BACK statement:

Syntax

BACK.

If this statement does not follow a RESERVE statement, the subsequent output appears beneath the page header. The system sets SY-COLNO to 1 and SY-LINNO according to the length of the page header. In combination with the RESERVE statement, another rule applies (see [Positioning Output in the First Line of a Line Block \[Page 992\]](#)).

If you specify BACK at the TOP-OF-PAGE event, the system does not set the output position to beneath the entire page header, but only to beneath the standard page header. Any output written now overwrites the self-defined page header specified at TOP-OF-PAGE.

Example for Absolute Positioning

```
REPORT SAPMZTST NO STANDARD PAGE HEADING LINE-SIZE 60.
DATA: X(3), Y(3).
X = SY-COLNO. Y = SY-LINNO.
TOP-OF-PAGE.
WRITE: 'Position of Header: ', X, Y.
ULINE.

START-OF-SELECTION.

SKIP TO LINE 10.
POSITION 20.
X = SY-COLNO. Y = SY-LINNO.
WRITE: '* <- Position', X, Y.

SKIP TO LINE 12.
ULINE AT 20(20).

BACK.
X = SY-COLNO. Y = SY-LINNO.
WRITE: 'Position after BACK:', X, Y.
```

This program creates the following list page:

```
Position of Header:   1   1
-----
Position after BACK:  1   3

                                     * <- Position 20  10
-----
```

The system assigns the original values of SY-COLNO and SY-LINNO to the fields X and Y. Note that this assignment actually takes place at the START-OF-SELECTION event (see [Defining Processing Blocks \[Page 1209\]](#)). The original output position is the position of the first header line. The output is written there. SKIP TO LINE and POSITION place an asterisk '*' in column 20, line 10. SKIP TO LINE and AT produce underlining. Finally, BACK sets the output position to column 1, line 3 beneath the two-line page header.

Relative Positioning

Relative Positioning

Relative positions refer to output previously written to the list. Some relative positionings happen automatically. When using WRITE without position, an output appears one blank column after the previous output. If in the current line there is not enough space, a line feed occurs. ULINE and SKIP statements without positioning produce a line feed.

To program relative positioning, use either the SY-COLNO and SY-LINNO system fields together with the statements described in [Absolute Positioning \[Page 984\]](#) or the statements for relative positioning described below.

[Producing a Line Feed \[Page 990\]](#)

[Positioning Output Underneath Other Output \[Page 991\]](#)

[Positioning Output in the First Line of a Line Block \[Page 992\]](#)

[Examples for Relative Positioning \[Page 993\]](#)

Producing a Line Feed

To produce a line feed, use either the forward slash in the AT option of WRITE or ULINE or the NEW-LINE statement:

Syntax

NEW-LINE.

This statement positions output in a new line, setting SY-COLNO to 1 and increasing SY-LINNO by 1. The system only executes the statement if output was written to the screen since the last line feed. NEW-LINE does not create a blank line. To create a blank line, use the SKIP statement (see [Creating Blank Lines \[Page 1012\]](#)).

An automatic line feed occurs at the NEW-PAGE statement and at the beginning of an event.

Positioning Output Underneath Other Output

Positioning Output Underneath Other Output

You can position a WRITE output in the same column as a previous WRITE output. Use the formatting option UNDER of the WRITE statement:

Syntax

WRITE <f> UNDER <g>.

The system starts outputting <f> in the same column from which field <g> started. This statement is not limited to the current page, that is, <g> must not appear on the same page.

Make sure to adjust the vertical position so that you do not overwrite previous output.

The reference field <g> must be written exactly as in the corresponding WRITE statement, including all specifications such as offset (see [Specifying Offset Values for Data Objects \[Page 216\]](#)). If <g> is a text symbol (see [Text Symbols \[Page 157\]](#)), the system determines the reference field from the number of the text symbol.

For an example, see [Examples for Relative Positioning \[Page 993\]](#).

Positioning Output in the First Line of a Line Block

To set the next output line to the first line of a block of lines defined with the RESERVE statement (see [Conditional Page Break- Defining a Block of Lines \[Page 966\]](#)), use the BACK statement as follows:

Syntax

```
RESERVE.
```

```
.....
```

```
BACK.
```

If BACK follows RESERVE, the subsequent output appears in the first line written after RESERVE. You can use this statement, for example, to jump back to a certain line after writing an output from within a loop.

For an example, see [Examples for Relative Positioning \[Page 993\]](#).

Examples for Relative Positioning

Examples for Relative Positioning

The first example shows how to create a columnar list by means of a self-defined page header.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING
      LINE-SIZE 80 LINE-COUNT 7.

DATA: H1(10) VALUE ' Number',
      H2(10) VALUE ' Square',
      H3(10) VALUE ' Cube',
      N1 TYPE I, N2 TYPE I, N3 TYPE I,
      X TYPE I.

TOP-OF-PAGE.
      X = SY-COLNO + 8. POSITION X. WRITE H1.
      X = SY-COLNO + 8. POSITION X. WRITE H2.
      X = SY-COLNO + 8. POSITION X. WRITE H3.
      X = SY-COLNO + 16. POSITION X. WRITE SY-PAGNO.
ULINE.

START-OF-SELECTION.

DO 10 TIMES.
      N1 = SY-INDEX. N2 = SY-INDEX ** 2. N3 = SY-INDEX ** 3.
      NEW-LINE.
      WRITE: N1 UNDER H1,
            N2 UNDER H2,
            N3 UNDER H3.
ENDDO.
```

This program creates a two-page list. In the self-defined page header, column headers are positioned relatively by using the SY-COLNO system field and the POSITION statement. The actual list output is positioned underneath the fields of the header line using the UNDER option of the WRITE statement. The output appears as below:

Number	Square	Cube	1
1	1	1	
2	4	8	
3	9	27	
4	16	64	
5	25	125	
Number	Square	Cube	2
6	36	216	
7	49	343	
8	64	512	
9	81	729	
10	100	1.000	

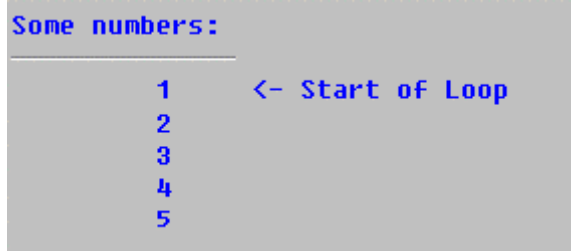
The different output positions of the individual fields result from the ABAP default of writing character strings left-justified and numeric fields right-justified. To influence the justification, use the formatting options LEFT-JUSTIFIED, RIGHT-

JUSTIFIED, and CENTERED of the WRITE statement (see [Formatting Options \[Page 896\]](#)).

The second example shows the effect of the BACK statement following RESERVE.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING LINE-SIZE 40.
DATA X TYPE I.
WRITE 'Some numbers:' NO-GAP.
X = SY-COLNO.
ULINE AT / (X) .
RESERVE 5 LINES.
DO 5 TIMES.
  WRITE / SY-INDEX.
ENDDO.
X = SY-COLNO.
BACK.
WRITE AT X '  <- Start of Loop'.
```

This program creates the following output:



```
Some numbers:
1      <- Start of Loop
2
3
4
5
```

After outputting the first two lines, the RESERVE statement is used to define the subsequent five lines as a block. The output following BACK is written to the first line of the block. Note how the SY-COLNO system field is used to underline the first line and to position the last WRITE output.

Formatting Output

Formatting Output

To format the list output, ABAP offers several formatting options.

The formatting options of the WRITE statement are described in [Formatting Options \[Page 896\]](#).

Other important formatting options, for example, to determine the color of the output or to make list fields accept input, are the formatting options of the FORMAT statement. They are described in the subsequent topics. You can use all options of the FORMAT statement as formatting options of the WRITE statement.

[The FORMAT Statement \[Page 996\]](#)

[Colors in Lists \[Page 997\]](#)

[Enabling Fields for Input \[Page 1003\]](#)

[Outputting Fields as Hotspots \[Page 1005\]](#)

The FORMAT Statement

To set formatting options statically in the program, use the FORMAT statement as follows:

Syntax

FORMAT <option₁> [ON|OFF] <option₂> [ON|OFF]....

The formatting options <option_i> set in the FORMAT statement apply to all subsequent output until they are turned off using the OFF option. The ON option to turn on a formatting option is optional, that is, you can leave it out.

To set the formatting options dynamically at runtime, use the FORMAT statement as follows:

Syntax

FORMAT <option₁> = <var₁> <option₂> = <var₂>....

The system interprets the variables <var_i> as numbers. Therefore, they should be of data type I. If the contents of <var_i> is zero, the variable has the same effect as the OFF option. If the contents of <var_i> is unequal to zero, the variable either has the same effect as the ON option or, together with the COLOR option, acts like the corresponding color number (see [Colors in Lists \[Page 997\]](#)).

If you use the same formatting options for a WRITE statement that follows the FORMAT statement, the settings of the WRITE statement overwrite the corresponding settings of the FORMAT statement for the current output.

For each new event, the system resets all formatting options to their default values. For a list of events, see [Events and their Event Keywords \[Page 1211\]](#). All formatting options have the default value OFF, except the INTENSIFIED option (see [Colors in Lists \[Page 997\]](#)).

To set all formatting options to OFF in one go, use:

Syntax

FORMAT RESET.

The following topics describe the available formatting options.

Colors in Lists

Colors in Lists

The options COLOR, INTENSIFIED, and INVERSE of the FORMAT statement influence the colors of the output list.

To set colors in the program, use:

Syntax

FORMAT COLOR <n> [ON] INTENSIFIED [ON|OFF] INVERSE [ON|OFF].

To set colors at runtime, use:

Syntax

FORMAT COLOR = <c> INTENSIFIED = <int> INVERSE = <inv>.

These formatting options do not apply to horizontal lines created by ULINE. They have the following functions:

- COLOR sets the color of the line background. If, in addition, INVERSE ON is set, the system changes the foreground color instead of the background color.

For <n> you can set either a color number or a color specification. Instead of color number 0, however, you must use OFF. If you set the color numbers at runtime, all values of <c> that are less than zero or greater than seven, lead to undefined results. The following table summarizes the different possibilities:

	<n>	<c>	Color	Intended for
OFF	or COL_BACKGROUND	0	depends on GUI	background
1	or COL_HEADING	1	grey-blue	headings
2	or COL_NORMAL	2	light grey	list bodies
3	or COL_TOTAL	3	yellow	totals
4	or COL_KEY	4	blue-green	key columns
5	or COL_POSITIVE	5	green	positive threshold value
6	or COL_NEGATIVE	6	red	negative threshold value
7	or COL_GROUP	7	violett	group levels

The default setting is COLOR OFF.

- INTENSIFIED determines the color palette for the line background.
With one exception (COLOR OFF), the color palette for the line background specified above can be intensified or normal. The default setting is INTENSIFIED ON. For COLOR OFF, the system changes the foreground color instead of the background color. If, in addition, INVERSE ON is set, then INTENSIFIED OFF is without effect (again with the exception of COLOR OFF).
- INVERSE influences the foreground color.

With one exception (COLOF OFF), the system takes the COLOR specified from an inverse color palette and uses it as foreground color. The background color remains unchanged. For COLOR OFF, INVERSE has no effect, since this would set the foreground and the background to the same color.

The following statements have the same effect:

FORMAT INTENSIFIED ON.	and	SUMMARY.
FORMAT INTENSIFIED OFF.	and	DETAIL.

For reasons of better readability, SAP recommends to always use the FORMAT statement.

The following examples show the colors possible in lists and how to use them.

[Demonstrating the Colors Available in Lists \[Page 999\]](#)

[Example for Using Colors in Lists \[Page 1001\]](#)

For another demonstration of colors in lists, call the executable program SHOWCOLO in any system.

Demonstrating the Colors Available in Lists

Demonstrating the Colors Available in Lists

The following example shows the different combinations of the color formatting options:

```
REPORT SAPMZTST.

DATA I TYPE I VALUE 0.
DATA COL(15).

WHILE I < 8.

CASE I.
  WHEN 0. COL = 'COL_BACKGROUND '.
  WHEN 1. COL = 'COL_HEADING '.
  WHEN 2. COL = 'COL_NORMAL '.
  WHEN 3. COL = 'COL_TOTAL '.
  WHEN 4. COL = 'COL_KEY '.
  WHEN 5. COL = 'COL_POSITIVE '.
  WHEN 6. COL = 'COL_NEGATIVE '.
  WHEN 7. COL = 'COL_GROUP '.
ENDCASE.

FORMAT INTENSIFIED COLOR = I.
WRITE: /(4) I, AT 7 SY-VLINE,
      COL, SY-VLINE,
      COL INTENSIFIED OFF, SY-VLINE,
      COL INVERSE.

I = I + 1.

ENDWHILE.
```

In the FORMAT statement, the COLOR option for the subsequent WRITE statements is set at runtime. The other options are set individually for each WRITE statement in the program.

The output appears as shown in the following table:

Colors in Lists			
Color	Intensified ON	Intensified OFF	Inverse ON
0	COL_BACKGROUND	COL_BACKGROUND	COL_BACKGROUND
1	COL_HEADING	COL_HEADING	COL_HEADING
2	COL_NORMAL	COL_NORMAL	COL_NORMAL
3	COL_TOTAL	COL_TOTAL	COL_TOTAL
4	COL_KEY	COL_KEY	COL_KEY
5	COL_POSITIVE	COL_POSITIVE	COL_POSITIVE
6	COL_NEGATIVE	COL_NEGATIVE	COL_NEGATIVE
7	COL_GROUP	COL_GROUP	COL_GROUP

The standard page header was created as a text element. In the online help, due to technical reasons, the colors of this list differ slightly from the colors of the R/3 system.

Example for Using Colors in Lists

Example for Using Colors in Lists

This example shows how to use colors in lists to accentuate output.

The following executable program (report) is connected to the logical database F1S.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING LINE-SIZE 70.
```

```
TABLES: SPFLI, SFLIGHT.
```

```
DATA SUM TYPE I.
```

```
TOP-OF-PAGE.
```

```
WRITE 'List of Flights' COLOR COL_HEADING.
```

```
ULINE.
```

```
GET SPFLI.
```

```
FORMAT COLOR COL_HEADING.
```

```
WRITE: 'CARRID', 10 'CONNID', 20 'FROM', 40 'TO'.
```

```
FORMAT COLOR COL_KEY.
```

```
WRITE: / SPFLI-CARRID UNDER 'CARRID',
```

```
      SPFLI-CONNID UNDER 'CONNID',
```

```
      SPFLI-CITYFROM UNDER 'FROM',
```

```
      SPFLI-CITYTO UNDER 'TO'.
```

```
ULINE.
```

```
FORMAT COLOR COL_HEADING.
```

```
WRITE: 'Date', 20 'Seats Occupied', 50 'Seats Available'.
```

```
ULINE.
```

```
SUM = 0.
```

```
GET SFLIGHT.
```

```
IF SFLIGHT-SEATSOCC LE 10.
```

```
  FORMAT COLOR COL_NEGATIVE.
```

```
ELSE.
```

```
  FORMAT COLOR COL_NORMAL.
```

```
ENDIF.
```

```
WRITE: SFLIGHT-FLDATE UNDER 'Date',
```

```
      SFLIGHT-SEATSOCC UNDER 'Seats Occupied',
```

```
      SFLIGHT-SEATSMAX UNDER 'Seats Available'.
```

```
SUM = SUM + SFLIGHT-SEATSOCC.
```

```
GET SPFLI LATE.
```

```
ULINE.
```

```
WRITE: 'Total Bookings:' INTENSIFIED OFF,
```

```
      SUM UNDER SFLIGHT-SEATSOCC COLOR COL_TOTAL.
```

```
ULINE
```

```
SKIP.
```

The report creates the following output list:

Example for Using Colors in Lists

List of Flights			
CARRID	CONNID	FROM	TO
AA	0017	NEW YORK	SAN FRANCISCO
Date	Seats Occupied		Seats Available
01/30/1995	10		660
02/01/1995	20		660
06/01/1995	38		660
06/04/1995	38		660
Total Bookings:		106	
CARRID	CONNID	FROM	TO
AA	0026	FRANKFURT	NEW YORK
Date	Seats Occupied		Seats Available
01/20/1995	10		280
01/22/1995	20		280
05/22/1995	38		280
05/25/1995	38		280
Total Bookings:		106	
CARRID	CONNID	FROM	TO
AA	0064	SAN FRANCISCO	NEW YORK
Date	Seats Occupied		Seats Available
01/30/1995	10		220
02/01/1995	20		220
06/01/1995	38		220
06/04/1995	38		220

All headings appear using the background color COL_HEADING. The key fields from Table SPFLI use COL_KEY as background color. The list body at the event GET SFLIGHT has a different line background color (COL_NORMAL) than the list background (COL_BACKGROUND). In addition, flights where the number of bookings falls below a certain minimum number, have a red background. The total number of bookings for each flight has a yellow background.

Note that the system resets the formatting options for each new event to the default settings (COLOR OFF, INTENSIFIED ON). For this reason, in the above program the line background of the output 'Total Bookings:' at the GET LATE event is COL_BACKGROUND again. INTENSIFIED is set to OFF to get the same foreground color as for the other output.

In the online help, due to technical reasons, the colors of this list differ slightly from the colors of the R/3 system.

Enabling Fields for Input

Enabling Fields for Input

You can make output fields in lists input-enabled. The user can change these fields on the screen. The changes can then be printed, or you use the READ LINE statement from the interactive list processing to process the changes (see [Interactive Lists \[Page 1030\]](#)). If you make the contents of variables input-enabled in the output list, the changes the user enters do not affect the variables themselves.

To make output fields input-enabled from within the program, use the FORMAT statement as follows:

Syntax

FORMAT INPUT [ON|OFF].

To make output fields input-enabled at runtime, use:

Syntax

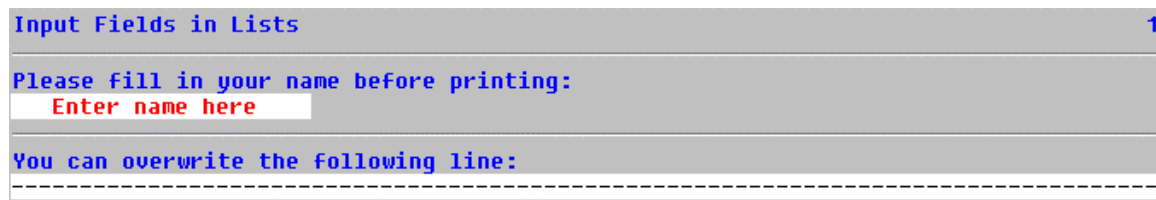
FORMAT INPUT = <i>.

Use the ON option (or <i> unequal to zero) to format subsequent output as input-enabled fields. Input-enabled fields have different background and foreground colors than the remainder of the list. For input fields, the options COLOR, INVERSE, and HOTSPOT have no effects. The INTENSIFIED option changes the foreground color of the input fields.

You can make horizontal lines input-enabled by formatting them as input fields. However, blank lines you created using SKIP cannot accept input.

```
REPORT SAPMZTST.  
  
WRITE 'Please fill in your name before printing:'.  
  
WRITE / ' Enter name here ' INPUT ON.  
ULINE.  
  
WRITE 'You can overwrite the following line:'.  
  
FORMAT INPUT ON INTENSIFIED OFF.  
ULINE.  
FORMAT INPUT OFF INTENSIFIED ON.
```

In this program, a WRITE statement directly receives the INPUT ON format and a horizontal ULINE line is formatted using the FORMAT statement. The heading is defined as a text element. The output appears as follows:



The screenshot shows the output of the program. It has a title bar 'Input Fields in Lists' with a blue icon and a page number '1' in the top right corner. The main content area has a light gray background. The first line is 'Please fill in your name before printing:' in blue. Below it is an input field with a white background and red text 'Enter name here'. The second line is 'You can overwrite the following line:' in blue. Below it is a horizontal line with a dashed underline, indicating it is an input-enabled field.

Due to INTENSIFIED OFF, the foreground color of the second input field differs from that of the first. The user can enter into the input fields on the screen, for example:

Input Fields in Lists	1
Please fill in your name before printing:	
Zippo Typefast	
You can overwrite the following line:	
-----Demonstration for overwriting lines. -----	

Outputting Fields as Hotspots

Outputting Fields as Hotspots

Hotspots are special areas of an output list. If the user clicks once onto a hotspot field, an event is triggered (for example, AT LINE-SELECTION). For fields that are not defined as hotspots, double-clicking or a function key are needed to trigger an event. For information on events during list processing, see [Interactive Lists \[Page 1030\]](#).

To output areas as hotspots, use the following option of the FORMAT statement:

Syntax

FORMAT HOTSPOT [ON|OFF].

To designate fields as hotspots at runtime, use:

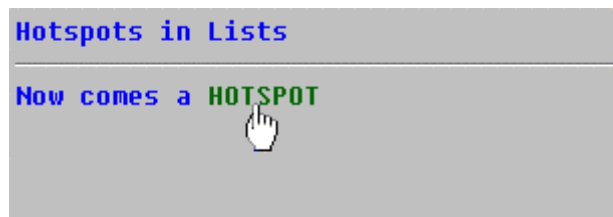
FORMAT HOTSPOT = <h>.

The ON option (or <h> unequal to zero) formats subsequent output as hotspot. If the user moves the mouse to such a field, the mouse pointer changes to a hand with pointed finger. As long as this hand is visible, a single click triggers an event. In addition to the changed mouse pointer, you may want to use a different color to accentuate the hotspot.

You cannot use the HOTSPOT option if INPUT ON is set, since with HOTSPOT ON the cursor can no longer be positioned on an input field. In addition, you cannot format horizontal lines created with ULINE and blank lines created with SKIP as hotspots.

```
REPORT SAPMZTST.  
INCLUDE <LIST>.  
  
START-OF-SELECTION.  
  
WRITE 'Now comes a'.  
  
FORMAT HOTSPOT ON COLOR 5 INVERSE ON.  
WRITE 'HOTSPOT'.  
FORMAT HOTSPOT OFF COLOR OFF.  
  
AT LINE-SELECTION.  
  
WRITE / 'New list AT-LINE-SELECTION'.  
SKIP.  
WRITE 'This is also a hotspot:'.  
  
WRITE ICON_LIST AS ICON HOTSPOT.
```

In this program, at the START-OF-SELECTION event part of the first line is formatted as hotspot. The standard page header is defined as a text element. If the user moves the mouse over the word HOTSPOT in the output, the mouse pointer changes to a hand:



A single click triggers the AT-LINE-SELECTION event. At this event, the program creates a secondary list containing another hotspot. The hotspot in the secondary list is an icon:



For information on the AT-LINE-SELECTION event and on secondary lists, see [Interactive Lists \[Page 1030\]](#).

Special Output Formats

For a summary of all formatting options of the WRITE statement, see [Formatting Options \[Page 896\]](#). The current topic describes some special formatting options. These options format output according to certain entries that have to be made in special database tables. Usually, the customer maintains these tables when customizing the application (for information on Customizing, refer to the documentation [Customizing \[Ext.\]](#)).

You can use the following output formats:

[Country-specific and User-specific Output Formats \[Page 1008\]](#)

[Currency-specific Output Formats \[Page 1010\]](#)

[Unit-specific Output Formats \[Page 1011\]](#)

Country-specific and User-specific Output Formats

The output formats of number and date fields are defined in the user master record of the individual program user. You can change these settings from within your program, using this statement:

Syntax

SET COUNTRY <c>.

For <c>, set either a country key defined in Table T005X or SPACE.

If <c> is not SPACE, the system turns off the settings from the user master record and searches Table T005X for the country key. If the key exists, the system sets SY-SUBRC to 0 and formats the output of all subsequent WRITE statements according to the settings defined in T005X. If the country key you specified does not exist, the system sets SY-SUBRC to 4 and formats for all subsequent WRITE statements the decimal characters as period '.' and date specifications as MM/DD/YY.

If <c> is SPACE, the system does not read Table T005X but uses the settings in the user master record. In this case, SY-SUBRC is always zero.

Maintaining Table T005X is part of the Customizing. However, you can use *System* → *Services* → *Table maintenance* to display or change entries.

```
REPORT SAPMZTST LINE-SIZE 40.
DATA: NUM TYPE P DECIMALS 3 VALUE '123456.789'.
ULINE.
WRITE: / 'INITIAL:'.
WRITE: / NUM, SY-DATUM.
ULINE.
SET COUNTRY 'US'.
WRITE: / 'US, SY-SUBRC:', SY-SUBRC.
WRITE: / NUM, SY-DATUM.
ULINE.
SET COUNTRY 'GB'.
WRITE: / 'GB, SY-SUBRC:', SY-SUBRC.
WRITE: / NUM, SY-DATUM.
ULINE.
SET COUNTRY 'DE'.
WRITE: / 'DE, SY-SUBRC:', SY-SUBRC.
WRITE: / NUM, SY-DATUM.
ULINE.
SET COUNTRY 'XYZ'.
WRITE: / 'XYZ, SY-SUBRC:', SY-SUBRC.
WRITE: / NUM, SY-DATUM.
ULINE.
SET COUNTRY SPACE.
WRITE: / 'SPACE, SY-SUBRC:', SY-SUBRC.
WRITE: / NUM, SY-DATUM.
ULINE.
```


Country-specific and User-specific Output Formats

This program outputs the packed number NUM and the system field SY-DATUM using different formatting options.

Initial:		
	123.456,789	1996/02/07
<hr/>		
US,	SY-SUBRC:	0
	123,456.789	02/07/1996
<hr/>		
GB,	SY-SUBRC:	0
	123,456.789	02-07-1996
<hr/>		
DE,	SY-SUBRC:	0
	123.456,789	07.02.1996
<hr/>		
XYZ,	SY-SUBRC:	4
	123,456.789	02/07/1996
<hr/>		
SPACE,	SY-SUBRC:	0
	123.456,789	1996/02/07
<hr/>		

The first and the last output are user-specific. For all other output, the system reads Table T005X. It does not find the entry 'XYZ' and sets the output format itself. The entries in T005X are customer-specific.

Currency-specific Output Formats

To format the output of a number field according to a certain currency, use the CURRENCY option of the WRITE statement:

Syntax

```
WRITE <f> CURRENCY <c>.
```

This statement determines the number of decimal places in the output according to the currency <c>. If the contents of <c> exists in Table TCURX as currency key CURRKEY, the system sets the number of decimal places according to the entry CURRDEC in TCURX. Otherwise, it uses the default setting of two decimal places. This means that Table TCURX must contain only exceptions where the number of decimal places is unequal to 2.

The output format for currencies does not depend on the decimal places of a number that may exist in the program. The system uses only the sequence of digits. This sequence of digits thus represents an amount specified in the smallest unit of the currency in use, for example Cents for US Dollar (USD) or Francs for Belgian Francs (BEF). For processing currency amounts in ABAP programs, SAP therefore recommends to use data type P without decimal places.

```
REPORT SAPMZTST LINE-SIZE 40.

DATA: NUM1 TYPE P DECIMALS 4 VALUE '12.3456',
      NUM2 TYPE P      VALUE '123456'.

SET COUNTRY 'US'.

WRITE: 'USD', NUM1 CURRENCY 'USD', NUM2 CURRENCY 'USD',
      / 'BEF', NUM1 CURRENCY 'BEF', NUM2 CURRENCY 'BEF',
      / 'KUD', NUM1 CURRENCY 'KUD', NUM2 CURRENCY 'KUD'.
```

This program defines two packed numbers NUM1 and NUM2, containing the same sequence of digits, but different numbers of decimal places. These numbers appear in the output in several currencies:

USD	1,234.56	1,234.56
BEF	123,456	123,456
KUD	123.456	123.456

For each currency, the output formats of NUM1 and NUM2 are the same, since they refer to the sequence of digits only. The currency US Dollar (USD) appears in the default setting of two decimal places, since the smallest unit is one Cent and a hundredth of a Dollar. For Belgian Francs (BEF), CURRDEC in TCURX is set to 0, since the Belgian Franc has no smaller units. Dinars from Kuwait (KUD) have units of a thousandth and therefore three decimal places (CURRDEC is 3).

Unit-specific Output Formats

Unit-specific Output Formats

You can format fields of type P according to certain units. For example, quantities should have no decimal places, weight specifications should have three decimal places, and so on. To do this, use the UNIT option of the WRITE statement:

Syntax

WRITE <f> UNIT <u>.

This statement sets the number of decimal places according to the unit <u>. The contents of <u> must be an entry in the database table T006 in the column MSEHI. The entry in column DECAN then determines the number of decimal places of the field <f> to be displayed. If the system does not find the entry <u> in Table T006, it ignores the option.

The following restrictions apply for this operation:

- <f> must be a packed number (type P).
- If <f> has less decimal places than the unit <u>, the system ignores the option.
- If <f> has more decimal places than the unit <u>, the system uses this option only, if the operation does not truncate any digits unequal to 0.

```
REPORT SAPMZTST LINE-SIZE 40.
```

```
DATA: NUM1 TYPE P DECIMALS 1 VALUE 1,  
      NUM2 TYPE P DECIMALS 4 VALUE '2.5'.
```

```
SET COUNTRY 'US'.
```

```
WRITE: 'KG', NUM1 UNIT 'KG', NUM2 UNIT 'KG',  
      / 'PC', NUM1 UNIT 'PC', NUM2 UNIT 'PC'.
```

This program defines two packed numbers, NUM1 with one decimal place and NUM2 with four decimal places. If the unit 'KG' (kilograms) has three decimal places in Table T006 and 'PC' (pieces) has zero decimal places in T006, the output appears as follows:

KG	1.0	2.500
PC	1	2.5000

The system ignores the option UNIT 'KG' for NUM1, since NUM1 has less than three decimal places. The UNIT 'PC' option shortens the output of NUM1 to zero decimal places. For NUM2, the system ignores the option UNIT 'PC', since otherwise a decimal place unequal to zero would have been truncated.

Creating Blank Lines

To create blank lines, use the SKIP statement as follows:

Syntax

SKIP [<n>].

The system writes <n> blank lines into the current list, starting at the current line. If you omit the <n> option, the system creates one blank line.

SKIP knows the following restrictions:

- If the number of lines remaining on the current page is too small, the above SKIP statement produces a page break, displaying the page footer if any. The system positions the next output to the first line beneath the page header of the new page.
- At the beginning of a page, the system executes the above statement only, if this page is the first page of a list level or if the page was created using the NEW-PAGE statement. For all other pages, the system ignores this statement at the beginning of a page.
- If the above statement is the last output statement of the last list page (that is, there are no more WRITE or ULINE statements), the system ignores it.

In the default setting, the system does not output any blank lines created using the WRITE statement with the AT / option. A blank line is a line that contains character strings only and whose individual fields consist of nothing but blank characters. However, if you intend to output blank lines created by WRITE statements when outputting character strings, use this statement:

Syntax

SET BLANK LINES ON|OFF.

With the ON option, the system in the output no longer suppresses blank lines created using WRITE statements. To reset the default setting, use the OFF option.

You use this statement, for example, to represent empty table entries in the list. Note that the system displays a line containing nothing but, for example, empty input fields or empty checkboxes only if you specify SET BLANK LINES ON beforehand.

The following program creates five blank lines. The output '*****' appears in the sixth line.

```
REPORT SAPMZTST.
```

```
SKIP 5.
```

```
WRITE '*****'.
```

The following program does not create any blank lines. The output '*****' appears in the first line. The SET BLANK LINES OFF statement is used only to accentuate the default setting.

```
REPORT SAPMZTST.
```

Creating Blank Lines

```
SET BLANK LINES OFF.
```

```
DO 5 TIMES
```

```
  WRITE / ' '.
```

```
ENDDO:
```

```
WRITE '*****'.
```

The program below creates five blank lines, since the SET BLANK LINES ON statement is used. The output '*****' appears in the sixth line.

```
REPORT SAPMZTST.
```

```
SET BLANK LINES ON.
```

```
DO 5 TIMES
```

```
  WRITE / ' '.
```

```
ENDDO.
```

```
SET BLANK LINES OFF.
```

```
WRITE / '*****'.
```

Drawing Lines, Frames, and Grids

ABAP offers several possibilities to create horizontal and vertical lines. For a list of the corresponding statements, see [Lines and Blank Lines on the Output Screen \[Page 899\]](#).

The system automatically joins lines that meet to united lines or frames. Lines meet if in the direction of at least one of the lines no blank characters or blank lines separate the lines. Depending on the type and number of lines meeting at a certain point, you can draw the following line sections.

[Straight Lines \[Page 1015\]](#)

[Corners \[Page 1016\]](#)

[T Sections \[Page 1017\]](#)

[Crosses \[Page 1018\]](#)

For a demonstration of automatically joined lines, call the executable program SHOWLINE in any system.

To exceptionally prevent the system from joining lines, you can use special lines:

[Using Special Lines \[Page 1019\]](#)

Note in this context that page header and page footer occupy lines of the page. This may lead to undesired joined lines if you forget to create enough blank lines. For example, outputting vertical lines '|' in the first line after the standard page header automatically joins these lines to the underline of the page header.

You can combine the different line segments available to layout your list in any way. The following examples show

[Programming Frames \[Page 1022\]](#)

[Programming Grids \[Page 1023\]](#)

Straight Lines

Straight Lines

The example below shows how to create horizontal and vertical straight lines.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING.  
  
SKIP TO LINE 3.  
ULINE AT 2 (1) .  
WRITE 4 '-' .  
WRITE 6 '--' .  
WRITE 9 '---' .  
ULINE AT 12 (4) .  
  
SKIP TO LINE 1.  
POSITION 18.  
WRITE '|' .  
  
SKIP TO LINE 3.  
DO 4 TIMES.  
  NEW-LINE.  
  POSITION 18.  
  WRITE '|' .  
ENDDO.
```

The output looks like this:



The first ULINE statement creates a horizontal line of one column. The hyphen in the first WRITE statement appears as normal output fields. The two hyphens of the second WRITE statement create a straight line of two columns width. The next three hyphens together with the ULINE statement create a straight line of seven columns width.

Outputting the first '|' character creates a vertical line in the first line. The other four '|' characters are joined to a vertical straight line of four lines length, starting at line 3.

Corners

The example below shows how to create different corners.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING.  
  
WRITE  '--'.  
WRITE  / '|'.  
  
SKIP TO LINE 1.  
ULINE AT 5(6).  
NEW-LINE.  
WRITE 10 '|'.  
  
SKIP TO LINE 4.  
WRITE: '|      |',  
      / '-----'.  
      /
```

The output looks like this:



Wherever the ends of vertical lines meet the ends of horizontal lines, corners appear.

T Sections

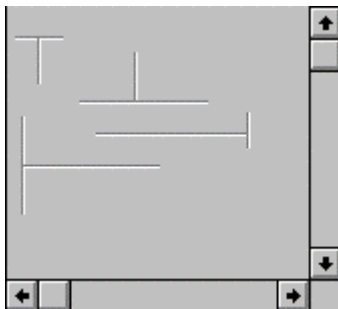
T Sections

The example below shows how to create different T sections.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING.  
  
WRITE '----'.  
WRITE /2 '|   |'.  
ULINE AT /5(8).  
  
SKIP TO LINE 4.  
DO 3 TIMES.  
  WRITE '|'.  
  NEW-LINE.  
ENDDO.  
SKIP TO LINE 5.  
WRITE '-----'.  
SKIP TO LINE 4.  
  
ULINE AT 6(10).  
WRITE 15 '|'.  

```

The output looks like this:



Wherever line ends meet lines at a right angle, T sections appear.

Crosses

The example below shows how to create crosses.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING.
```

```
WRITE ' |'.
```

```
WRITE /'-----'.
```

```
WRITE /' |'.
```

```
SKIP TO LINE 1.
```

```
DO 3 TIMES.
```

```
WRITE 12 SY-VLINE.
```

```
NEW-LINE.
```

```
ENDDO.
```

```
SKIP TO LINE 2.
```

```
ULINE AT 12(1).
```

The output looks like this:



If two lines intersect, a cross appears.

Using Special Lines

Using Special Lines

If you use tightly nested frames or tight hierarchy representations, you might want to keep certain line sections apart, even though there is no space left for inserting a blank character or blank line between them.

In this case, you can use special lines defined as system-defined constants in the include program <LINE>:

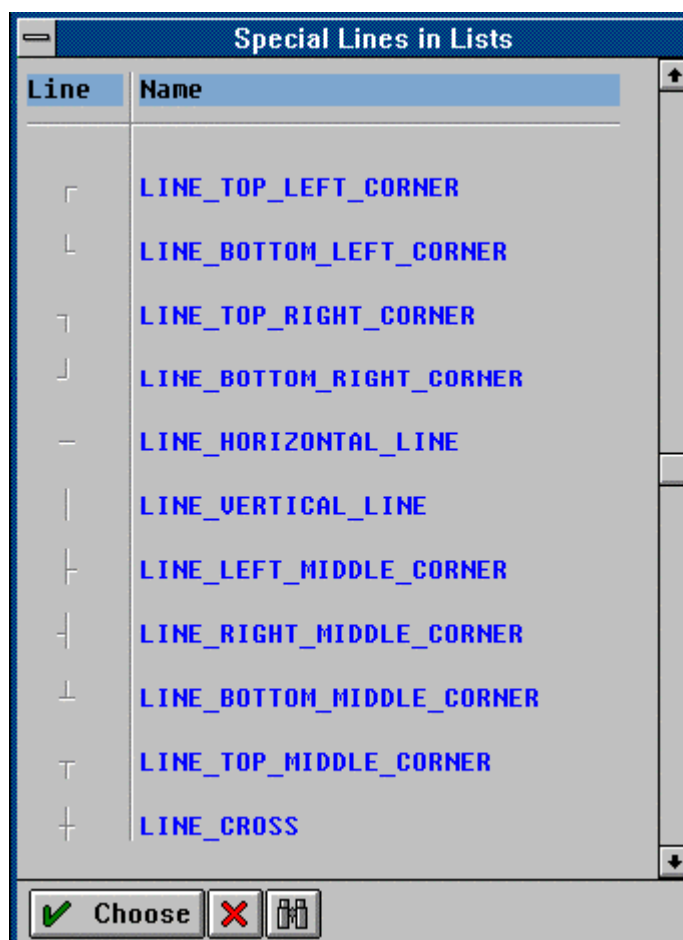
Syntax

WRITE <lin> AS LINE.

To be able to use special lines in your program, you must include the include program <LINE> or the more comprehensive include program <LIST> into your program. The include program <LINE> contains a short description of the special lines.

The system displays special lines in the output list exactly the way they are defined. Lines are joined only where they really meet. The system does not automatically lengthen special lines to make them meet.

The easiest way to output special lines is to use a ready-made keyword structure (see [Using WRITE via a Statement Structure \[Page 901\]](#)). From the screen *Assemble a WRITE Statement* select the radio button *Line* and then choose *Display*. The following dialog window appears:



It contains all available special lines which you can easily include into your program code.

The following program on one hand shows how to use special lines to create a tight pattern. On the other hand, it also demonstrates how you can program lines in lists dynamically.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING LINE-SIZE 60.
```

```
INCLUDE <LINE>.
```

```
DATA: X0 TYPE I VALUE 10,  
      Y0 TYPE I VALUE 10,  
      N TYPE I VALUE 16,  
      I TYPE I VALUE 0,  
      X TYPE I, Y TYPE I.
```

```
X = X0. Y = Y0. PERFORM POS.
```

```
WHILE I LE N.
```

```
  WRITE LINE_BOTTOM_LEFT_CORNER AS LINE.
```

```
  X = X + 1. PERFORM POS.
```

```
  ULINE AT X(I).
```

```
  X = X + I. PERFORM POS.
```

```
  WRITE LINE_BOTTOM_RIGHT_CORNER AS LINE.
```

```
  Y = Y - 1. PERFORM POS.
```

```
  DO I TIMES.
```

```
    WRITE '|'.  
    Y = Y - 1. PERFORM POS.
```

```
  ENDDO.
```

```
  WRITE LINE_TOP_RIGHT_CORNER AS LINE.
```

```
  I = I + 1.
```

```
  X = X - I. PERFORM POS.
```

```
  ULINE AT X(I).
```

```
  X = X - 1. PERFORM POS.
```

```
  WRITE LINE_TOP_LEFT_CORNER AS LINE.
```

```
  Y = Y + 1. PERFORM POS.
```

```
  DO I TIMES.
```

```
    WRITE '|'.  
    Y = Y + 1. PERFORM POS.
```

```
  ENDDO.
```

```
  I = I + 1.
```

```
ENDWHILE.
```

```
FORM POS.
```

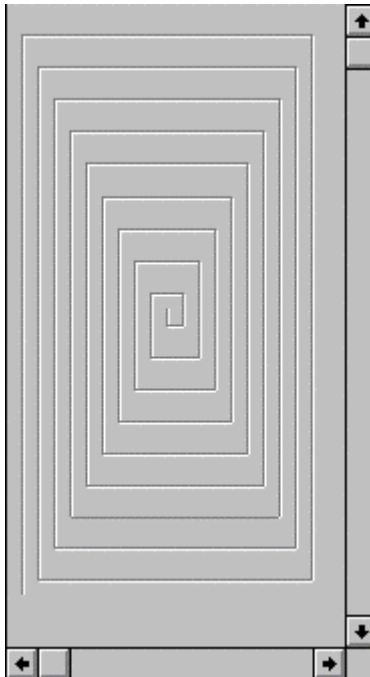
```
  SKIP TO LINE Y.
```

```
  POSITION X.
```

```
ENDFORM.
```

In this program, the position X,Y is set for each output using the subroutine POS.
The output appears as follows:

Using Special Lines



The program creates a tight helix structure which would not be possible without using special lines. You can set the number of coils using the variable N.

Programming Frames

You can use the line types available in ABAP to program frames. The sample program below defines a macro WRITE_FRAME which you can use instead of the WRITE <f> statement. The system draws a frame around a field <f> specified in WRITE_FRAME that dynamically adapts to the length of the field.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING LINE-SIZE 60.

DATA: X TYPE I, Y TYPE I, L TYPE I.

DEFINE WRITE_FRAME.
  X = SY-COLNO. Y = SY-LINNO.
  WRITE: '|' NO-GAP, &1 NO-GAP, '|' NO-GAP.
  L = SY-COLNO - X.
  Y = Y - 1. SKIP TO LINE Y. POSITION X.
  ULINE AT X(L).
  Y = Y + 2. SKIP TO LINE Y. POSITION X.
  ULINE AT X(L).
  Y = Y - 1. X = SY-COLNO. SKIP TO LINE Y. POSITION X.
END-OF-DEFINITION.

SKIP.
WRITE 'Demonstrating'.
WRITE_FRAME 'dynamic frames'.
WRITE 'in'.
WRITE_FRAME 'ABAP/4'.
WRITE 'output lists.'
```

The function of the macro WRITE_FRAME defined in the above program is demonstrated in the output below. For information on macros, see [Defining and Calling Macros \[Page 440\]](#).



Demonstrating dynamic frames in ABAP/4 output lists.

Programming Grids

Programming Grids

You can use the line types available in ABAP to program a grid for a table-type list. The sample program below defines two macros NEW_GRID and WRITE_GRID that belong together. NEW_GRID is used to initialize a grid and for line feeds within the grid. You can use WRITE_GRID instead of the WRITE <f> statement. For each field output using WRITE_GRID, the system draws a vertical grid line to the right of the field and a horizontal grid line below it. The horizontal line dynamically adapts to the length of the field. The lines of all output fields together form a grid.

```
REPORT SAPMZTST LINE-SIZE 60 NO STANDARD PAGE HEADING.
```

```
TABLES SPFLI.
```

```
DATA: X TYPE I, Y TYPE I, L TYPE I.
```

```
TOP-OF-PAGE.
```

```
WRITE 3 'List of Flights in a Dynamic Grid'
```

```
    COLOR COL_HEADING.
```

```
ULINE.
```

```
START-OF-SELECTION.
```

```
DEFINE NEW_GRID.
```

```
Y = SY-LINNO. Y = Y + 2. SKIP TO LINE Y.
```

```
X = SY-COLNO. POSITION X. WRITE '|'
```

```
END-OF-DEFINITION.
```

```
DEFINE WRITE_GRID.
```

```
X = SY-COLNO. Y = SY-LINNO. POSITION X.
```

```
WRITE: &1, '|'
```

```
L = SY-COLNO - X + 1.
```

```
X = X - 2. Y = Y + 1. SKIP TO LINE Y. POSITION X.
```

```
ULINE AT X(L).
```

```
Y = Y - 1. X = SY-COLNO. SKIP TO LINE Y. POSITION X.
```

```
END-OF-DEFINITION.
```

```
GET SPFLI.
```

```
NEW_GRID.
```

```
WRITE_GRID: SPFLI-CARRID,
```

```
    SPFLI-CONNID,
```

```
    SPFLI-CITYFROM,
```

```
    SPFLI-CITYTO.
```

The functions of the macros NEW_GRID and WRITE_GRID defined in the above program are demonstrated in the output below. For information on macros, see [Defining and Calling Macros \[Page 440\]](#). The executable program is connected to the logical database F1S. After entering the corresponding values on the selection screen, the output may appear as follows:

List of Flights in a Dynamic Grid			
AA	0017	NEW YORK	SAN FRANCISCO
DL	1699	NEW YORK	SAN FRANCISCO
UA	0007	NEW YORK	SAN FRANCISCO

Note that the topmost grid line comes from the ULINE statement in the statement block following TOP-OF-PAGE. The system automatically joins the vertical lines of the list body with this line.

Determining Which Part of a Page to Scroll Horizontally

Determining Which Part of a Page to Scroll Horizontally

You can restrict the area of a page that can be scrolled horizontally by the user with the scrollbars or from within the program (see [Scrolling from within the Program \[Page 975\]](#)).

The following topics describe:

[Excluding Lines from Horizontal Scrolling \[Page 1026\]](#)

[Left Boundary for Horizontal Scrolling \[Page 1028\]](#)

You can combine these two possibilities.

Excluding Lines from Horizontal Scrolling

To exclude a line (for example, a header or comment line) from horizontal scrolling, define the line feed for that line as follows:

Syntax

NEW-LINE NO-SCROLLING.

The line following the statement cannot be scrolled horizontally. However, it can be scrolled vertically.

To undo the above statement, use:

Syntax

NEW-LINE SCROLLING.

This statement makes sense only if after NEW-LINE NO-SCROLLING no line was output.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING  
LINE-COUNT 3 LINE-SIZE 140.
```

```
START-OF-SELECTION.
```

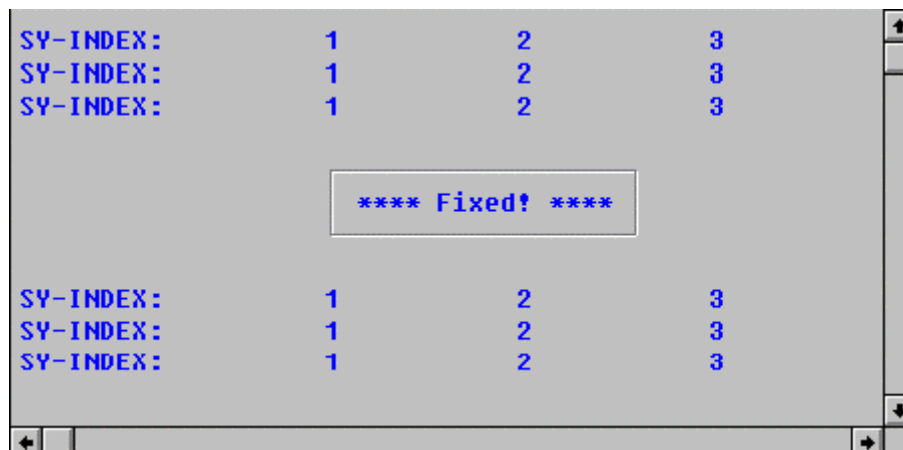
```
DO 3 TIMES.  
WRITE: / 'SY-INDEX:'.  
DO 10 TIMES.  
WRITE SY-INDEX.  
ENDDO.  
ENDDO.
```

```
NEW-LINE NO-SCROLLING.  
ULINE AT 20(20).  
NEW-LINE NO-SCROLLING.  
WRITE AT 20 '| **** Fixed! **** |'.  
NEW-LINE NO-SCROLLING.  
ULINE AT 20(20).
```

```
DO 3 TIMES.  
WRITE: / 'SY-INDEX:'.  
DO 10 TIMES.  
WRITE SY-INDEX.  
ENDDO.  
ENDDO.
```

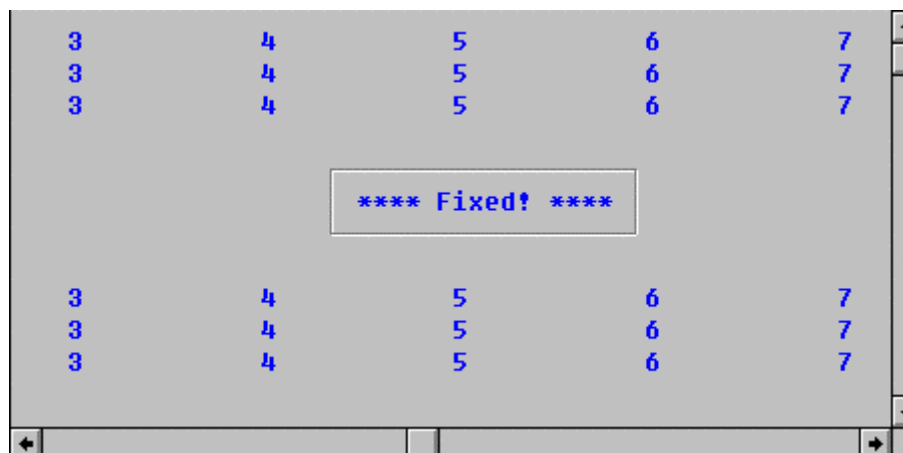
This program creates three pages of three lines each without page header or footer. The three lines of the second page cannot be scrolled due to NEW-LINE NO-SCROLLING. The program output looks like this:

Excluding Lines from Horizontal Scrolling



SY-INDEX:	1	2	3
SY-INDEX:	1	2	3
SY-INDEX:	1	2	3
**** Fixed! ****			
SY-INDEX:	1	2	3
SY-INDEX:	1	2	3
SY-INDEX:	1	2	3

If the user scrolls to the right, the output may look like this:



3	4	5	6	7
3	4	5	6	7
3	4	5	6	7
**** Fixed! ****				
3	4	5	6	7
3	4	5	6	7
3	4	5	6	7

The lines of the first and third pages scroll past the fixed lines.

Left Boundary for Horizontal Scrolling

To determine the left boundary of the horizontally scrollable area, use:

Syntax

SET LEFT SCROLL-BOUNDARY [COLUMN <col>].

Without the COLUMN option, the left boundary of the scrollable area of the current page is set to the current output position; with the COLUMN option, it is set to position <col>. Now, only the part to the right of this area can be scrolled horizontally.

The above statement applies to the entire current page, and only to it. You must repeat the statement for each new page, otherwise the system uses the default value (left list margin).

To set the same scrollable area for all pages of a list, you can execute the statement, for example, at the TOP-OF-PAGE event.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING
      LINE-COUNT 3 LINE-SIZE 140.

START-OF-SELECTION.

DO 3 TIMES.
  WRITE: /10 'SY-INDEX:'.
DO 10 TIMES.
  WRITE SY-INDEX.
ENDDO.
ENDDO.

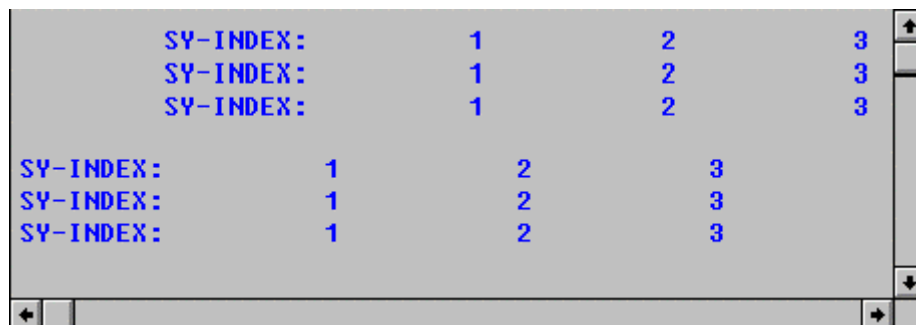
SET LEFT SCROLL-BOUNDARY COLUMN 20.

DO 3 TIMES.
  WRITE: / 'SY-INDEX:'.
DO 10 TIMES.
  WRITE SY-INDEX.
ENDDO.
ENDDO.

SET LEFT SCROLL-BOUNDARY COLUMN 10.
```

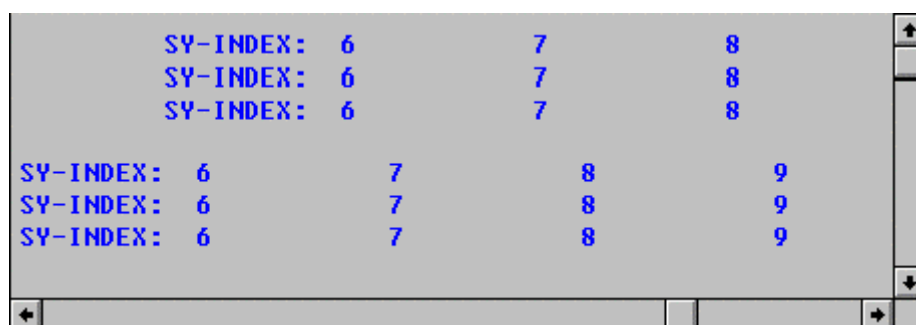
This program creates two pages of three lines each without page header or footer. The first SET statement affects the first page, since the automatic page break does not occur until the first WRITE statement of the second DO loop. The second SET statement affects the second page. The program output looks like this:

Left Boundary for Horizontal Scrolling



SY-INDEX:	1	2	3
SY-INDEX:	1	2	3
SY-INDEX:	1	2	3
SY-INDEX:	1	2	3
SY-INDEX:	1	2	3
SY-INDEX:	1	2	3

If the user scrolls to the right, the output may look like this:



SY-INDEX:	6	7	8
SY-INDEX:	6	7	8
SY-INDEX:	6	7	8
SY-INDEX:	6	7	8
SY-INDEX:	6	7	8
SY-INDEX:	6	7	8

The scrollable area of the first page disappears at a different position beneath the fixed area at the left margin than the scrollable area of the second page.

Interactive Lists

ABAP allows you to create interactive lists. On an interactive list on the screen, the user can select lines, type input, and enter commands using function keys, menu bars, or pushbuttons. Interactive lists enhance the classical type of output list with dialog functionality, thus coming close to dialog programming. Interactive lists provide the user with the so-called "interactive reporting" facility.

The subsequent sections describe

[What is Interactive Reporting? \[Page 1031\]](#)

[Event Control for Interactive Lists \[Page 1033\]](#)

[Basic Lists and Secondary Lists \[Page 1035\]](#)

[User Interfaces of Interactive Lists \[Page 1046\]](#)

[Passing Data from List to Report \[Page 1076\]](#)

[Manipulating Interactive Lists \[Page 1097\]](#)

[Calling Programs \[Page 1112\]](#)

What is Interactive Reporting?

What is Interactive Reporting?

A classical non-interactive report consists of one program that creates a single list. This means that after the report was started, the list it creates has to contain all data requested, regardless of the number of details the user wants to see. This procedure may result in extensive lists from which the user has to pick the relevant data. For background processing, this is the only possible method. After starting a background job, there is no way of influencing the program. The desired selections must be made beforehand and the list must provide detailed information.

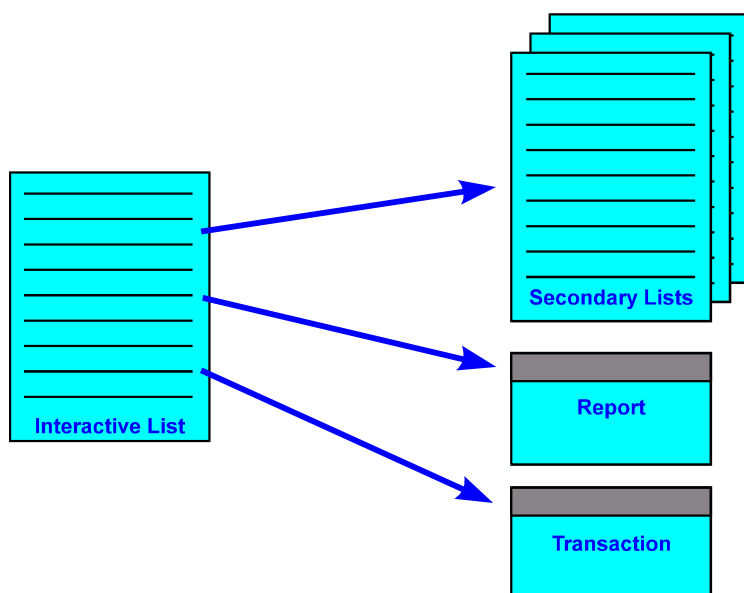
For dialog sessions, there are no such restrictions. The user is present during the execution of the program and can control and manipulate the program flow directly. To be able to use all advantages of the online environment, classical reporting was developed into interactive reporting.

Interactive reporting allows the user to participate actively in retrieving and presenting data during the session. Instead of one extensive and detailed list, with interactive reporting you create a condensed basic list from which the user can call detailed information by positioning the cursor and entering commands. Interactive reporting thus reduces information retrieval to the data actually required.

Detailed information is presented in secondary lists. A secondary list may either overlay the basic list completely or appear in an additional dialog window on the same screen. The secondary list can itself be interactive again.

Apart from creating secondary lists, interactive reporting also allows you to call transactions or other executable programs (reports) from lists. These programs then use values displayed in the list as input values. The user can, for example, call a transaction from within a list to change the database table whose data is displayed in the list.

Interactive Reporting



This section describes how to program lists for dialogs and explains the ABAP statements you can use to call executable programs (reports) and transactions and to pass data between the different components.

What is Interactive Reporting?

For an example showing the main features of interactive reporting, see [The HIDE Technique \[Page 1083\]](#).

Event Control for Interactive Lists

ABAP programs are controlled by event keywords (see bc060e.doc026). And event keywords are what you need to use lists interactively.

Events for Interactive Lists

The following events are specific for the context of interactive lists:

- AT LINE-SELECTION
- AT PF<nn>
- AT USER-COMMAND

If you define a processing block for one of these events in your program, the program can react on certain user actions. If the user then executes a defined action on the displayed list, the system triggers the corresponding event. The system writes the output of all output statements programmed in the processing blocks of one of the above events to a so-called secondary list. For more information on secondary lists, see [Basic Lists and Secondary Lists \[Page 1035\]](#).

Actions on Interactive Lists

An action that the user can execute on the list and that triggers a certain event, must be determined in the list's interface definition. You can define an individual interface for each of your lists. By default, the events occur after the following actions on lists:

- AT LINE-SELECTION occurs after the user double-clicks on a line, clicks once on a hotspot, or chooses *Edit* → *Choose*.
- AT PF<nn> occurs after the user presses the corresponding function key.
- AT USER-COMMAND occurs after the user chooses a self-defined action.

For more information on user actions and user interfaces, see [User Interfaces of Interactive Lists \[Page 1046\]](#).

Consequences from Event Control

The fact that you use event keywords to program interactive lists, has the following important consequences: As described in [Controlling the Flow of ABAP Programs by Events \[Page 1208\]](#), you cannot nest processing blocks, since each new event keyword terminates the previous processing block. Therefore, you cannot process other events within the processing blocks of interactive lists.

Especially, you **cannot**

- use events such as GET and GET LATE to retrieve data for secondary lists, but must use SELECT statements. You can use the logical database assigned to the executable program (report) only for the basic list. If you want to use logical databases during interactive events, you must call an independent executable program (report) using SUBMIT (see [Calling Programs \[Page 1112\]](#))
- use the event TOP-OF-PAGE to influence the list structure of secondary lists. To layout the page headers of the secondary lists, you must use the event TOP-OF-PAGE DURING LINE-SELECTION (see [Page Headers for Secondary Lists \[Page 1041\]](#)). However, the system processes the event END-OF-PAGE also on secondary lists.

- use separate processing blocks to process further interactive events. A certain user action always triggers the same processing block in your program. You must use control statements (IF, CASE) within the processing block to make sure that the system processes the desired statements. Several system fields help you in this context (see [System Fields for Secondary Lists \[Page 1040\]](#)).

Basic Lists and Secondary Lists

This section gives you an overview on basic and secondary lists.

[Creating the Basic List \[Page 1036\]](#)

[Creating Secondary Lists \[Page 1037\]](#)

[Maintaining Lists \[Page 1039\]](#)

[System Fields for Secondary Lists \[Page 1040\]](#)

[Page Headers for Secondary Lists \[Page 1041\]](#)

[Messages in Lists \[Page 1195\]](#)

Creating the Basic List

An ABAP program places the output data created while processing the events related to data retrieval (START-OF-SELECTION, GET, and so on) into the so-called basic list.

By default, a basic list has a standard page header (see [Standard Page Header \[Page 943\]](#)). If, while creating the basic list, the events TOP-OF-PAGE and END-OF-PAGE occur, the system writes all subsequent output into the page header or page footer of the basic list. After processing all events related to data retrieval, the system displays the basic list on the output screen.

The SY-LSIND system field contains the index of the list currently created. While creating a basic list, SY-LSIND equals 0.

For examples for basic lists, see [Creating Complex Lists \[Page 938\]](#).

Creating Secondary Lists

Creating Secondary Lists

A list is an interactive list if the user interface allows actions that trigger events and if the corresponding interactive event keywords occur in the program (see [Event Control for Interactive Lists \[Page 1033\]](#)). All output statements executed during an interactive list event write their data into a **new** list (list level) with the index SY-LSIND.

Each time an interactive list event occurs, the system automatically increases SY-LSIND by 1.

The system displays this list after processing the entire processing block of the event keyword or after leaving the processing block due to EXIT or CHECK. By default, the new list overlays the previous list completely. If you want to program a window that overlays only part of the previous list, see [Displaying Lists in Dialog Windows \[Page 1070\]](#).

All lists created during an interactive list event are secondary lists.

Each interactive list event creates a new secondary list. With one ABAP program, you can maintain one basic list and up to 20 secondary lists. If the user creates a list on the next level (that is, SY-LSIND increases), the system stores the previous list and displays the new one. Only one list is active, and that is always the most recently created list. To delete existing lists, see [Maintaining Lists \[Page 1039\]](#).

For secondary lists, the system does not display a standard page header. To create page headers for secondary lists, see [Page Headers for Secondary Lists \[Page 1041\]](#).

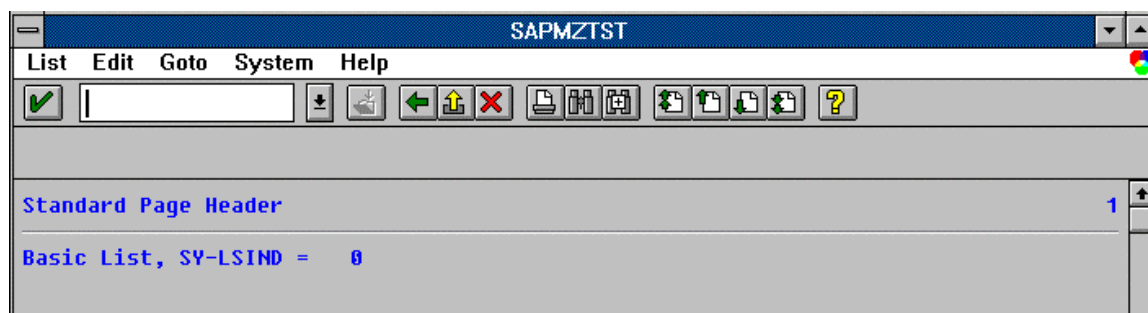
```
REPORT SAPMZTST.
```

```
WRITE: 'Basic List, SY-LSIND =', SY-LSIND.
```

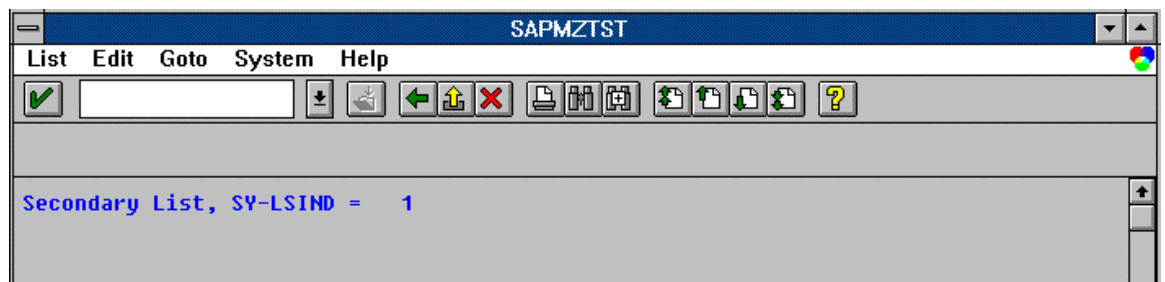
```
AT LINE-SELECTION.
```

```
WRITE: 'Secondary List, SY-LSIND =', SY-LSIND.
```


After executing the program, the system displays the following basic list:



If you compare the toolbar to the standard toolbar of [User Interface of the Standard List \[Page 946\]](#), you find no difference. Nevertheless, the effect of the event keyword AT LINE-SELECTION is that after positioning the cursor on a line, this event is triggered by choosing *Edit* → *Choose*, double-clicking with the mouse, or pressing the function key F2 (see [Allowing Line Selection \[Page 1047\]](#)). Therefore, the list is interactive and after such an action by the user, the output screen changes:



The first secondary list overlays the basic list. This list has no standard page header. It is an interactive list again. By choosing *Choose*, the user can now create up to 20 of these lists. Trying to produce more than 20 lists results in a runtime error.

Using *Back* , the user can return to previous lists. For more information on returning to previous lists, see [Maintaining Lists \[Page 1039\]](#).

Maintaining Lists

Maintaining Lists

To return from a high list level to the next-lower level (SY-LSIND), the user chooses *Back* on a secondary list. The system then releases the currently displayed list and activates the list created one step earlier. The system deletes the contents of the released list.

To explicitly specify the list level into which you want to place output, set the SY-LSIND field. The system accepts only index values which correspond to existing list levels. It then deletes all existing list levels whose index is greater or equal to the index you specify. For example, if you set SY-LSIND to 0, the system deletes **all** secondary lists and overwrites the basic list with the current secondary list.

The system reacts to a manipulation of SY-LSIND only at the end of an event, directly before displaying the secondary list. So, if within the processing block you use statements whose INDEX options access the list with the index SY-LSIND (such as SCROLL), make sure that you manipulate the SY-LSIND field only after processing these statements. The best way to avoid unintentional confusion is to always enter the statement that manipulate SY-LSIND as the last statement of the processing block.

```
REPORT SAPMZTST.
```

```
WRITE: 'Basic List, SY-LSIND =', SY-LSIND.
```

```
AT LINE-SELECTION.
```

```
IF SY-LSIND = 3.
```

```
  SY-LSIND = 0.
```

```
ENDIF.
```

```
WRITE: 'Secondary List, SY-LSIND =', SY-LSIND.
```

After executing the program, the system displays the basic list that contains the following line:

Basic List, SY-LSIND = 0

Due to the AT LINE-SELECTION statement, the list is interactive and enables the user action *Choose* (see [Allowing Line Selection \[Page 1047\]](#)). If the user positions the cursor on the list line and chooses *Choose* to trigger the AT LINE-SELECTION event, the system displays a secondary list that contains the following line:

Secondary List, SY-LSIND = 1

Choosing *Choose* again produces:

Secondary List, SY-LSIND = 2

Goto → *Back* leads to the previous list level. Choosing *Choose* for the third time produces a secondary list that, due to the IF condition, contains the following line:

Secondary List, SY-LSIND = 0

The system deletes the list levels 1 and 2. If you choose *Goto* → *Back* now, you return to the screen from which you started the report. If you choose *Choose*, the system creates a secondary list with index 1. However, the list on level 0 is no longer a basic list (no page header), but a secondary list as well.

System Fields for Secondary Lists

With each interactive event, the system automatically sets the following system fields:

System field	Information
SY-LSIND	Index of the list created during the current event (basic list = 0)
SY-LISTI	Index of the list level from which the event was triggered
SY-LILLI	Absolute number of the line from which the event was triggered
SY-LISEL	Contents of the line from which the event was triggered
SY-CUROW	Position of the line in the window from which the event was triggered (counting starts with 1)
SY-CUCOL	Position of the column in the window from which the event was triggered (counting starts with 2)
SY-CPAGE	Page number of the first displayed page of the list from which the event was triggered
SY-STARO	Number of the first line of the first page displayed of the list from which the event was triggered (counting starts with 1). Possibly, a page header occupies this line.
SY-STACO	Number of the first column displayed in the list from which the event was triggered (counting starts with 1)
SY-UCOMM	Function code that triggered the event
SY-PFKEY	Status of the displayed list

The system fields SY-UCOMM and SY-PFKEY are important if you use self-defined list interfaces (see [Defining Individual User Interfaces \[Page 1051\]](#)).

You can use the information contained in the system fields listed above to structure the secondary lists. For more information and an example, see [Passing Data Automatically \[Page 1077\]](#).

Page Headers for Secondary Lists

Page Headers for Secondary Lists

On secondary lists, the system does not display a standard page header and it does not trigger the event TOP-OF-PAGE. To create page headers for secondary list, you must enhance TOP-OF-PAGE:

Syntax

TOP-OF-PAGE DURING LINE-SELECTION.

The system triggers this event for **each** secondary list. If you want to create different page headers for different list levels, you must program the processing block of this event accordingly, for example by using system fields such as SY-LSIND or SY-PFKEY in control statements (IF, CASE).

When the user scrolls vertically through secondary lists, the page header remains and only the list lines beneath the header scroll.

```
REPORT SAPMZTST.
WRITE 'Basic List'.
AT LINE-SELECTION.
WRITE 'Secondary List'.
TOP-OF-PAGE DURING LINE-SELECTION.
CASE SY-LSIND.
  WHEN 1.
    WRITE 'First Secondary List'.
  WHEN 2.
    WRITE 'Second Secondary List'.
  WHEN OTHERS.
    WRITE: 'Secondary List, Level:', SY-LSIND.
ENDCASE.
ULINE.
```

After the user executes the above program, the system displays a basic list resembling the list from the example in [Creating Secondary Lists \[Page 1037\]](#). The user can choose *Choose* to create secondary lists. The secondary list of list level 1 looks as follows:

First Secondary List

Secondary List

The secondary list of list level 3 looks like this:

Secondary List, Level: 3

Secondary List

Due to the CASE control statement within the TOP-OF-PAGE DURING LINE-SELECTION processing block, the system creates a different page header for each secondary list.

Messages in Lists

Messages in Lists

ABAP allows you to react to incorrect or doubtful user input by displaying messages that influence the program flow depending on how serious the error was. Handling messages is mainly a topic of dialog programming where it is described in detail (see [Handling Errors and Messages \[Page 1172\]](#)). This topic only explains the influence messages have on interactive list processing.

You store and maintain messages in Table T100. Messages are sorted by language, by a two-character ID, and by a three-digit number. You can assign different message types to each message you output. The influence of a message on the program flow depends on the message type.

In your program, use the MESSAGE statement to output messages statically or dynamically and to determine the message type.

Use the MESSAGE-ID option of the REPORT or PROGRAM statement, if you want to use messages of a certain ID statically in your program:

Syntax

REPORT <rep> MESSAGE-ID <id>.

Due to this statement, the report <rep> can use all messages stored in Table T100 under the ID <id>. If you specify message IDs dynamically, you can omit this option.

To specify a message number and type statically, use:

Syntax

MESSAGE <c><num> [WITH <f₁>... <f₄>].

This statement outputs the message stored in Table T100 under number <num>, that has the same ID <id> as in the REPORT statement, as message type <c>.

To specify a message ID, type, and number dynamically at runtime, use:

Syntax

MESSAGE ID <id> TYPE <c> NUMBER <num> [WITH <f₁>... <f₄>].

This statement outputs the message whose ID, number, and type are stored in the fields <id>, <num>, and <c>. For this statement, you do not need the MESSAGE-ID option in the REPORT statement.

A message can have five different types. These message types have the following effects during list processing:

- A (=Abend):
The system displays a message of this message type in a dialog window. After the user confirms the message using ENTER, the system terminates the entire transaction (for example SE38).
- E (=Error) or W (=Warning):
The system displays a message of this message type in the status line. After the user chooses ENTER, the system acts as follows:
 - While creating the basic list, the system terminates the report.

- While creating a secondary list, the system terminates the corresponding processing block and keeps displaying the previous list level.
- I (=Information):
The system displays a message of this message type in a dialog window. After the user chooses ENTER, the system resumes processing at the current program position.
- S (=Success):
The system displays a message of this message type on the output screen in the status line of the currently created list.

Ampersand characters '&' serve as placeholders within a message. If you use the WITH option, the system replaces the placeholders '&' in the message one after the other with the contents of the fields <f_i> and the numbered placeholders '&i' according to the number. To output a '&' character in the message, you must write '&&'.

To create, display, or change messages, simply double-click on the ID <id> or the message number in the ABAP Editor. If the object does not yet exist, the system asks you whether you want to create it.

You can easily include static MESSAGE statements into your program by choosing *Edit* → *Insert statement...* from the ABAP Editor. Select MESSAGE as statement:

 MESSAGE ID HB Typ E Number 100

With this method, you can either copy existing messages or create a new object (ID and number). In addition, the system includes the message text as comment into your program.

Suppose, the following messages are stored in Table T100 under the ID HB:

```
100 Example for Message on Lists
200 Level &1 not allowed
```

The following program uses these messages:

```
REPORT SAPMZTST MESSAGE-ID HB NO STANDARD PAGE HEADING.
```

```
WRITE 'Basic List'.
```

```
MESSAGE S100.
```

```
AT LINE-SELECTION.
```

```
IF SY-LSIND = 1.
```

```
  MESSAGE ID 'HB' TYPE 'I' NUMBER 100.
```

```
ENDIF.
```

```
IF SY-LSIND = 2.
```

```
  MESSAGE E200 WITH SY-LSIND.
```

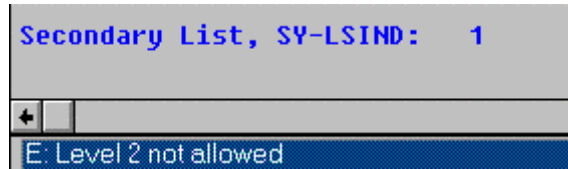
```
ENDIF.
```

```
WRITE: 'Secondary List, SY-LSIND:', SY-LSIND.
```

After executing the program, the system displays the basic list and the success message 100 in the status line. After selecting a line by double-clicking, the event AT LINE-SELECTION occurs. While the system creates the first secondary list, it

Messages in Lists

displays a dialog window with the information message 100. You cannot create the second secondary list, because the message 200 has message type E:



User Interfaces of Interactive Lists

If you want the user to communicate with the system during list display, the list must be interactive. You can define specific interactive possibilities in the status of the list's user interface (GUI). To define the statuses of interfaces in the R/3 system, use the Menu Painter tool. In the Menu Painter, assign function codes to certain interactive functions. After an user action occurs on the completed interface, the ABAP processor checks the function code and, if valid, triggers the corresponding event. From within the program, you can use the SY-UCOMM system field to access the function code.

You can define individual interfaces for your report and assign them in the report to any list level. If you do not specify self-defined interfaces in the report but use at least one of the three interactive event keywords AT LINE-SELECTION, AT PF<nn>, or AT USER-COMMAND in the program, the system automatically uses appropriate predefined standard interfaces. These standard interfaces provide the same functions as the standard list described under [The Standard List \[Page 939\]](#). Depending on the event keyword, they provide additional possibilities for the user to interact with the system.

The two topics below describe the events AT LINE-SELECTION and AT PF<nn> in connection with their predefined standard interfaces.

[Allowing Line Selection \[Page 1047\]](#)

[Allowing Function Key Selection \[Page 1049\]](#)

The AT USER-COMMAND event serves mainly to handle own function codes. In this case, you should create an individual interface with the Menu Painter and define such function codes. The following topic explains how to define individual interfaces and how to use the AT USER-COMMAND event.

[Defining Individual User Interfaces \[Page 1051\]](#)

You can also display lists in dialog windows instead of the fullscreen:

[Displaying Lists in Dialog Windows \[Page 1070\]](#)

And you can trigger interactive events not only from the interface but also from within the program:

[Triggering Events from within the Program \[Page 1074\]](#)

The user interfaces described in this section apply to programs of type 1 (online program) only. The system sets the interface accordingly and uses the list processor. Note that programs for creating transactions use a different environment.

Allowing Line Selection

Allowing Line Selection

To allow the user to select a line from the list, define and write a processing block for the AT LINE-SELECTION event in your program:

Syntax

AT LINE-SELECTION.

<statements>.

If you do not define an individual interface for the list, the system automatically uses a predefined interactive interface. The appearance of this interface is the same as for the standard interface (see [The Standard List \[Page 939\]](#)).

However, the system reacts to the following user actions:

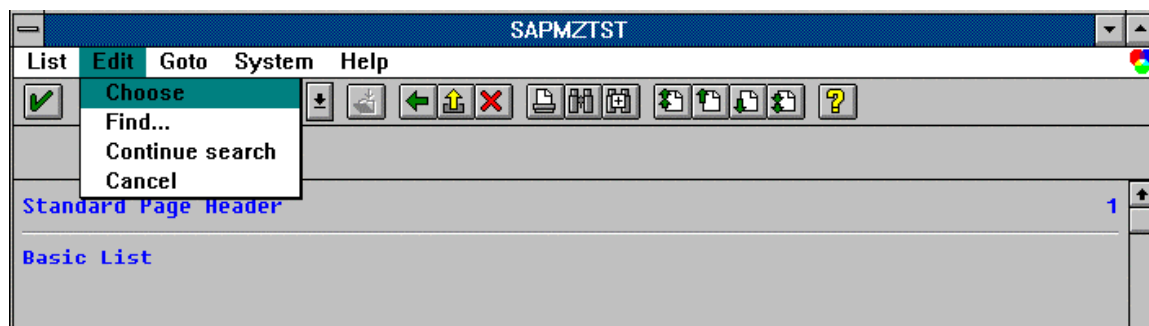
- choosing menu entry *Edit* → *Choose*.
- pressing the function key F2.
- double-clicking on a list line or single-clicking on a hotspot (see [Outputting Fields as Hotspots \[Page 1005\]](#)).

After positioning the cursor on a list line (including header or footer line) and choosing one of the four actions described above, the AT LINE-SELECTION event occurs.

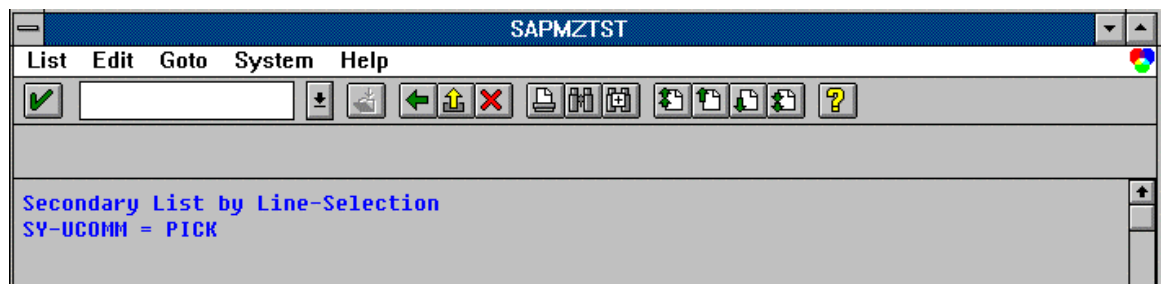
Internally, the function code PICK triggers the AT LINE-SELECTION event. In the predefined interface, *Edit* → *Choose* and F2 are assigned to PICK. Setting the function key F2, automatically activates the functionality of the mouse. For more information, see [Defining Individual User Interfaces \[Page 1051\]](#).

```
REPORT SAPMZTST.
WRITE 'Basic List'.
AT LINE-SELECTION.
WRITE: 'Secondary List by Line-Selection',
      / 'SY-UCOMM =', SY-UCOMM.
```

After executing the program, the system displays this interface:



The user can choose between the actions described above. All secondary lists created by the program look like this:



For each interactive user action, SY-UCOMM returns the value 'PICK'.

Allowing Function Key Selection

Allowing Function Key Selection

To allow the user to select an action by pressing a function key, define and write a processing block for the AT PF<nn> event in your program:

Syntax

AT PF<nn>.

<statements>.

If you do not define an individual interface for the list, the system uses a predefined interactive interface. The difference between standard interface and this predefined interface is that all function keys of the keyboard F<nn> that are not used for predefined system functions, are set to the function codes PF<nn>, where <nn> is a number between 01 and 24. If the user presses one of these keys, the system triggers the corresponding event. The position of the cursor, in this case, is not relevant.

To view a list of the keys predefined with system functions, include the AT PF<nn> statement into your program, display the output list, and press the right mouse button on the list:

Help	F1
Choose	F2
Back	F3
Possible entries	F4
F5	F5
F6	F6
F7	F7
F8	F8
F9	F9
F11	F11
Cancel	F12
F13	F13
F14	F14
Exit	F15
F16	F16
F17	F17
F18	F18
F19	F19
F20	F20
First page	F21
Previous page	F22
Next page	F23
Last page	F24

All keys with a text to the left are used for predefined system functions. These functions have higher priority than self-defined events. The system does not trigger an event AT PF<nn> for keys whose function code is predefined..

Use the AT PF<nn> event for testing purposes only. In your final program version, use the AT USER-COMMAND event together with self-defined interfaces and

individual, meaningful function keys instead. On a self-defined user interface, you can tell the user about the functionality of an action by the text you choose for menu entries or pushbuttons. Using nothing but function keys deprives you of this user-friendly feature.

```
REPORT SAPMZTST.
WRITE 'Basic List'.
AT PF5.
  PERFORM OUT.
AT PF6.
  PERFORM OUT.
AT PF7.
  PERFORM OUT.
AT PF8.
  PERFORM OUT.
FORM OUT.
  WRITE: 'Secondary List by PF-Key Selection',
        / 'SY-LSIND =', SY-LSIND,
        / 'SY-UCOMM =', SY-UCOMM.
ENDFORM.
```

After executing the program, the system displays the basic list. The user can press the function keys F5, F6, F7, and F8 to create secondary lists. If, for example, the 14th key the user presses is F6, the output on the displayed secondary list looks as follows:

```
Secondary List by PF-Key Selection
SY-LSIND = 14
SY-UCOMM = PF06
SY-UCOMM returns the function code PF06.
```

Defining Individual User Interfaces

Defining Individual User Interfaces

You can use individual user interfaces for interactive lists. You can define your own function codes and provide the user with the usual actions to activate such a function by offering entries in the menu bar, pushbuttons, icons, and PF codes. Calling a function assigned to one of your own function codes triggers the AT USER-COMMAND event. In the event's processing block, you use control statements (IF, CASE) and the SY-UCOMM system field to program the desired statements for each function code.

A user interface consists of a status and a title. The status consists of menu bar, function keys, standard toolbar, and application toolbar. To all these elements, you can assign function codes.

The subsequent topics describe

[Defining a Status for Interactive Lists \[Page 1052\]](#)

[Defining Titles for Interactive Lists \[Page 1061\]](#)

[Using Your Own Function Codes in the Program \[Page 1063\]](#)

Defining a Status for Interactive Lists

To define the status of a user interface, use the Menu Painter. The Menu Painter is one of the tools of the ABAP Development Workbench. For a complete description of how to use the Menu Painter, see the documentation [BC ABAP Workbench Tools \[Ext.\]](#).

Furthermore, you can use the Object Browser of the ABAP Development Workbench to display, create, or copy a status. For a description of the Object Browser, see the documentation [BC ABAP Workbench Tools \[Ext.\]](#) as well.

The two topics below only deal with what you have to know about the Menu Painter to create an individual interface for interactive lists:

[Starting the Menu Painter Tool for Interactive Lists \[Page 1053\]](#)

[Using the Menu Painter for Interactive Lists \[Page 1055\]](#)

Starting the Menu Painter Tool for Interactive Lists

Starting the Menu Painter Tool for Interactive Lists

To start the Menu Painter tool, you can either choose *Menu Painter* directly from the initial screen of the ABAP Development Workbench or proceed as described below, which is more appropriate in connection with interactive lists:

1. Set a status for the current list in your program.

To set a status, use the statement:

Syntax

```
SET PF-STATUS <stat>.
```

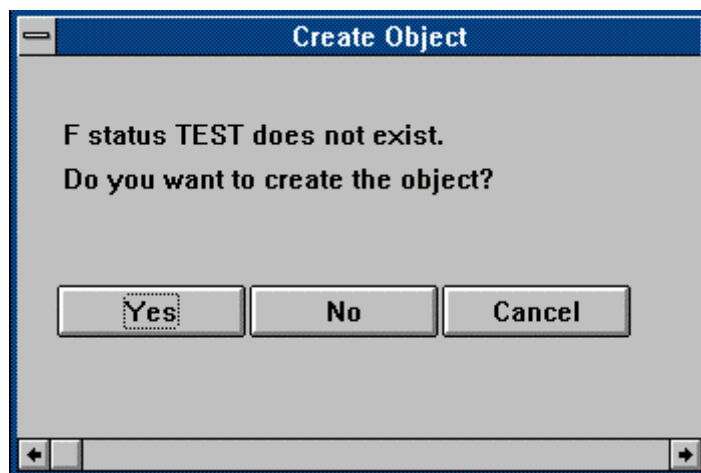
This statement sets a status for the user interface. This status is active until you set another status. The name of <stat> can be up to eight characters long. Choose a meaningful name that describes for what the status is used (e.g. SET PF-STATUS 'MAIN'). For more information on how to set a status, see [Setting a Status \[Page 1064\]](#).

To view a list of existing statuses for your program (if any), use the Object Browser of the ABAP Development Workbench (see the documentation [BC ABAP Workbench Tools \[Ext.\]](#)).

2. In the ABAP Editor, double-click the mouse on <stat>.

If the status <stat> exists, the system leaves the ABAP Editor (allowing you to save your program) and directly takes you to the Menu Painter for the status <stat>. Continue with point 4.

If the status <stat> does not exist, the system displays the following dialog window:



3. Choose Yes.

The system leaves the ABAP Editor (allowing you to save your program) and displays the dialog window *Create Status*, which you can fill in as follows:

Starting the Menu Painter Tool for Interactive Lists

The screenshot shows the 'Create Status' dialog box. The 'Program' field is set to 'SAPHZTST' and the 'Status' field is set to 'TEST'. The 'Maint. language' is 'English'. Under 'Status attributes', the 'Short text' is 'Test for ABAP/4 User's Guide'. In the 'Status type' section, the 'List' radio button is selected. The dialog has 'OK' and 'Cancel' buttons at the bottom left.

For the *Status type* of interactive lists, choose *List* or *List in dialog box*. The system then automatically loads function codes predefined for list processing into the Menu Painter. The system uses the status type you select to support you with the entries required in the Menu Painter. These predefined entries conform to the standards of the SAP Style Guide (see the documentation [BC - SAP Style Guide \[Ext.\]](#)) and thus help you to keep to the standards.

Use the status type *List* for lists displayed on a fullscreen with a complete user interface. Use the status type *List in dialog box* for lists that you want to display in a dialog window without menu bar and standard toolbar (see [Displaying Lists in Dialog Windows \[Page 1070\]](#)).

Choose Enter.

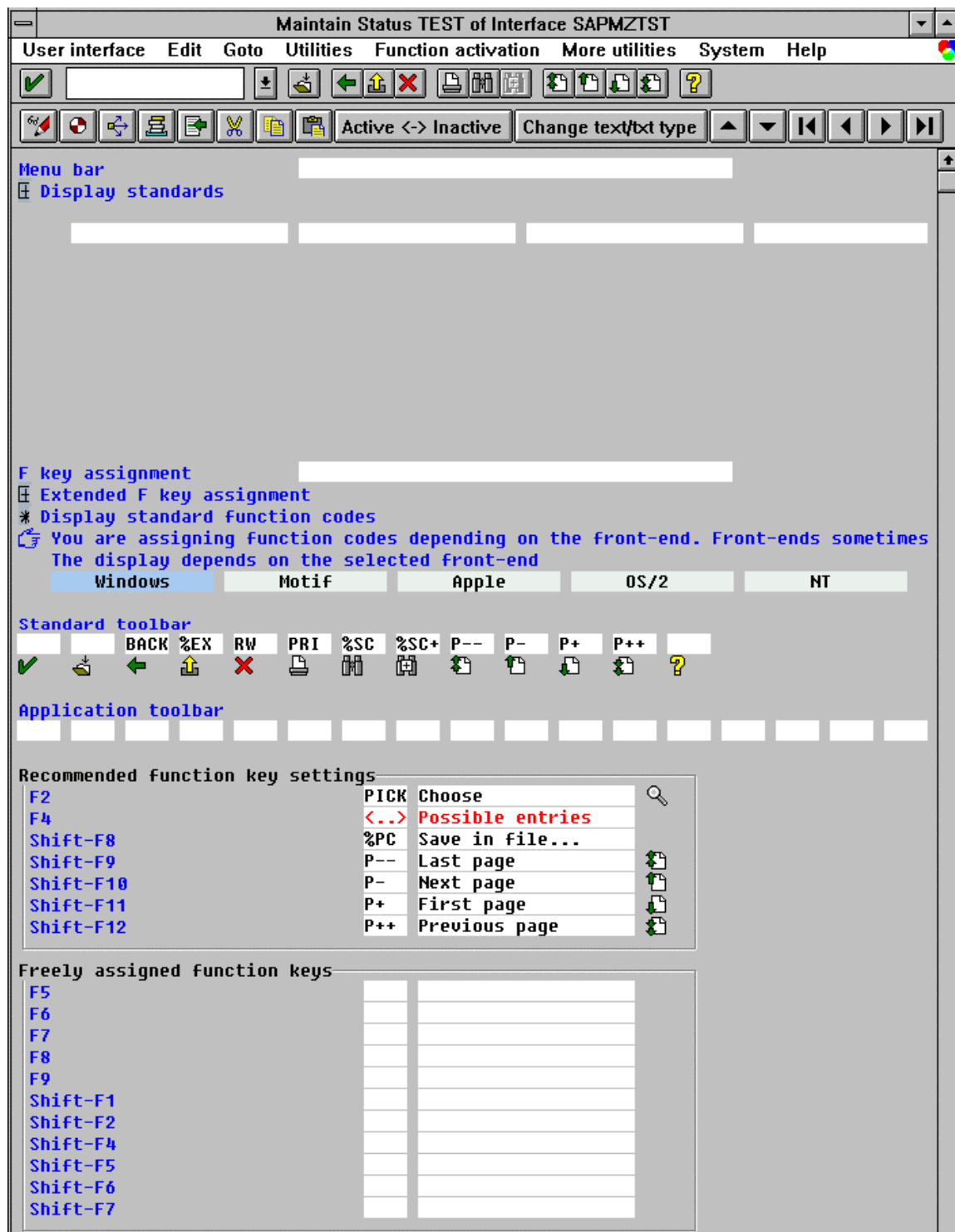
The system displays the Menu Painter for the status.

4. Define the status as described in [Using the Menu Painter for Interactive Lists \[Page 1055\]](#).

Using the Menu Painter for Interactive Lists

Use the Menu Painter to create your own function codes, thus making entries into the status. A status created for a *List* already contains some standard function codes for list processing. Activate *Display standards*. The Menu Painter for interactive lists then looks as follows:

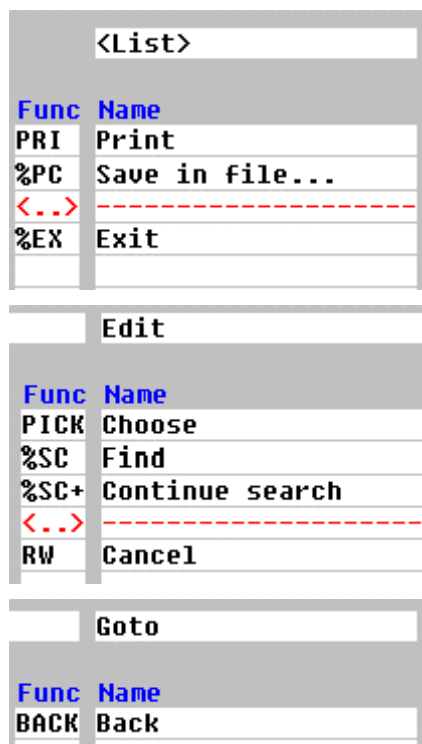
Using the Menu Painter for Interactive Lists














The development environment you select (in this case WINDOWS) does not influence the functionality of the Menu Painter or the interfaces you define with it. You need not create an interface for each environment. The only thing that changes is the display of the function keys, because they may be set differently for the different environments (for example, Ctrl-F12 in WINDOWS is Alt-Shift-F12 in Motif).

Using the Menu Painter for Interactive Lists

The proposed menu bar contains the following list-specific entries:



The following function codes are predefined and assigned to the different elements of the status. Note that this setting is list-specific and would not be displayed if you create a new status for a screen.

Code	Menu	Status element Standard toolbar	Function key	Description
%PC	List		Shift-F8	Write list to file
PRI	List		Ctrl-P	Print displayed list
%EX	List		Shift-F3	Exit processing
PICK	Edit		F2	Event AT LINE-SELECTION
RW	Edit		F12, ESC	Cancel processing
%SC	Edit		Ctrl-F	Search pattern
%SC+	Edit		Ctrl-G	Continue search
BACK	Goto		F3	Back one level
P--			F21	Scroll to first window page
P-			F22	Scroll to previous window page
P+			F23	Scroll to next window page
P++			F24	Scroll to last window page

Using the Menu Painter for Interactive Lists

In addition, the following function codes are predefined, but not set as status functions. You can assign them freely to any empty status element:

Code	Description
PF<nn>	Event AT PF<nn>
PP<n>	Scroll to top of list page <n>
PP- [<n>]	Scroll backward one or <n> list pages, respectively
PP+ [<n>]	Scroll forward one or <n> list pages, respectively
PS<n>	Scroll to column <n>
PS--	Scroll to first column of the list
PS- [<n>]	Scroll left by one or <n> columns, respectively
PS+ [<n>]	Scroll right by one or <n> columns, respectively
PS++	Scroll to last column of the list
PZ<n>	Scroll to line <n>
PL- [<n>]	Scroll backward to first line of the page or by <n> lines
PL+ [<n>]	Scroll forward to last line of the page or by <n> lines
/...	For other system commands

The system (the ABAP processor) directly queries and processes all function codes of the above tables, except PICK and PF<nn>. These function codes do not trigger events, and you cannot use them for the AT USER-COMMAND event. The function code PICK triggers the AT LINE-SELECTION event, whenever the cursor is positioned on a list line. The function codes PF<nn> always trigger the AT PF<nn> event. Therefore, you cannot use them for the AT USER-COMMAND event either.

The function codes described in the above tables are fixed. For any other function, you can define any individual four-character function code. Use meaningful short forms, such as SORT to trigger a sorting process. Since many system-defined functions start with P, you should not use P as the first letter of any of your own function codes.

As far as function keys are concerned, beware of these two special cases:

Function key F2:

Double-clicking the mouse is always equivalent to pressing function key F2. Each function code you assign to F2 is therefore activated by double-clicking. Double-clicking triggers the AT LINE-SELECTION event only if the function code PICK is assigned to function key F2. If you assign your own function code to F2, double-clicking triggers the AT USER-COMMAND event. If you assign a predefined function code to F2, double-clicking triggers the corresponding system action.

Function key F10:

Function key F10 always places the cursor into the menu bar to select a menu function. You cannot assign any other (own or predefined) function code to F10.

The list-specific predefinitions in the Menu Painter are proposals which you can modify according to your requirements. You can

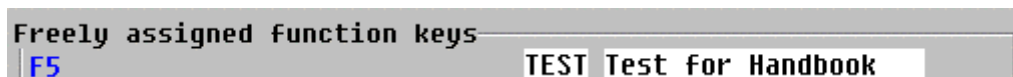
Using the Menu Painter for Interactive Lists

- replace function code PICK with your own function code to prevent the AT LINE-SELECTION event from being triggered in the report. You can then program all reactions to user actions in a single processing block (AT USER-COMMAND).
- delete predefined function codes whose functionality you do not want to support. For example, you may not want the user to print a list directly or to save a list in a file on the presentation server.
- modify the standard key settings. For example, you can assign your own function code to F3 to navigate within the lists according to your requirements, instead of returning one list level (*Back*). This may be important if you keep several lists on the same logical level and therefore do not want to delete the displayed list as would the standard F3 setting. Or you may want to display a warning before leaving a list level (see [Messages in Lists \[Page 1195\]](#)).

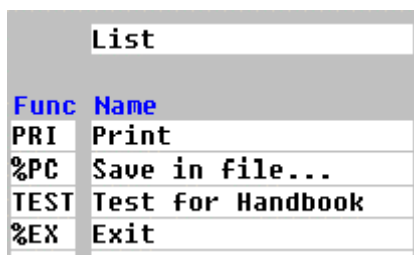
```
REPORT SAPMZTST.
SET PF-STATUS 'TEST'.
WRITE: 'Basic list, SY-LSIND =', SY-LSIND.
AT LINE-SELECTION.
WRITE: 'LINE-SELECTION, SY-LSIND =', SY-LSIND.
AT USER-COMMAND.
CASE SY-UCOMM.
WHEN 'TEST'.
WRITE: 'TEST, SY-LSIND =', SY-LSIND.
ENDCASE.
```

For this report, define a status TEST in the Menu Painter:

1. Assign function code TEST to function key F5.



2. Enter the function code TEST into the menu *List*.

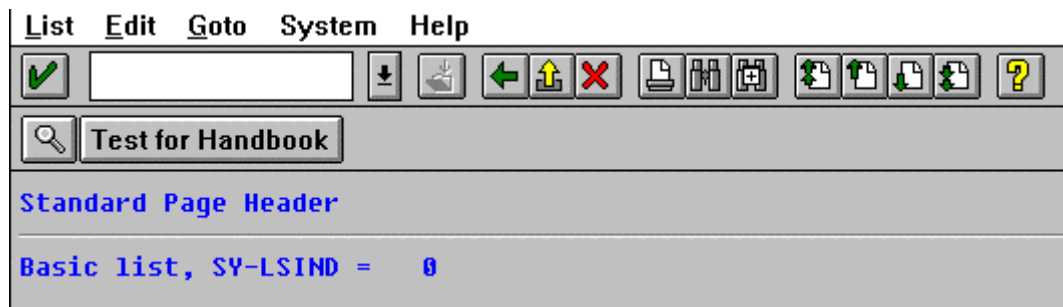


3. Define the function codes PICK and TEST as pushbuttons.



After saving and generating the status, the output screen of the report looks as follows:

Using the Menu Painter for Interactive Lists



The user can trigger the AT USER-COMMAND event either by pressing F5, or by choosing *List* → *Test for Handbook*, or by clicking the pushbutton *Test for Handbook*. The user can trigger the AT LINE-SELECTION event by selecting a line. For more information on the AT USER-COMMAND event, see [Using Your Own Function Codes in the Program \[Page 1063\]](#).

Defining Titles for Interactive Lists

Defining Titles for Interactive Lists

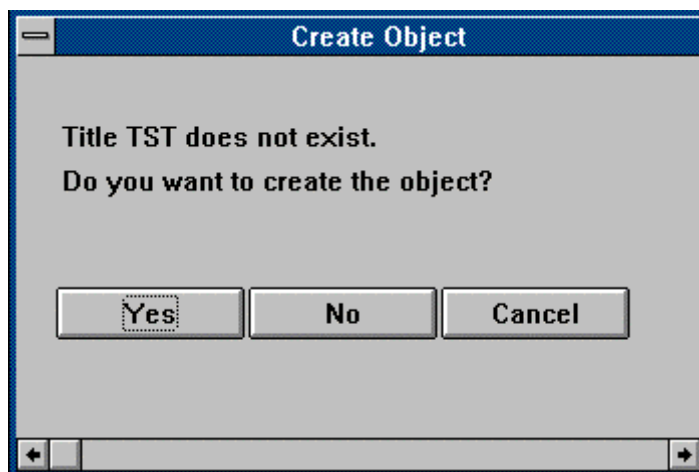
By default, the system uses the program title as the title of the output screen of a report. To choose another title for the output screens, use this statement:

Syntax

SET TITLEBAR <tit> [WITH <g₁>... <g₉>].

This statement sets the title of the user interface of your output list. The title is active for all output screens until you specify another using SET TITLEBAR. <tit> is the name of a title and can be up to three characters long. Titles are connected to the program as are the self-defined statuses. To maintain a title, you can use either the Object Browser of the ABAP Development Workbench or the initial screen of the Menu Painter tool (see the documentation [BC ABAP Workbench Tools \[Ext.\]](#)).

You can maintain the titles of your report from within the ABAP Editor as well. Double-click on <tit> in the above statement. If the title <tit> does not exist, the system displays the following dialog window:



If you choose Yes, the system displays another dialog window where you can enter the desired title. You can specify up to nine different placeholders &1.... &9. You then use the WITH option of the SET TITLEBAR statement to replace these placeholders in the title at runtime with the contents of the corresponding fields <g₁>.... <g₉>. The system also replaces '&' placeholders in succession by the contents of the corresponding <g_i> parameters. To output the wildcard '&' itself, enter '&&'.

Defining Titles for Interactive Lists

The title together with the replacements can be up to 70 characters long. The system ignores superfluous characters and replaces missing variables <g_i> with blanks.

```
REPORT SAPMZTST.
WRITE 'Click me!' HOTSPOT COLOR 5 INVERSE ON.
AT LINE-SELECTION.
SET TITLEBAR 'TST' WITH SY-LSIND.
WRITE 'Click again!' HOTSPOT COLOR 5 INVERSE ON.
```

This program sets a new title for each secondary list. If you define Title TST as in the above figure, for example, the fifth list level has the following title bar:

For more information on hotspots, see [Outputting Fields as Hotspots \[Page 1005\]](#).

Using Your Own Function Codes in the Program

Using Your Own Function Codes in the Program

To use your own function codes in the program, you must set the status of the self-defined interface and program a processing block for the AT USER-COMMAND event.

[Setting a Status \[Page 1064\]](#)

[The AT USER-COMMAND Event \[Page 1067\]](#)

Setting a Status

You set the status with the SET PF-STATUS statement, as mentioned in [Starting the Menu Painter Tool for Interactive Lists \[Page 1053\]](#). The complete syntax is:

Syntax

SET PF-STATUS <stat> [EXCLUDING <f>|<itab>] [IMMEDIATELY].

This statement sets the status <stat> of the current output list. You must define the status <stat> for your program. By setting the status, you enable the user to choose the functions defined in the status. The status is active for all subsequent list levels until you set another status. The SY-PFKEY system field always contains the status of the current list.

Using SET PF-STATUS, you can display different user interfaces for different list levels to provide the user with different functions according to the individual requirements. To set the system-defined list status corresponding to the program, use SET PF-STATUS SPACE.

Use the EXCLUDING option to influence the functionality of a status from within the program. If the individual user interfaces for the different list levels do not differ much, you may define a single all-including status and deactivate unwanted function codes for each list level using EXCLUDING. Specify <f> to deactivate the function code stored in field <f>. Specify <itab> to deactivate all function codes stored in the internal table <itab>. Field <f> and the lines of table <itab> should be of type C, and have length 4.

Use the IMMEDIATELY option to change the status of the currently displayed list (SY-LISTI) within the processing block of the interactive event. Without this option, the system changes the status of the current secondary list (SY-LSIND) that is displayed only at the end of the processing block.

```
REPORT SAPMZTST.

DATA FCODE(4) OCCURS 10 WITH HEADER LINE.
FCODE = 'FC1 '. APPEND FCODE.
FCODE = 'FC2 '. APPEND FCODE.
FCODE = 'FC3 '. APPEND FCODE.
FCODE = 'FC4 '. APPEND FCODE.
FCODE = 'FC5 '. APPEND FCODE.
FCODE = 'PICK'. APPEND FCODE.

SET PF-STATUS 'TEST'.
WRITE: 'PF-Status:', SY-PFKEY.

AT LINE-SELECTION.

IF SY-LSIND = 20.
  SET PF-STATUS 'TEST' EXCLUDING FCODE.
ENDIF.

WRITE: 'Line-Selection, SY-LSIND:', SY-LSIND,
      / '      SY-PFKEY:', SY-PFKEY.

AT USER-COMMAND.
```

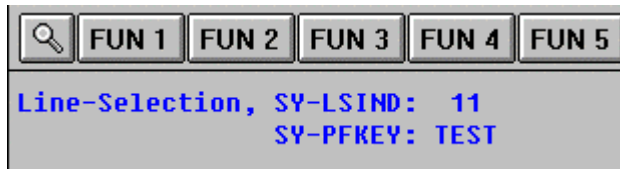
.....

Suppose that the function codes FC1 to FC5 are defined in the status TEST and assigned to pushbuttons:

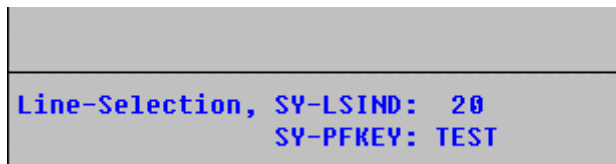
Setting a Status



After executing the program, the user can, for example, create secondary lists by selecting a line. For all secondary lists up to level 20, the user interface TEST is the same as for the basic list:



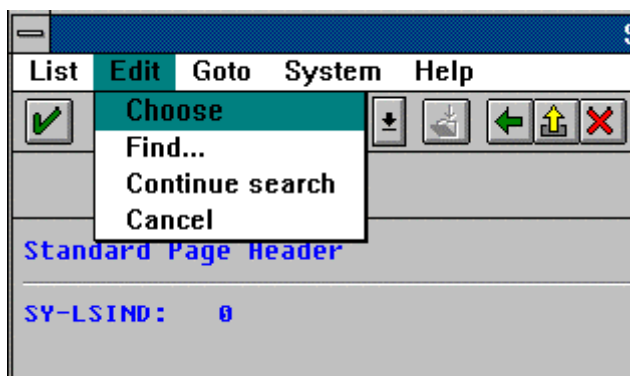
On list level 20, EXCLUDING ITAB deactivates all function codes that create secondary lists.



This prevents the user from causing a program abend by trying to create secondary list number 21.

```
REPORT SAPMZTST.
WRITE: 'SY-LSIND:', SY-LSIND.
AT LINE-SELECTION.
SET PF-STATUS 'TEST' IMMEDIATELY.
```

After executing the program, the output screen shows the basic list and the user interface predefined for line selection (see [Allowing Line Selection \[Page 1047\]](#)):



If the user chooses *Choose*, the user interface changes. However, since the AT LINE-SELECTION processing block does not contain an output statement, the system does not create a secondary list:

	FUN 1	FUN 2	FUN 3	FUN 4	FUN 5
Standard Page Header					
SY-LSIND: 0					

The status TEST is defined as in the previous example.

The AT USER-COMMAND Event

The AT USER-COMMAND Event

To allow your program to react to a user action triggering a self-defined function code, define and program a processing block for the AT USER-COMMAND event.

Syntax

AT USER-COMMAND.

<statements>.

The AT USER-COMMAND event occurs whenever the user selects a self-defined function code from a self-defined user interface. The event does not occur if the user selects function codes predefined for system functions or the function code PICK, which always triggers the AT LINE-SELECTION event. For more information to these function codes, see the tables in [Using the Menu Painter for Interactive Lists \[Page 1055\]](#).

Use the SY-UCOMM system field within the processing block of the AT USER-COMMAND event to distinguish between the different function codes.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING.
```

```
WRITE: 'Basic List',  
      / 'SY-LSIND:', SY-LSIND.
```

```
TOP-OF-PAGE.
```

```
WRITE 'Top-of-Page'.  
ULINE.
```

```
TOP-OF-PAGE DURING LINE-SELECTION.
```

```
CASE SY-PFKEY.  
  WHEN 'TEST'.  
    WRITE 'Self-defined GUI for Function Codes'.  
    ULINE.  
  ENDCASE.
```

```
AT LINE-SELECTION.
```

```
SET PF-STATUS 'TEST' EXCLUDING 'PICK'.  
PERFORM OUT.  
SY-LSIND = SY-LSIND - 1.
```

```
AT USER-COMMAND.
```

```
CASE SY-UCOMM.  
  WHEN 'FC1'.  
    PERFORM OUT.  
    WRITE / 'Button FUN 1 was pressed'.  
  WHEN 'FC2'.  
    PERFORM OUT.  
    WRITE / 'Button FUN 2 was pressed'.  
  WHEN 'FC3'.  
    PERFORM OUT.  
    WRITE / 'Button FUN 3 was pressed'.  
  WHEN 'FC4'.  
    PERFORM OUT.
```

```

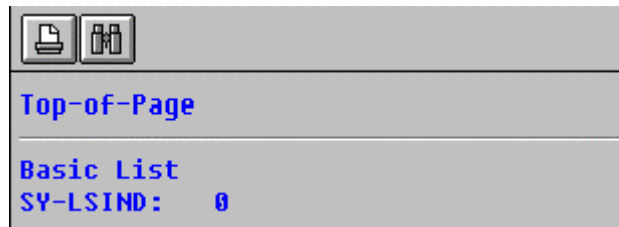
WRITE / 'Button FUN 4 was pressed'.
WHEN 'FC5'.
  PERFORM OUT.
  WRITE / 'Button FUN 5 was pressed'.
ENDCASE.

SY-LSIND = SY-LSIND - 1.

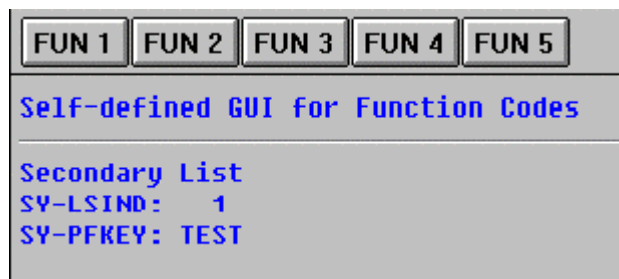
FORM OUT.
  WRITE: 'Secondary List',
        / 'SY-LSIND:', SY-LSIND,
        / 'SY-PFKEY:', SY-PFKEY.
ENDFORM.

```

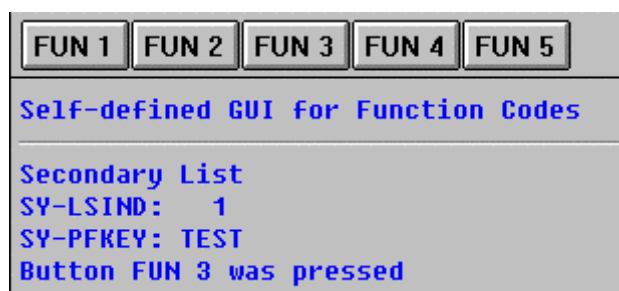
After executing the program, the system displays the following basic list with a self-defined page header:



By double-clicking on a line, the user can trigger the AT LINE-SELECTION event. The system sets status TEST and deactivates function code PICK. Status TEST is the same as the one defined in the examples in [Setting a Status \[Page 1064\]](#). The page header of the secondary list depends on the status. The secondary list created by double-clicking looks as follows:



Here, double-clicking on a line does no longer trigger an event. On the other hand, the user can now use five self-defined pushbuttons from the toolbar. They trigger the AT USER-COMMAND event. Due to the CASE statement, the system reacts differently for different pushbuttons. For example, after clicking on *FUN 3*, the screen looks like this:



The AT USER-COMMAND Event

For each interactive event, the system decreases the SY-LSIND system field by one, thus neutralizing the automatic increase. All secondary lists now have the same level as the basic list. They overwrite the basic list. Note that during the creation of the secondary lists SY-LSIND still contains 1.

Displaying Lists in Dialog Windows

To display a secondary list in a dialog window instead of the fullscreen, use the WINDOW statement:

Syntax

WINDOW STARTING AT <left> <upper> [ENDING AT <right> <lower>].

This statement causes the current list (index SY-LSIND) to be displayed in a dialog window. Use <left> and <upper> to specify column and line of the upper left corner, thereby referring to the output screen of the basic list. If <upper> equals 0, the list appears on a fullscreen. The coordinates of the lower right corner depend on the space required by the secondary list. You can specify the ENDING option, using <right> and <lower> to set the column and line of the lower right corner. The window will not exceed these values. By default, the system uses the values of the lower right corner of the window on which the event occurred.

If the width of the dialog window is smaller than the width of the preceding list, the system creates a horizontal scroll bar on the dialog window. To prevent that, you must adapt the width of the secondary list to the width of the dialog window by using the following statement:

NEW-PAGE LINE-SIZE <width>.

(siehe [Page Width of List Levels \[Page 974\]](#)).

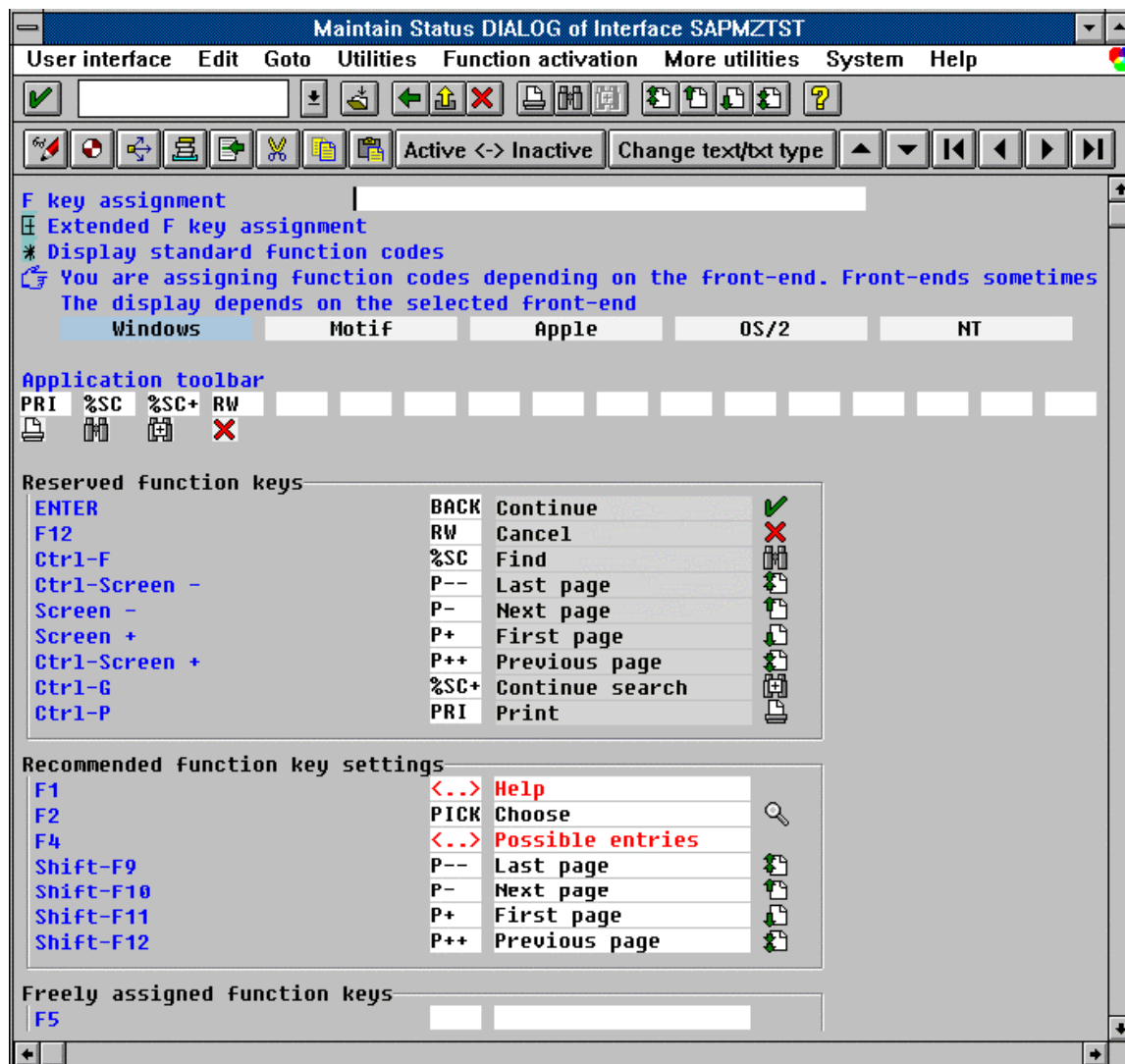
The WINDOW statement takes effect only within the processing block of an interactive event, that is, only for secondary lists. The list functions for lists in dialog windows are the same as for fullscreen lists.

The user interface of a dialog window differs from the fullscreen display. Dialog windows have no menu bar and no standard toolbar. The application toolbar appears at the lower window margin. This is a feature common to all dialog windows in the R/3 system.

If you do not set a self-defined status, but program processing blocks for AT LINE-SELECTION or AT PF<nn> in your report from which you write lists to dialog windows, the system uses predefined interfaces for these windows.

To define a status for a dialog window yourself, proceed as described in [Starting the Menu Painter Tool for Interactive Lists \[Page 1053\]](#). However, under point 3 mark *List in dialog box* as *Status type*. The Menu Painter appears, holding the predefined entries corresponding to this status type:

Displaying Lists in Dialog Windows



The system does not provide a menu bar or a standard toolbar. In the application toolbar, the function codes PRI, %SC, %SC+, and RW are preset to allow the user to print the list, search for patterns, and leave the window. For any other function, use the Menu Painter as described in [Using the Menu Painter for Interactive Lists \[Page 1055\]](#).

```
REPORT SAPMZTST NO STANDARD PAGE HEADING.
```

```
SET PF-STATUS 'BASIC'.
```

```
WRITE 'Select line for a demonstration of windows'.
```

```
AT USER-COMMAND.
```

```
  CASE SY-UCOMM.
```

```
    WHEN 'SELE'.
```

```
      IF SY-LSIND = 1.
```

```
        SET PF-STATUS 'DIALOG'.
```

```
        SET TITLEBAR 'W11'.
```

```
        WINDOW STARTING AT 5 3 ENDING AT 40 10.
```

Displaying Lists in Dialog Windows

```

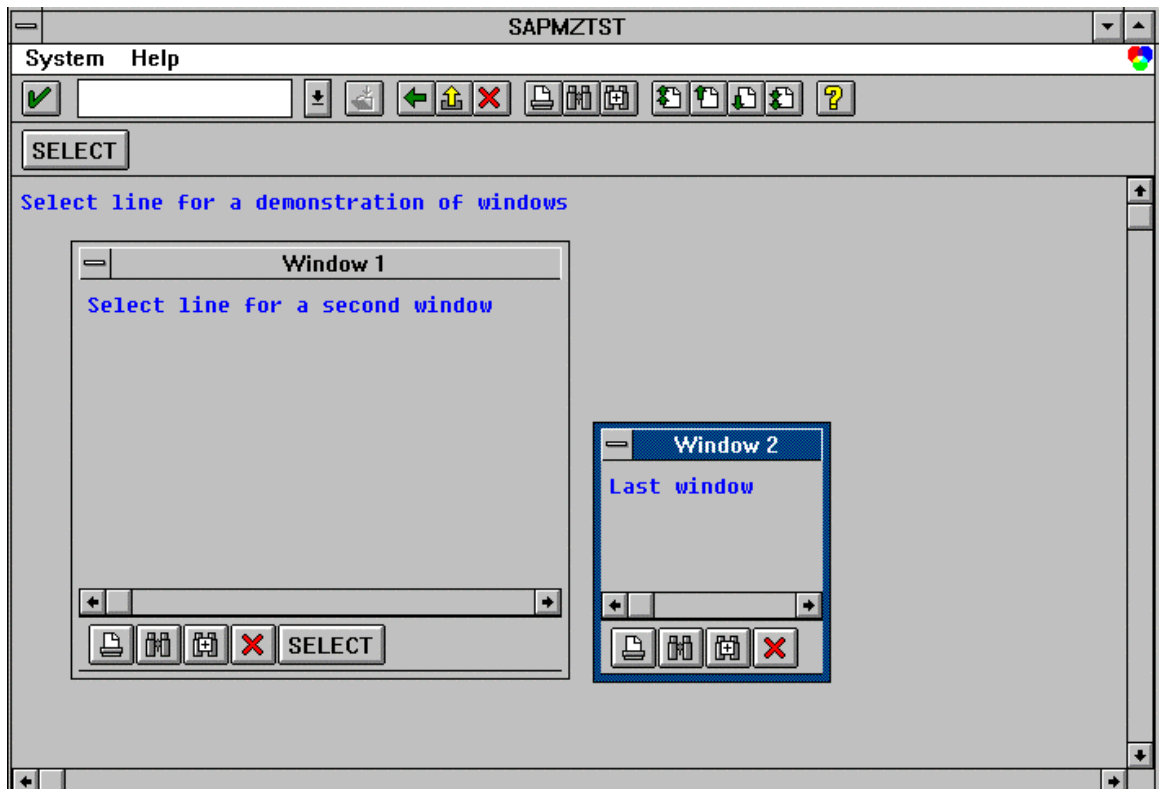
WRITE 'Select line for a second window'.
ELSEIF SY-LSIND = 2.
  SET PF-STATUS 'DIALOG' EXCLUDING 'SELE'.
  SET TITLEBAR 'WI2'.
  WINDOW STARTING AT 45 10 ENDING AT 60 12.
  WRITE 'Last window'.
ENDIF.
ENDCASE.

```

This program sets status BASIC for the basic list. In status BASIC, the function code PICK proposed for function key F2 due to status type *List* is replaced by the self-defined function code SELE (text *SELECT*) which is also assigned to a pushbutton:



For this reason, *SELECT*, F2, and double-clicking the mouse all trigger the event AT USER-COMMAND. In the corresponding processing block, list levels 1 and 2 have the status DIALOG and appear in a dialog window. In status DIALOG, exactly as for status BASIC, the function code PICK proposed for function key F2 due to status type *List* is replaced by function code SELE and assigned to the application toolbar, following RW. After executing the program and selecting a line twice, the output screen appears as follows:



Displaying Lists in Dialog Windows

The title bars W11 and W12 of the dialog windows are self-defined. In the second dialog window, the EXCLUDING option of the SET PF-STATUS statement deactivates function code SELE.

Note the horizontal scroll bars on the dialog windows. The widths of the secondary lists are not adapted to the widths of the dialog windows and, therefore, correspond to the standard width of the basic list.

Triggering Events from within the Program

Instead of letting the user trigger an interactive event by an action on the output screen, you can yourself trigger events from within the program. Use the statement:

Syntax

SET USER-COMMAND <fc>.

This statement takes effect after the current list is completed. Before the system displays the list, it triggers the event that corresponds to the function code stored in <fc>, independent of the applied user interface. This means that for self-defined function codes, the AT USER-COMMAND event occurs, and for the function codes PICK and PF<nn>, the events AT LINE-SELECTION and AT PF<nn> occur.

Function code PICK triggers an event only if the cursor is located on a list line (see example).

For the function codes defined for system functions, the system does not trigger an event but executes the corresponding actions. For example, to display a list together with the dialog window *Search for*, specify '%SC' in <fc>. By specifying the appropriate scroll function code, you can scroll a list to a certain position before displaying it. For a summary of the system-defined function codes, see [Using the Menu Painter for Interactive Lists \[Page 1055\]](#).

If you use several SET USER-COMMAND statements while creating a list, the system executes only the last one.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING.
```

```
SET USER-COMMAND 'MYCO'.
```

```
WRITE 'Basic List'.
```

```
AT USER-COMMAND.
```

```
  CASE SY-UCOMM.
```

```
    WHEN 'MYCO'.
```

```
      WRITE 'Secondary List from USER-COMMAND,'.
```

```
      WRITE: 'SY-LSIND', SY-LSIND.
```

```
      SET USER-COMMAND 'PF05'.
```

```
    ENDCASE.
```

```
AT PF05.
```

```
  WRITE 'Secondary List from PF05,'.
```

```
  WRITE: 'SY-LSIND', SY-LSIND.
```

```
  SET CURSOR LINE 1.
```

```
  SET USER-COMMAND 'PICK'.
```

```
AT LINE-SELECTION.
```

```
  WRITE 'Secondary List from LINE-SELECTION,'.
```

```
  WRITE: 'SY-LSIND', SY-LSIND.
```

```
  SET USER-COMMAND 'PS+10'.
```

This program creates one basic list and three secondary lists. After executing the program, the system immediately displays the third secondary list, scrolled to the right by ten columns:

Triggering Events from within the Program**List from LINE-SELECTION, SY-LSIND 3**

To scroll, the program uses the system-defined function code PS+10. To view the other lists, the user chooses *Back*.

Level 2:

Secondary List from PF05, SY-LSIND 2

Level 1:

Secondary List from USER-COMMAND, SY-LSIND 2

Level 0:

Basic List

Note that in the event AT PF05, the SET CURSOR statement is used to position the cursor on a list line in order to support function code PICK. For more information on how to set the cursor from within a program, see [Setting the Cursor from within the Program \[Page 1100\]](#).

Passing Data from List to Report

To effectively use interactive lists for interactive reporting, it is not sufficient for the program to react to events triggered by user actions on the output list. You must also be able to interpret the lines selected by the user and their contents.

To do this, you use information that the interactive lists pass to the program. ABAP provides three ways of passing data:

- Passing data automatically using system fields
You use automatic data transfer mainly for auxiliary data that you need to better localize user actions.
- Using statements in the program to fetch data
You use program-controlled data transfer to transfer the contents of individual output fields into the processing blocks of the interactive events and to continue processing with these values.
- Passing list attributes
If you do not save certain attributes of a list level, such as number of pages or lines per page, while creating the list, you can later on retrieve these data using the DESCRIBE LIST statement.

The following topics describe these possibilities:

[Passing Data Automatically \[Page 1077\]](#)

[Passing Data by Program Statements \[Page 1082\]](#)

[Passing List Attributes \[Page 1094\]](#)

Passing Data Automatically

Automatic data transfer happens by means of the system fields that are filled by the system for each interactive event. For an overview of the relevant system fields, see [System Fields for Secondary Lists \[Page 1040\]](#).

The subsequent topics describe

[Data from System Fields of Interactive Lists \[Page 1078\]](#)

[Using SY-LISEL \[Page 1080\]](#)

Data from System Fields of Interactive Lists

From system fields, you retrieve the following information: the index of a list, the position of the list in the output window, and the location of the cursor. The only system field that contains the contents of the selected line is SY-LISEL.

The example below demonstrates how the system fills these system fields during interactive events.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING
      LINE-COUNT 12 LINE-SIZE 40.

DATA: L TYPE I, T TYPE C.

DO 100 TIMES.
  WRITE: / 'Loop Pass:', SY-INDEX.
ENDDO.

TOP-OF-PAGE.

WRITE: 'Basic List, Page', SY-PAGNO.
      ULINE.

TOP-OF-PAGE DURING LINE-SELECTION.

WRITE 'Secondary List'.
      ULINE.

AT LINE-SELECTION.

DESCRIBE FIELD SY-LISEL LENGTH L TYPE T.

WRITE: 'SY-LSIND:', SY-LSIND,
      / 'SY-LISTI:', SY-LISTI,
      / 'SY-LILLI:', SY-LILLI,
      / 'SY-CUROW:', SY-CUROW,
      / 'SY-CUCOL:', SY-CUCOL,
      / 'SY-CPAGE:', SY-CPAGE,
      / 'SY-STARO:', SY-STARO,
      / 'SY-LISEL:', 'Length =', L, 'Type =', T,
      / SY-LISEL.
```

This program creates a list of ten pages. After executing the program, the user should scroll the list and position the cursor as shown below:

Basic List, Page		10
<hr/>		
Loop Pass:		97
Loop Pass:		98
Loop Pass:		99
Loop Pass:		100

After choosing *Select* or F2, the secondary list looks like this:

Data from System Fields of Interactive Lists

```
Secondary List
SY-LSIND:    1
SY-LISTI:    0
SY-LILLI:    117
SY-CUROW:    3
SY-CUCOL:    5
SY-CPAGE:    10
SY-STARO:    7
SY-LISEL: Length =      255  Type = C
Loop Pass:   97
```

SY-LSIND is the index of the current list, SY-LISTI is the index of the previous list. SY-LILLI is the number of the selected line in the absolute list (nine times twelve lines for the previous pages plus nine lines on the current page), SY-CUROW is the position of the selected line on the screen. SY-CUCOL is the position of the cursor in the window. This position exceeds the corresponding unscrolled list column by one. SY-CPAGE is the currently displayed page of the list. SY-STARO is the number of the topmost actual list line displayed on the current page. This does not include the page header. SY-CPAGE and SY-STARO do not depend on the cursor position. For SY-LISEL, the program displays length, data type, and contents. The length of SY-LISEL is always 255, independent of the list's width.

Using SY-LISEL

The SY-LISEL system field is a field of type C with a length of 255 characters. It contains the selected line as one single character string, thus making it difficult for you to retrieve the values of individual fields. To process certain parts of SY-LISEL, you must specify the corresponding offsets (see [Specifying Offset Values for Data Objects \[Page 216\]](#)).

To transfer values from fields, do not use SY-LISEL for the reason specified above. The methods described in [Passing Data by Program Statements \[Page 1082\]](#) are much more suitable. However, to layout the header lines of secondary lists or to check whether the selected line is neither blank nor filled with underscores, SY-LISEL would provide a solution.

```
PROGRAM SAPMZTST NO STANDARD PAGE HEADING.

DATA NUM TYPE I.

SKIP.
WRITE 'List of Quadratic Numbers between One and Hundred'.
SKIP.
WRITE 'List of Cubic Numbers between One and Hundred'.

TOP-OF-PAGE.

WRITE 'Choose a line!'.
ULINE.

TOP-OF-PAGE DURING LINE-SELECTION.

WRITE SY-LISEL.
ULINE.

AT LINE-SELECTION.
IF SY-LISEL(4) = 'List'.
CASE SY-LILLI.
WHEN 4.
DO 100 TIMES.
NUM = SY-INDEX ** 2.
WRITE: / SY-INDEX, NUM.
ENDDO.
WHEN 6.
DO 100 TIMES.
NUM = SY-INDEX ** 3.
WRITE: / SY-INDEX, NUM.
ENDDO.
ENDCASE.
ENDIF.
```

After executing the program, the system displays the following output screen:

Using SY-LISEL

Choose a line?

List of Quadratic Numbers between One and Hundred

List of Cubic Numbers between One and Hundred

If the user selects, for example, the upper line, the following secondary list appears.

List of Quadratic Numbers between One and Hundred

1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64

When selecting the lower line on the basic list, the list of cubic numbers appears.

The contents of the selected line becomes the header of the secondary list. The IF statement uses the first four characters of SY-LISEL to make sure that only valid lines are selected. Due to the logical conditions of the processing block following AT LINE-SELECTION, a line selection on a secondary list does not produce any further output.

This example also emphasizes the concept of using condensed information in the basic list to display the relevant details in the secondary lists.

Passing Data by Program Statements

To pass individual output fields or additional information from a line to the corresponding processing block during an interactive event, use these statements:

- HIDE

The HIDE statement is one of the fundamental statements for interactive reporting. Using the HIDE technique, you can at the moment you create a list level define, which information later to pass to the subsequent secondary lists.

- READ LINE

Use the statements READ LINE and READ CURRENT LINE to explicitly read data from the lines of existing list levels. These statements are tightly connected to the HIDE technique.

- GET CURSOR

Use the statements GET CURSOR FIELD and GET CURSOR LINE to pass the output field or output line on which the cursor was positioned during the interactive event to the processing block.

The following topics describe these statements:

[The HIDE Technique \[Page 1083\]](#)

[Reading Lines from Lists \[Page 1087\]](#)

[Reading Lists at the Cursor Position \[Page 1091\]](#)

The HIDE Technique

The HIDE Technique

You use the HIDE technique while creating a list level to store line-specific information for later use.

Syntax

HIDE <f>.

This statement stores the contents of variable <f> in relation to the current output line (system field SY-LINNO) internally in the so-called HIDE area. The variable <f> must not necessarily appear on the current line.

To improve the readability of your program, place the HIDE statement always directly after the output statement for the variable <f> or after the last output statement for the current line. To hide several variables, concatenate several HIDE statements (see [Concatenating Similar Statements \[Page 94\]](#)).

As soon as the user selects a line for which you stored HIDE fields, the system fills the variables in the program with the values stored. A line can be selected

- by an interactive event.
For each interactive event, the HIDE fields of the line on which the cursor is positioned during the event are filled with the stored values.
- by the READ LINE statement.
See [Reading Lines from Lists \[Page 1087\]](#).

You can think of the HIDE area as a table, in which the system stores the names and values of all HIDE fields for each list and line number. As soon as they are needed, the system reads the values from the table.

The example below presents some of the essential features of interactive reporting. The basic list contains summarized information. By means of the HIDE technique, each secondary list contains more details.

```
The following program is connected to the logical database F1S.
PROGRAM SAPMZTST NO STANDARD PAGE HEADING.
TABLES: SPFLI, SBOOK.
DATA: NUM TYPE I,
      DAT TYPE D.
START-OF-SELECTION.
  NUM = 0.
  SET PF-STATUS 'FLIGHT'.
  GET SPFLI.
  NUM = NUM + 1.
  WRITE: / SPFLI-CARRID, SPFLI-CONNID,
         SPFLI-CITYFROM, SPFLI-CITYTO.
  HIDE: SPFLI-CARRID, SPFLI-CONNID, NUM.
```

```

END-OF-SELECTION.

CLEAR NUM.

TOP-OF-PAGE.

WRITE 'List of Flights'.
ULINE.
WRITE 'CA CONN FROM      TO'.
ULINE.

TOP-OF-PAGE DURING LINE-SELECTION.

CASE SY-PFKEY.
  WHEN 'BOOKING'.
    WRITE SY-LISEL.
    ULINE.
  WHEN 'WIND'.
    WRITE: 'Booking', SBOOK-BOOKID,
           / 'Date ', SBOOK-FLDATE.
    ULINE.
ENDCASE.

AT USER-COMMAND.

CASE SY-UCOMM.
  WHEN 'SELE'.
    IF NUM NE 0.
      SET PF-STATUS 'BOOKING'.
      CLEAR DAT.
      SELECT * FROM SBOOK WHERE CARRID = SPFLI-CARRID
             AND CONNID = SPFLI-CONNID.
      IF SBOOK-FLDATE NE DAT.
        DAT = SBOOK-FLDATE.
        SKIP.
        WRITE / SBOOK-FLDATE.
        POSITION 16.
      ELSE.
        NEW-LINE.
        POSITION 16.
      ENDIF.
      WRITE SBOOK-BOOKID.
      HIDE: SBOOK-BOOKID, SBOOK-FLDATE, SBOOK-CUSTTYPE,
            SBOOK-SMOKER, SBOOK-LUGGWEIGHT, SBOOK-CLASS.
    ENDSELECT.
    IF SY-SUBRC NE 0.
      WRITE / 'No bookings for this flight'.
    ENDIF.
    NUM = 0.
    CLEAR SBOOK-BOOKID.
  ENDIF.
  WHEN 'INFO'.
    IF NOT SBOOK-BOOKID IS INITIAL.
      SET PF-STATUS 'WIND'.
      SET TITLEBAR 'BKI'.
      WINDOW STARTING AT 30 5 ENDING AT 60 10.
      WRITE: 'Customer type :', SBOOK-CUSTTYPE,

```

The HIDE Technique

```

      / 'Smoker      :', SBOOK-SMOKER,
      / 'Luggage weight :', SBOOK-LUGGWEIGHT,
      / 'Class      :', SBOOK-CLASS.
    ENDIF.
  ENDCASE.

```

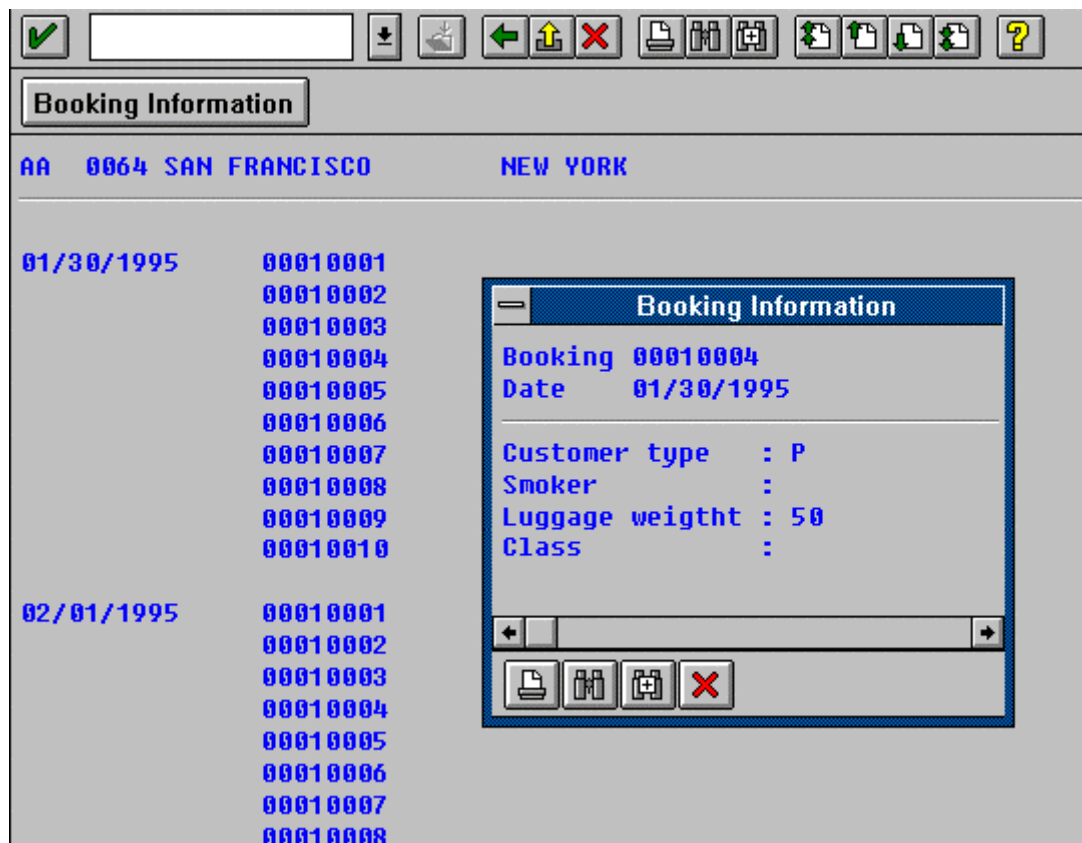
At the event START-OF-SELECTION, the system sets the status FLIGHT for the basic list. In status FLIGHT, function code SELE (text *SELECT*) is assigned to function key F2 and to a pushbutton. Thus, the user actions of double-clicking the mouse, pressing F2, and clicking on pushbutton *SELECT* all trigger the AT USER-COMMAND event.

The basic list appears as below:

SELECT			
List of Flights			
CA	CONN	FROM	TO
AA	0017	NEW YORK	SAN FRANCISCO
AA	0026	FRANKFURT	NEW YORK
AA	0064	SAN FRANCISCO	NEW YORK
DL	1699	NEW YORK	SAN FRANCISCO
DL	1984	SAN FRANCISCO	NEW YORK
LH	0400	FRANKFURT	NEW YORK
LH	0402	FRANKFURT	NEW YORK
LH	0454	FRANKFURT	SAN FRANCISCO
LH	0455	SAN FRANCISCO	FRANKFURT
LH	2402	FRANKFURT	BERLIN
LH	2407	BERLIN	FRANKFURT
LH	2415	BERLIN	FRANKFURT
LH	2436	FRANKFURT	BERLIN
LH	2462	FRANKFURT	BERLIN
LH	2463	BERLIN	FRANKFURT
LH	3577	ROM	FRANKFURT
UA	0007	NEW YORK	SAN FRANCISCO
UA	0941	FRANKFURT	SAN FRANCISCO
UA	3504	SAN FRANCISCO	FRANKFURT

The three fields SPFLI-CARRID, SPFLI-CONNID, and NUM are stored in the HIDE area while creating the basic list. After selecting a line, the system displays the secondary list defined in the AT USER-COMMAND event for function code SELE. In the AT USER-COMMAND event, the system refills all fields of the selected line that were stored in the HIDE area. You use NUM to check whether the user selected a line from the actual list. The secondary list has status BOOKING, where F2 is assigned to function code INFO (text: *Booking Information*). The secondary list presents data that the program selected by means of the HIDE fields of the basic list. For each list line displayed, the system stores additional information in the HIDE area.

If the user selects a line of the secondary list, the system displays the "hidden" information in a dialog window with the status WIND. For the status WIND, the proposals of the status type *List in dialog box* have been accepted. The program uses SBOOK-BOOKID to check whether the user selected a valid line. The first secondary list together with another secondary list created from there looks, for example, like this:



The program itself sets all page headers and the title bar of the dialog window.

Reading Lines from Lists

The system internally stores all lists created in succession by one program. Therefore, you can access any list from within the program that was created during the current dialog session and has not been deleted by returning to a previous list level (see [Maintaining Lists \[Page 1039\]](#)). To read lines, use the statements READ LINE and READ CURRENT LINE.

To read a line from a list after an interactive list event occurred, use the READ LINE statement:

Syntax

```
READ LINE <lin> [INDEX <idx>]
      [FIELD VALUE <f1> [INTO <g1>]... <fn> [INTO <gn>]]
      [OF CURRENT PAGE|OF PAGE <p>].
```

The statement without any options stores the contents of line <lin> from the list on which the event was triggered (index SY-LILLI) in the SY-LISEL system field and fills all HIDE information stored for this line back into the corresponding fields (see [The HIDE Technique \[Page 1083\]](#)). As far as SY-LISEL and the HIDE area are concerned, READ LINE has the same effect as an interactive line selection.

If the selected line <lin> exists, the system sets SY-SUBRC to 0, otherwise to 4.

The options have the following effects:

- INDEX <idx>
The system reads the information for line <lin> from the list of level <idx>.
- FIELD VALUE <f₁> [INTO <g₁>]... <f_n> [INTO <g_n>]
The system interprets the output values of the variables <f_i> in line <lin> as character strings and places them either into the same fields <f_i> or, when using INTO, into the fields <g_i>. When refilling the fields, the system applies the conversion rules (see [Source Type Character \[Page 220\]](#)).

Fields that do not appear in a line, do not affect the target field. If a field appears several times in a line, the system uses only the first one.

The system transports the field contents using the output format, that is, including all formatting characters. This may lead to inconvertibilities, such as converting editing characters to decimal characters and others.

You mainly use this option to process user entries in list fields that accept input, since you cannot use the HIDE technique in this case.
- OF CURRENT PAGE
With this option, <lin> is not the number of the line of the entire list, but the number of the line of the currently displayed page of the addressed list. The system field SY-CPAGE stores the corresponding page number.
- OF PAGE <p>
With this option, <lin> is not the number of the line of the entire list, but the number of a line on page <p> of the addressed list.

Depending on the contents of the line read, you may want to fill fields again from the same line. To do this, you use the READ CURRENT LINE statement in your program. This statement reads a line twice in succession:

Syntax

READ CURRENT LINE [FIELD VALUE <f₁> [INTO <g₁>]...].

This statement reads a line read before by an interactive event (F2) or by READ LINE. The FIELD VALUE option is the same as for READ LINE.

The following program is connected to the logical database F1S.

REPORT SAPMZTST NO STANDARD PAGE HEADING.

TABLES: SFLIGHT.

DATA: BOX, LINES TYPE I, FREE TYPE I.

START-OF-SELECTION.

SET PF-STATUS 'CHECK'.

GET SFLIGHT.

WRITE: BOX AS CHECKBOX, SFLIGHT-FLDATE.

HIDE: SFLIGHT-FLDATE, SFLIGHT-CARRID, SFLIGHT-CONNID,
SFLIGHT-SEATSMAX, SFLIGHT-SEATSOCC.

END-OF-SELECTION.

LINES = SY-LINNO - 1.

TOP-OF-PAGE.

WRITE: 'List of Flights from',
(12) CITY_FR, 'to', CITY_TO.
ULINE.

TOP-OF-PAGE DURING LINE-SELECTION.

WRITE: 'Date:', SFLIGHT-FLDATE.
ULINE.

AT USER-COMMAND.

CASE SY-UCOMM.

WHEN 'READ'.

BOX = SPACE.

SET PF-STATUS 'CHECK' EXCLUDING 'READ'.

DO LINES TIMES.

READ LINE SY-INDEX FIELD VALUE BOX.

IF BOX = 'X'.

FREE = SFLIGHT-SEATSMAX - SFLIGHT-SEATSOCC.

IF FREE > 0.

NEW-PAGE.

WRITE: 'Company:', SFLIGHT-CARRID,
'Connection: ', SFLIGHT-CONNID,
/ 'Number of free seats:', FREE.

ENDIF.

ENDIF.

Reading Lines from Lists

```
ENDDO.
ENDCASE.
```

If after executing the report, the user fills in the departure and arrival airports on the selection screen as follows

From	FRANKFURT
To	BERLIN

the system displays an output screen on which the user may mark checkboxes like this:

List of Flights from FRANKFURT to BERLIN

<input checked="" type="checkbox"/>	1995/01/30
<input type="checkbox"/>	1995/02/01
<input type="checkbox"/>	1995/06/01
<input type="checkbox"/>	1995/06/04
<input checked="" type="checkbox"/>	1995/01/20
<input checked="" type="checkbox"/>	1995/01/22
<input type="checkbox"/>	1995/05/22
<input type="checkbox"/>	1995/05/25
<input checked="" type="checkbox"/>	1995/01/30
<input type="checkbox"/>	1995/02/01
<input type="checkbox"/>	1995/06/01
<input type="checkbox"/>	1995/06/04

The self-defined page header uses the parameters CITY_FR and CITY_TO. They are defined in the logical database F1S. How to find out the names of such parameters using F1 and *Technical Information*, is described in the example of the topic [INITIALIZATION \[Page 1213\]](#).

The program uses the status CHECK, where the self-defined function code READ (text *Read Lines*) is assigned to function key F5 and to a pushbutton in the application toolbar. The corresponding user action triggers the AT USER-COMMAND event. The system now reads all lines of the basic list using READ LINE in a DO loop. For each line, it fills the corresponding fields with all values previously stored in the HIDE area. By means of the FIELD VALUE option, the system in addition reads the checkbox BOX. If seats are still free, the system writes the company, the connection, and the number of free seats into a secondary list for each line marked in the checkbox.

Date: 1995/01/30		
Company: LH	Connection:	2402
Number of free seats:		370
Date: 1995/01/20		
Company: LH	Connection:	2436
Number of free seats:		270
Date: 1995/01/22		
Company: LH	Connection:	2436
Number of free seats:		260
Date: 1995/01/30		
Company: LH	Connection:	2462
Number of free seats:		210

The system starts a new page with an individual page header for each output. On the secondary list, no self-defined user action is supported, since the EXCLUDING option of the SET PF-STATUS statement deactivates the function code READ.

After returning from the secondary list, the checkboxes are still filled. The example in [Modifying Field Formatting \[Page 1110\]](#) shows how to clear them.

Reading Lists at the Cursor Position

Reading Lists at the Cursor Position

To retrieve information on the cursor position during an interactive event, use the GET CURSOR statement to refer to either the field or the line.

For information on the field, use this syntax:

Syntax

```
GET CURSOR FIELD <f> [OFFSET <off>] [LINE <lin>]  
[VALUE <val>] [LENGTH <len>].
```

This statement transfers the name of the field on which the cursor is positioned during a user action into the variable <f>. If the cursor is on a field, the system sets SY-SUBRC to 0, otherwise to 4.

The system transports the names of globale variables, constants, field symbols, or reference parameters of subroutines. For literals, local fields, and VALUE parameters of subroutines, the system sets SY-SUBRC to 0, but transfers SPACE as the name.

The options have the following effects:

- **OFFSET <off>**
The field <off> contains the position of the cursor within the field. If the cursor is on the first column, <off> = 0.
- **LINE <lin>**
The field <lin> contains the number of the list line on which the cursor is positioned (SY-LILLI).
- **VALUE <val>**
The field <val> contains the character string output representation of the field on which the cursor is positioned. The representation includes formatting characters.
- **LENGTH <len>**
The field <len> contains the output length of the field on which the cursor is positioned.

For information on the line, use this syntax:

Syntax

```
GET CURSOR LINE <lin> [OFFSET <off>] [VALUE <val>] [LENGTH <len>].
```

This statement transfers the number of the line on which the cursor is positioned during a user action into the variable <lin>. If the cursor is on a list line, the system sets SY-SUBRC to 0, otherwise to 4. You can use this statement to prevent the user from selecting invalid lines.

The options have the following effects:

- **OFFSET <off>**
The field <off> contains the position of the cursor within the list line. If the cursor is on the first column, <off> = 0.
- **VALUE <val>**

Reading Lists at the Cursor Position

The field <val> contains the character string output representation of the line on which the cursor is positioned. The representation includes formatting characters.

- LENGTH <len>

The field <len> contains the output length of the line on which the cursor is positioned.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING LINE-SIZE 40.
```

```
DATA: HOTSPOT(10) VALUE 'Click me!',
      F(10), OFF TYPE I, LIN TYPE I, VAL(40), LEN TYPE I.
```

```
FIELD-SYMBOLS <FS>.
```

```
ASSIGN HOTSPOT TO <FS>.
```

```
WRITE 'Demonstration of GET CURSOR statement'.
```

```
SKIP TO LINE 4.
```

```
POSITION 20.
```

```
WRITE <FS> HOTSPOT COLOR 5 INVERSE ON.
```

```
AT LINE-SELECTION.
```

```
WINDOW STARTING AT 5 6 ENDING AT 45 20.
```

```
GET CURSOR FIELD F OFFSET OFF
```

```
  LINE LIN VALUE VAL LENGTH LEN.
```

```
WRITE: 'Result of GET CURSOR FIELD: '.
```

```
ULINE AT /(28).
```

```
WRITE: / 'Field: ', F,
```

```
      / 'Offset:', OFF,
```

```
      / 'Line: ', LIN,
```

```
      / 'Value: ', (10) VAL,
```

```
      / 'Length:', LEN.
```

```
SKIP.
```

```
GET CURSOR LINE LIN OFFSET OFF VALUE VAL LENGTH LEN.
```

```
WRITE: 'Result of GET CURSOR LINE: '.
```

```
ULINE AT /(27).
```

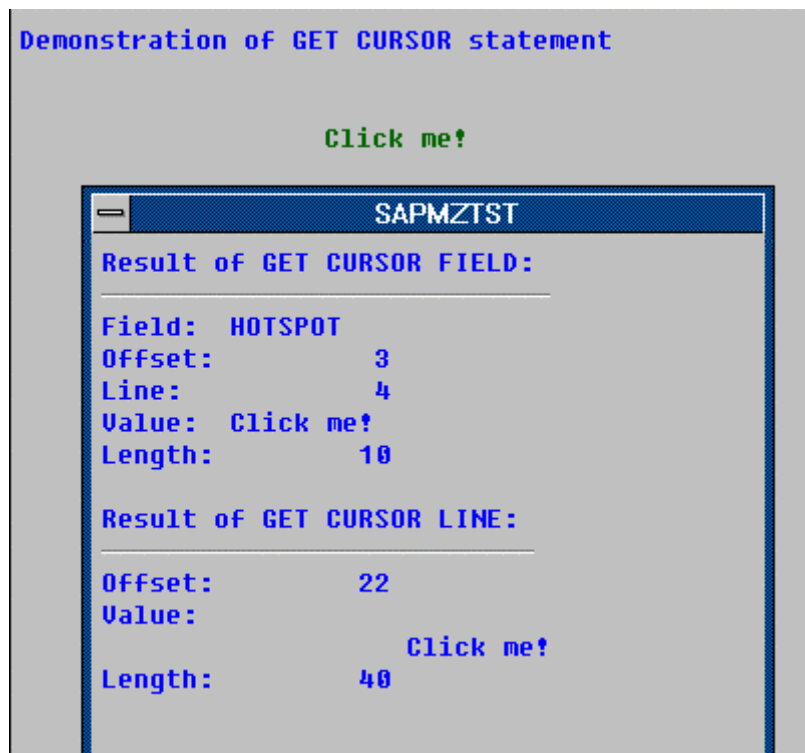
```
WRITE: / 'Offset:', OFF,
```

```
      / 'Value: ', VAL,
```

```
      / 'Length:', LEN.
```

In this program, the HOTSPOT field is assigned to the field symbol <FS> and displayed as hotspot on the output screen. If the user positions the cursor on the letter 'c' of the hotspot and selects the line, the following dialog window appears:

Reading Lists at the Cursor Position



The system outputs the results of the GET CURSOR statements at the AT LINE-SELECTION event. Note that after GET CURSOR FIELD, the name of the field assigned to the field symbol <FS> is stored in F, and not the name of the field symbol.

Passing List Attributes

If, while processing list levels, you need to know attributes that you forgot to store in variables during list creation, or if you use list levels created by another report, use the DESCRIBE LIST statement.

To retrieve the number of lines or pages of a list, use:

Syntax

DESCRIBE LIST NUMBER OF LINES|PAGES <n> [INDEX <idx>].

This statement writes the number of lines or pages of the list level <idx> into the variable <n>. If a list with index <idx> does not exist, the system sets SY-SUBRC unequal to 0, otherwise to 0.

To retrieve the page number for a certain line number, use:

Syntax

DESCRIBE LIST LINE <lin> PAGE <pag> [INDEX <idx>].

This statement writes for list level <idx> the page number on which the list line number <lin> is found into the variable <pag>. The system sets SY-SUBRC as follows: if a list with index <idx> does not exist, to 8; if a line with number <lin> does not exist, to 4; otherwise to 0.

To retrieve the attributes of a certain page, use:

Syntax

DESCRIBE LIST PAGE <pag> [INDEX <idx>] [<options>]

This statement retrieves for list level <idx> the attributes specified in <options> for page <pag>. The system sets SY-SUBRC as follows: if a list with index <idx> does not exist, to 8; if a page with number <pag> does not exist, to 4; otherwise to 0.

The <options> of the statement are:

- LINE-SIZE <col>
writes the page width into the variable <col>.
- LINE-COUNT <len>
writes the page length into the variable <len>.
- LINES <lin>
writes the number of displayed lines into the variable <lin>.
- FIRST-LINE <lin1>
writes the absolute number of the first line into the variable <lin1>.
- TOP-LINES <top>
writes the number of page header lines into the variable <top>.
- TITLE-LINES <tit>
writes the number of list header lines of the standard page header into the variable <tit>.

Passing List Attributes

- HEAD-LINES <head>

writes the number of column header lines of the standard page header into the variable <head>.

- END-LINES <end>

writes the number of page footer lines into the variable <end>.

Use DESCRIBE LIST for completed lists only, since for lists in creation (index is SY-LSIND) some attributes are not up to date.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING
      LINE-SIZE 40 LINE-COUNT 5(1).
```

```
DATA: LIN TYPE I, PAG TYPE I,
      COL TYPE I, LEN TYPE I, LIN1 TYPE I,
      TOP TYPE I, TIT TYPE I, HEAD TYPE I, END TYPE I.
```

```
DO 4 TIMES.
  WRITE / SY-INDEX.
ENDDO.
```

```
TOP-OF-PAGE.
WRITE 'Demonstration of DESCRIBE LIST statement'.
ULINE.
```

```
END-OF-PAGE.
ULINE.
```

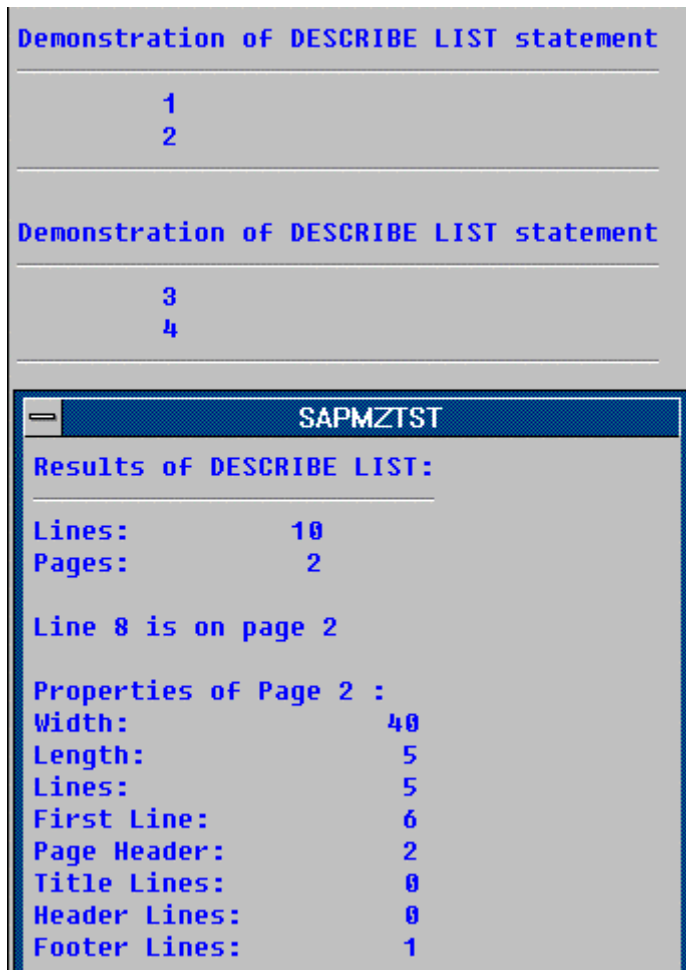
```
AT LINE-SELECTION.
NEW-PAGE LINE-COUNT 0.
WINDOW STARTING AT 1 13 ENDING AT 40 25.
DESCRIBE LIST: NUMBER OF LINES LIN INDEX 0,
               NUMBER OF PAGES PAG INDEX 0.
WRITE: 'Results of DESCRIBE LIST: '.
ULINE AT /(25).
WRITE: / 'Lines: ', LIN,
       / 'Pages: ', PAG.
SKIP.
DESCRIBE LIST LINE SY-LILLI PAGE PAG INDEX 0.
WRITE: / 'Line', (1) SY-LILLI, 'is on page', (1) PAG.
SKIP.
DESCRIBE LIST PAGE PAG INDEX 0 LINE-SIZE COL
      LINE-COUNT LEN
      LINES LIN
      FIRST-LINE LIN1
      TOP-LINES TOP
      TITLE-LINES TIT
      HEAD-LINES HEAD
      END-LINES END.
WRITE: 'Properties of Page', (1) PAG, ':',
      / 'Width: ', COL,
      / 'Length: ', LEN,
```

```

/'Lines: ', LIN,
/'First Line: ', LIN1,
/'Page Header: ', TOP,
/'Title Lines: ', TIT,
/'Header Lines:', HEAD,
/'Footer Lines:', END.

```

This program creates a two-page list of five lines per page. Two lines are used for the self-defined page header and one line for the page footer. If the user chooses the number 3 on the second page, the following dialog window appears:



While creating the secondary list, all variants of the DESCRIBE LIST statement apply to the basic list. The system displays the results in the dialog window. Note that the lines and pages to be described are defined interactively using SY-LILLI.

Manipulating Interactive Lists

Manipulating Interactive Lists

This topic describes how to manipulate the presentation and attributes of interactive lists. You learn

[Scrolling through Interactive Lists \[Page 1098\]](#)

[Setting the Cursor from within the Program \[Page 1100\]](#)

[Modifying List Lines \[Page 1107\]](#)

Scrolling through Interactive Lists

To scroll through interactive lists from within the program, use the SCROLL statement. How to use this statement with basic lists, is described in detail in [Scrolling from within the Program \[Page 975\]](#).

For using the SCROLL statement in interactive lists, beware of the following:

- You can use the SCROLL statement for completed lists only. If you use SCROLL before the first output statement of a list, it does not affect this list. If you use SCROLL after the first output statement of a list, it affects the entire list, that is, all subsequent output statements as well.
- While creating a secondary list, a SCROLL statement without INDEX option always refers to the previously displayed list on which the interactive event occurred (index SY-LISTI).
- Only when creating the basic list does the SCROLL statement refer to the currently created list.
- You can use the INDEX option to explicitly scroll on existing list levels. To do this, the lists need not be displayed. When the user displays the list again, it is scrolled to the specified position. If the specified list level does not exist, the system sets SY-SUBRC to 8.
- If, during an interactive event, you want to scroll through the list you are currently creating, use SY-LSIND as the index in the SCROLL statement. Note that a manipulation of SY-LSIND takes effect only at the end of the event on the created list, independent of where you place the statement in the processing block. If you want to set the list level explicitly, you must manipulate SY-LSIND in the last statement of the processing block to make sure that a SCROLL statement within the processing block accesses the correct list.

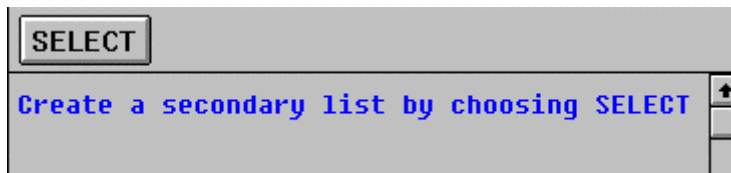
Another way of scrolling interactive lists from within the program is to use the SET USER-COMMAND statement in connection with the corresponding system-defined function codes (P...). For more information, see [Triggering Events from within the Program \[Page 1074\]](#).

```
REPORT SAPMZTST NO STANDARD PAGE HEADING LINE-SIZE 50.
SET PF-STATUS 'SELECT'.
WRITE 'Create a secondary list by choosing SELECT'.
AT USER-COMMAND.
NEW-PAGE LINE-SIZE 200.
CASE SY-UCOMM.
  WHEN 'SELE'.
    SET PF-STATUS 'SCROLLING'.
    DO 200 TIMES. WRITE SY-INDEX. ENDDO.
    SCROLL LIST RIGHT BY 48 PLACES INDEX SY-LSIND.
    SY-LSIND = SY-LSIND - 1.
  WHEN 'LEFT'.
    SCROLL LIST LEFT BY 12 PLACES.
  WHEN 'RGHT'.
```

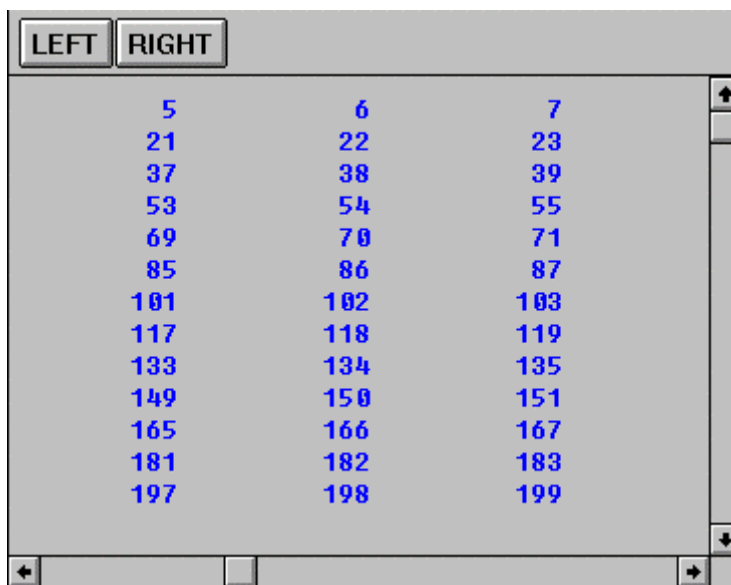
Scrolling through Interactive Lists

SCROLL LIST RIGHT BY 12 PLACES.
ENDCASE.

This program creates a basic list of one line with the status SELECT. In the status SELECT, the function code SELE (text *SELECT*) is assigned to function key F2 and to a pushbutton of the application toolbar.



After choosing *SELECT*, the system triggers the AT USER-COMMAND event and creates a secondary list with status SCROLLING. In the status SCROLLING, the function codes LEFT (text *LEFT*) and RGTH (text *RIGHT*) are assigned to the function keys F5 and F6 and to the application toolbar. The secondary list is 200 characters wide. The SCROLL statement scrolls the secondary list after its creation (SY-LSIND = 1) by 48 columns to the right. Then, SY-LSIND is decreased by 1 and the scrolled list replaces the basic list.



By clicking on *LEFT* and *RIGHT*, the user can scroll to the left and to the right in the displayed list. The SCROLL statements are programmed for the corresponding function codes within the AT USER-COMMAND event.

Setting the Cursor from within the Program

You can set the cursor on the current list dynamically from within your program. You can do this to support the user with entering values into input fields or selecting fields or lines. If input fields occur on a list, the system by default places the cursor into the first input field.

To set the cursor, use the SET CURSOR statement. This statement sets the cursor in the most recently created list. While creating the basic list, this is always the basic list itself. While creating a secondary list, this is the previous list.

With SET CURSOR, you can set the cursor to an absolute position, to a field, or to a line.

[Setting the Cursor Explicitly \[Page 1101\]](#)

[Setting the Cursor to a Field \[Page 1103\]](#)

[Setting the Cursor to a Line \[Page 1105\]](#)

Setting the Cursor Explicitly

Setting the Cursor Explicitly

To set the cursor to a certain position in the output window, use:

Syntax

SET CURSOR <col> <lin>.

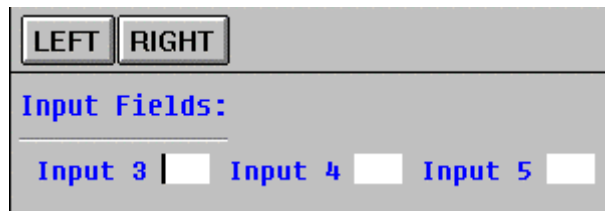
This statement sets the cursor to column <col> of line <lin> of the output window.

The system sets the cursor only to positions that are visible in the display. For positions outside the displayed area, it ignores the statement. To set the cursor to a part of the list currently not displayed, you must scroll the list first (see [Scrolling through Interactive Lists \[Page 1098\]](#)).

You can set the cursor only to lines that contain list output. These include lines skipped with the SKIP statement, but no underlines. If <lin> is below the bottom list line, the system sets the cursor to the bottom line.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING LINE-SIZE 80.
SET PF-STATUS 'SCROLLING'.
NEW-LINE NO-SCROLLING.
WRITE 'Input Fields:'.
NEW-LINE NO-SCROLLING.
WRITE '-----'.
NEW-LINE.
DO 5 TIMES.
  WRITE: ' Input', (1) SY-INDEX, ' ' INPUT ON NO-GAP.
ENDDO.
FORMAT INPUT OFF.
SET CURSOR 11 3.
AT USER-COMMAND.
CASE SY-UCOMM.
  WHEN 'LEFT'.
    IF SY-STACO > 1.
      SCROLL LIST LEFT BY 12 PLACES.
    ENDIF.
  WHEN 'RIGHT'.
    IF SY-STACO < 40.
      SCROLL LIST RIGHT BY 12 PLACES.
    ENDIF.
ENDCASE.
SET CURSOR 11 3.
```

This program creates a basic list that contains five input fields. The status SCROLLING is the same as the one defined in the example in [Scrolling through Interactive Lists \[Page 1098\]](#). The cursor is set to the first input field. The user can use the pushbuttons *LEFT* and *RIGHT* to scroll the list horizontally. After each scroll movement, the cursor is set to an input field again. After clicking *RIGHT* twice, for example, the output screen appear as follows:



A screenshot of a SAP dialog box. At the top, there are two buttons labeled 'LEFT' and 'RIGHT'. Below them, the text 'Input Fields:' is displayed in blue. Underneath, there are three input fields, each preceded by a label in blue: 'Input 3', 'Input 4', and 'Input 5'. Each input field is represented by a small white rectangle with a vertical cursor line on the left side.

Setting the Cursor to a Field

Setting the Cursor to a Field

To set the cursor to a certain field on a line of the displayed list, use:

Syntax

SET CURSOR FIELD <f> LINE <lin> [OFFSET <off>].

This statement sets the cursor on line <lin> onto the field whose name is stored in <f>. If a field appears more than once on a line, the system sets the cursor to the first field. If the field does not appear on the line or if it is outside the displayed area, the system ignores the statement. In the last case, use the SCROLL statement to scroll the line into the visible area of the window (see [Scrolling through Interactive Lists \[Page 1098\]](#)).

Use the OFFSET option to set the cursor to position <off> of the field stored in <f>. <off> = 0 indicates the first position.

```
REPORT SAPMZTST LINE-SIZE 40.

DATA: INPUT1(5) VALUE '*****',
      INPUT2(5) VALUE '*****',
      INPUT3(5) VALUE '*****'.

SET PF-STATUS 'INPUT'.

WRITE 'Input Fields:'.
WRITE / '-----'.
SKIP.

WRITE: 'Input 1', INPUT1 INPUT ON,
      / 'Input 2', INPUT2 INPUT ON,
      / 'Input 3', INPUT3 INPUT ON.

AT USER-COMMAND.

CASE SY-UCOMM.
  WHEN 'INP1'.
    SET CURSOR FIELD 'INPUT1' LINE 6 OFFSET 4.
  WHEN 'INP2'.
    SET CURSOR FIELD 'INPUT2' LINE 7 OFFSET 4.
  WHEN 'INP3'.
    SET CURSOR FIELD 'INPUT3' LINE 8 OFFSET 4.
ENDCASE.
```

This program creates a basic list that contains three input fields. In the status INPUT, the function codes INP1, INP2, and INP3 are assigned to the function keys F5, F6, and F7 and to the application toolbar. After selecting one of these functions, the system triggers the AT USER-COMMAND event. It places the cursor on the corresponding input field. If the user chooses *INPUT 2*, the output screen looks like this:

The screenshot displays a standard SAP ABAP input screen. At the top, there are three buttons labeled 'INPUT 1', 'INPUT 2', and 'INPUT 3'. Below these is a 'Standard Page Heading' section with the text 'Standard Page Heading' and a page number '1'. Underneath is a section titled 'Input Fields:'. This section contains three input fields, each preceded by a label: 'Input 1', 'Input 2', and 'Input 3'. Each input field is represented by a red asterisk pattern. A vertical cursor is visible in the second input field, positioned between the first and second asterisks.

Note that the system includes the header lines when positioning the cursor.

Setting the Cursor to a Line

Setting the Cursor to a Line

To set the cursor to a certain line of the list in the output window, use:

Syntax

SET CURSOR LINE <lin> [OFFSET <off>].

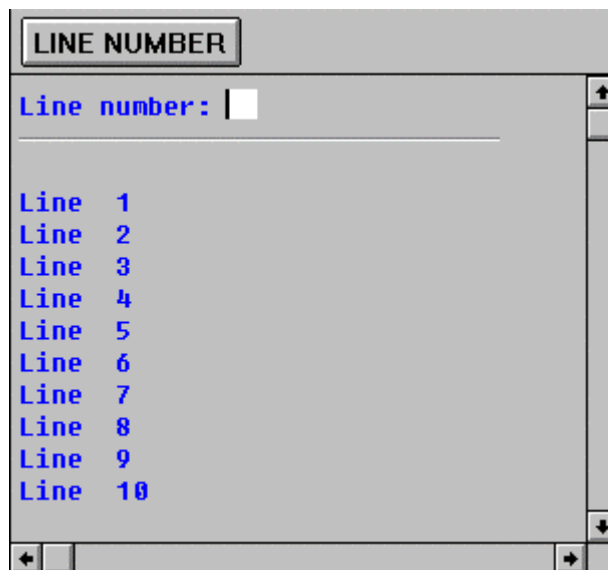
This statement sets the cursor to line <lin>. The system ignores the statement, if the line does not appear in the list or in the visible area of the window. In the last case, use the SCROLL statement first to scroll the line into the display (see [Scrolling through Interactive Lists \[Page 1098\]](#)).

Use the OFFSET option to set the cursor to column <off> of line <lin>. <off> = 0 indicates the first column. The system ignores the statement, if the position is outside the visible area of the list.

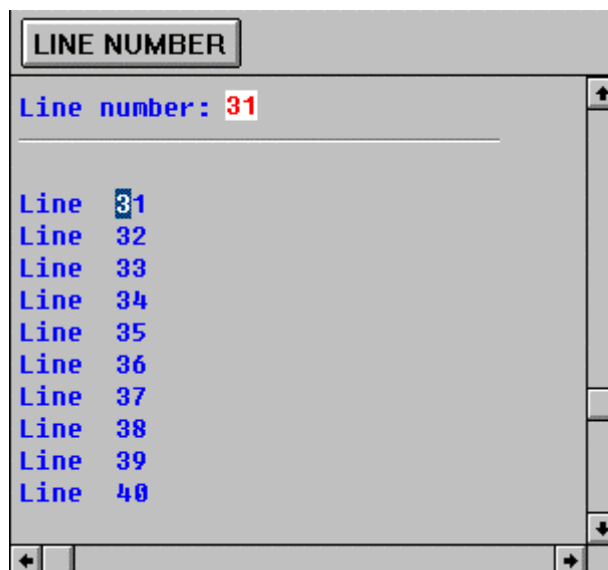
```
REPORT SAPMZTST LINE-SIZE 30 NO STANDARD PAGE HEADING.
DATA: INP(2), FLD(3), TOP(2).
SET PF-STATUS 'LINE_NUMBER'.
FLD = 'INP'.
DO 50 TIMES.
  WRITE: / 'Line ', (2) SY-INDEX.
ENDDO.
TOP-OF-PAGE.
WRITE: 'Line number:', INP INPUT ON.
ULINE.
SKIP.
AT USER-COMMAND.
  DESCRIBE LIST PAGE 1 TOP-LINES TOP.
  CASE SY-UCOMM.
    WHEN 'LINE'.
      READ LINE 1 FIELD VALUE INP.
      SCROLL LIST TO PAGE 1 LINE INP.
      INP = INP + TOP.
      SET CURSOR LINE INP OFFSET 6.
  ENDCASE.
```

This program creates the following basic list:

Setting the Cursor to a Line



In the status `LINE_NUMBER`, the function code `LINE` (text *LINE NUMBER*) is assigned to function key `F5` and to a pushbutton of the application toolbar. If the user enters a number into the input field and chooses *LINE NUMBER*, the system scrolls to the specified number and sets the cursor to it:



The system reads the input field using `READ LINE`. For setting the cursor, it uses `DESCRIBE LIST` to take into account the size of the page header. Note that this example makes excessive use of automatic type conversion.

Modifying List Lines

Modifying List Lines

To modify the lines of a completed list from within the program, use the MODIFY LINE statement. There are two ways to specify the line you want to modify:

- You can explicitly specify the line you want to modify:

Syntax

```
MODIFY LINE <n> [INDEX <idx>|OF CURRENT PAGE|OF PAGE <p>]  
      [<modifications>].
```

Without the first line of options, this statement modifies line <n> of the list on which an interactive event occurred (index SY-LISTI). With the options of the first line, you can specify the line you want to modify as follows:

- Use INDEX <idx> to specify line <n> of the list level with the index <idx>.
- Use OF CURRENT PAGE to specify line <n> of the currently displayed page (page number SY-CPAGE).
- Use OF PAGE <p> to specify line <n> of page <p>.

- You can refer to the line most recently read:

Syntax

```
MODIFY CURRENT LINE [<modifications>].
```

This statement modifies the line most recently read by means of line selection (F2) or of the READ LINE statement.

Without the option <modifications>, the above statements fill the current contents of the SY-LISEL system field into the specified line. The line's HIDE area is overwritten by the current values of the corresponding fields. However, this does not influence the displayed values.

If the system succeeded in modifying the specified line, it sets SY-SUBRC to 0, otherwise to a value unequal to 0.

Apart from the ones described above, the option <modifications> contains several other possibilities to modify the line. They are covered in these topics:

[Modifying Line Formatting \[Page 1108\]](#)

[Modifying Field Contents \[Page 1109\]](#)

[Modifying Field Formatting \[Page 1110\]](#)

Modifying Line Formatting

To modify the formatting of the line you want to change, use the option LINE FORMAT of the MODIFY statement as follows:

Syntax

MODIFY... LINE FORMAT <option₁> <option₂>....

This statement sets the output format of the entire modified line according to the format specified with <option_i>. You can specify the same format options as for the FORMAT statement (see [The FORMAT Statement \[Page 996\]](#)).

```
REPORT SAPMZTST LINE-SIZE 40 NO STANDARD PAGE HEADING.
```

```
DATA C TYPE I VALUE 1.
```

```
WRITE 'Select line to modify the background'.
```

```
AT LINE-SELECTION.
```

```
  IF C = 8.
```

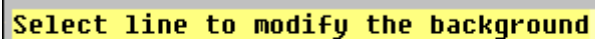
```
    C = 0.
```

```
  ENDIF.
```

```
  MODIFY CURRENT LINE LINE FORMAT COLOR = C.
```

```
  ADD 1 TO C.
```

This program creates an output line whose background color the user can change by selecting the line again and again:

A screenshot of a text line with a yellow background. The text is "Select line to modify the background" in a black, monospaced font. The line is highlighted against a gray background.

Modifying Field Contents

Modifying Field Contents

To explicitly modify the contents of fields in the line you want to change, use the option FIELD VALUE of the MODIFY statement:

Syntax

MODIFY... FIELD VALUE <f₁> [FROM <g₁>] <f₂> [FROM <g₂>]....

This statement overwrites the contents of the fields <f_i> in the list line with the current contents of the fields <f_i> or <g_i>. The System does not change the fields in the program. If necessary, the system converts the field type to type C.

If a field <f_i> occurs more than once on a list line, the system modifies only the first one. If a field <f_i> does not occur at all, the system ignores the option.

The system modifies existing fields <f_i> independent of the current contents you write from SY-LISEL into the line. If you made changes to the line at the output position of a field <f_i> using SY-LISEL, the FIELD VALUE option overwrites them.

```
REPORT SAPMZTST LINE-SIZE 40 NO STANDARD PAGE HEADING.  
DATA C TYPE I.  
WRITE: ' Number of selections:', (2) C.  
AT LINE-SELECTION.  
  ADD 1 TO C.  
  SY-LISEL(2) = '**'.  
  MODIFY CURRENT LINE FIELD VALUE C.
```

This program creates an output line, in which the user can modify the field C by selecting the line. At the same time, the system overwrites the first two characters of the line with two asterisks '**' due to the modification of SY-LISEL. After selecting the line four times, for example, the output looks like this:

```
** Number of selections: 4
```

Modifying Field Formatting

To modify the formatting of fields in the line you want to change, use the option FIELD FORMAT of the MODIFY statement as follows:

Syntax

MODIFY... FIELD FORMAT <f₁> <options₁> <f₂> <options₂>....

This statement sets the output format of the fields <f_i> occurring in the line according to the format specified in <options_i>. As <options_i>, you can specify one or more format options of the FORMAT statement (see [The FORMAT Statement \[Page 996\]](#)).

The option FIELD FORMAT overwrites the specifications of the LINE FORMAT option for the corresponding field(s). If a field <f_i> occurs more than once in the line, the system modifies only the first. If a field <f_i> does not occur in the line, the system ignores the option.

This example shows how you may continue processing checkboxes after evaluating them.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING.
```

```
DATA: BOX, LINES TYPE I, NUM(1).
```

```
SET PF-STATUS 'CHECK'.
```

```
DO 5 TIMES.
```

```
  NUM = SY-INDEX.
```

```
  WRITE: / BOX AS CHECKBOX, 'Line', NUM.
```

```
  HIDE: BOX, NUM.
```

```
ENDDO.
```

```
LINES = SY-LINNO.
```

```
TOP-OF-PAGE.
```

```
WRITE 'Select some checkboxes'.
```

```
ULINE.
```

```
AT USER-COMMAND.
```

```
  CASE SY-UCOMM.
```

```
    WHEN 'READ'.
```

```
      SET PF-STATUS 'CHECK' EXCLUDING 'READ'.
```

```
      BOX = SPACE.
```

```
      DO LINES TIMES.
```

```
        READ LINE SY-INDEX FIELD VALUE BOX.
```

```
        IF BOX = 'X'.
```

```
          WRITE: / 'Line', NUM, 'was selected'.
```

```
          BOX = SPACE.
```

```
          MODIFY LINE SY-INDEX
```

```
            FIELD VALUE BOX
```

```
            FIELD FORMAT BOX INPUT OFF
```

```
            NUM COLOR 6 INVERSE ON.
```

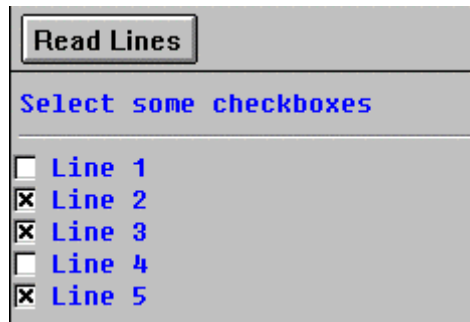
```
        ENDIF.
```

```
      ENDDO.
```

```
    ENDCASE.
```

Modifying Field Formatting

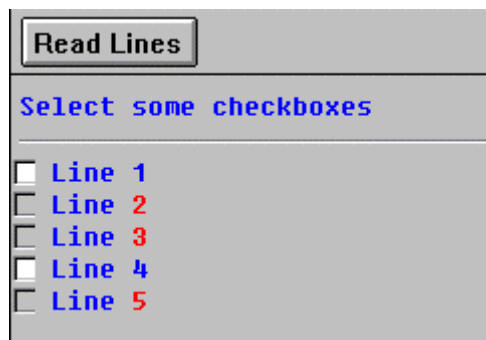
This program creates a basic list with the status CHECK. In the status CHECK, function code READ (text *Read Lines*) is assigned to function key F5 and to a pushbutton. The user can mark checkboxes and then choose *Read Lines*.



The screenshot shows a dialog box titled "Read Lines". Below the title bar is a header area with the text "Select some checkboxes". Below this is a list of five lines, each with a checkbox and the text "Line 1" through "Line 5". The checkboxes for "Line 2" and "Line 3" are checked, while the others are unchecked.

Checkbox	Text
<input type="checkbox"/>	Line 1
<input checked="" type="checkbox"/>	Line 2
<input checked="" type="checkbox"/>	Line 3
<input type="checkbox"/>	Line 4
<input checked="" type="checkbox"/>	Line 5

In the AT USER-COMMAND event, the system reads the lines of the list using READ LINE. It continues processing the selected lines on a secondary list. When returning to the basic list, the system deletes the marks in the checkboxes of the selected lines using MODIFY LINE and sets the format INPUT OFF to the checkboxes. In addition, it changes the format of field NUM.



The screenshot shows the same dialog box titled "Read Lines" with the header "Select some checkboxes". The list of five lines is shown, but now all checkboxes are unchecked. The line numbers "Line 2", "Line 3", and "Line 5" are displayed in red, while "Line 1" and "Line 4" are in blue.

Checkbox	Text
<input type="checkbox"/>	Line 1
<input type="checkbox"/>	Line 2
<input type="checkbox"/>	Line 3
<input type="checkbox"/>	Line 4
<input type="checkbox"/>	Line 5

The user can now mark only those lines that have not yet been changed.

You can include the lines of code executing the line modification from this program into the sample program in [Reading Lines from Lists \[Page 1087\]](#), making only slight modifications.

Calling Programs

If you need to program an extensive application, one single program will become very complex. To make the program easier to read, it is often reasonable to divide the required functions among several programs.

ABAP allows you to call reports as well as transactions using these statements:

	Report	Transaction
Call without return	SUBMIT	LEAVE TO TRANSACTION
Call and return	SUBMIT AND RETURN	CALL TRANSACTION

You can use these statements in any ABAP program. To call other programs in the context of dialog programming, see [Calling External Program Components \[Page 1323\]](#).

This topic shows how to call other programs from within reports. You use such calls mainly when processing interactive lists for interactive reporting. The user can explicitly call other programs by selecting list lines or choosing functions you provide. You can program the data transfer to the called programs according to the user action.

The topic is divided into

[Calling Executable Programs \(Reports\) \[Page 1113\]](#)

[Calling Transactions \[Page 1122\]](#)

Calling Executable Programs (Reports)

Calling Executable Programs (Reports)

To call executable programs (reports) from within other programs, use the SUBMIT statement.

To set the name of the called program statically in the program coding, write:

Syntax

SUBMIT <rep> [AND RETURN] [<options>].

To set the name of the called executable program (report) dynamically at runtime, write:

Syntax

SUBMIT (<rep>) [AND RETURN] [<options>].

The first statement starts the executable program (report) <rep>, the second statement starts the executable program (report) whose name is stored in field <rep>.

If the system cannot find the specified executable program (report) while executing the program, a runtime error occurs. If you specify the executable program (report) statically, you can double-click on <rep> in the ABAP Editor to create, display, or modify the called executable program (report).

If you omit the AND RETURN option, the system deletes all data and list levels of the calling executable program (report). After terminating the called executable program (report), it returns to the level from which you started the calling executable program (report).

If you use AND RETURN, the system stores the data of the calling executable program (report) and returns to the calling executable program (report) after processing the called executable program (report). The system resumes executing the calling executable program (report) from the statement following the call. To exit a executable program (report) called via SUBMIT... AND RETURN from within the program, see

[Exiting the Executable Program Called from Within the Program \[Page 1114\]](#)

Any executable program (report) called using SUBMIT, independent of the AND RETURN option, can access the ABAP memory of the calling program including the data clusters stored in the ABAP memory. For more information on clusters, see [Data Clusters in ABAP Memory \[Page 363\]](#).

The SUBMIT statement offers many other <options> for passing data and processing the called program. Some of them are described below.

[Manipulating the List Structure of the Called Executable Program \(Report\) \[Page 1116\]](#)

[Filling the Selection Screen of the Called Executable Program \(Report\) \[Page 1118\]](#)

For more information, read [Submitting Executable Programs \(Reports\) \[Page 1334\]](#) in the dialog programming part of this manual or the keyword documentation of SUBMIT.

Exiting the Executable Program Called from Within the Program

Exiting the Executable Program Called from Within the Program

Usually, the user exits a program you called using `SUBMIT... AND RETURN` by choosing F3 or F15 on list level 0 of the called report.

However, if you want to execute other statements, such as `EXPORT`ing data to the ABAP memory, before returning to the calling program, you must create a self-defined user interface for the called program. On this interface, you define your own function code for *Back* and process it at the `AT USER-COMMAND` event. After executing the desired statements, you leave the called report using this statement:

Syntax

`LEAVE.`

This statement leaves a report called using `SUBMIT... AND RETURN` and returns to the statement following the call in the calling program.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING.
```

```
DATA: ITAB TYPE I OCCURS 10,  
      NUM TYPE I.
```

```
SUBMIT SAPMZTS1 AND RETURN.
```

```
IMPORT ITAB FROM MEMORY ID 'HK'.
```

```
LOOP AT ITAB INTO NUM.  
  WRITE / NUM.  
ENDLOOP.
```

```
TOP-OF-PAGE.  
WRITE 'Report 1'.  
ULINE.
```

This program calls the following executable program (report):

```
REPORT SAPMZTS1 NO STANDARD PAGE HEADING.
```

```
DATA: NUMBER TYPE I,  
      ITAB TYPE I OCCURS 10.
```

```
SET PF-STATUS 'MYBACK'.
```

```
DO 5 TIMES.  
  NUMBER = SY-INDEX.  
  APPEND NUMBER TO ITAB.  
  WRITE / NUMBER.  
ENDDO.
```

```
TOP-OF-PAGE.  
WRITE 'Report 2'.  
ULINE.
```

```
AT USER-COMMAND.  
CASE SY-UCOMM.  
  WHEN 'MBCK'.  
    EXPORT ITAB TO MEMORY ID 'HK'.
```

Exiting the Executable Program Called from Within the Program

```
LEAVE.  
ENDCASE.
```

In the self-defined status MYBACK, the function code MBCK is assigned to the function keys F3 and F15:



If the user chooses *Back* on the interface MYBACK, the system transfers Table ITAB into the ABAP memory and then leaves SAPMZTS1. In SAPMZTST, it reads Table ITAB again.

Manipulating the List Structure of the Called Executable Program (Report)

Manipulating the List Structure of the Called Executable Program (Report)

To manipulate the list structure of an executable program (report) called using SUBMIT, write:

Syntax

SUBMIT... [LINE-SIZE <width>] [LINE-COUNT <length>].

If the called program contains **no** such options in the REPORT statement, the system formats the lists of the called program according to the options in the SUBMIT statement. If the called program's REPORT statement contains the corresponding options, the system uses these and ignores the options in the SUBMIT statement. For detailed information on these options, see [The Self-Defined List \[Page 953\]](#).

```
REPORT SAPMZTST NO STANDARD PAGE HEADING.

DATA: NAME(8) VALUE 'SAPMZTS1',
      WID TYPE I VALUE 80,
      LEN TYPE I VALUE 0.

SET PF-STATUS 'SELECT'.

WRITE: 'Select a report and its list format:',
      / '-----'.
SKIP.

WRITE: 'Report ', NAME INPUT ON,
      / 'Line size ', WID INPUT ON,
      / 'Page length', LEN INPUT ON.

AT USER-COMMAND.

CASE SY-UCOMM.
  WHEN 'SELE'.
    READ LINE: 4 FIELD VALUE NAME,
              5 FIELD VALUE WID,
              6 FIELD VALUE LEN.
    SUBMIT (NAME) LINE-SIZE WID LINE-COUNT LEN AND RETURN.
ENDCASE.
```

You can use this program to start executable programs that allow user-defined list formatting. On the basic list, the user can enter a report name and the desired list width and length by overwriting the default values:

Field	Value
Report	SAPMZTS1
Line size	80
Page length	0

Manipulating the List Structure of the Called Executable Program (Report)

At the AT USER-COMMAND event, the system reads the values and starts the specified executable program using SUBMIT. If the called program's REPORT statement does not contain LINE-SIZE or LINE-COUNT specifications, the system uses the values WID and LEN for creating the lists. After executing the called executable program (report), the user can change the input values on the basic list and call a new executable program (report).

Filling the Selection Screen of the Called Executable Program (Report)

When starting an executable program (report), the system usually displays the selection screen, on which the user enters the selection criteria and parameters of the connected logical database and of the program itself (see [Working with Selection Screens \[Page 795\]](#)). When you call an executable program (report) from within another report, you have several possibilities to fill the selection criteria and parameters of the called program.

Use the following options of the SUBMIT statement:

Syntax

```
SUBMIT... [VIA SELECTION-SCREEN]
  [USING SELECTION-SET <var>]
  [WITH <sel> <criterion>]
  [WITH FREE SELECTIONS <freesel>]
  [WITH SELECTION-TABLE <rspar>].
```

These options have the following effects:

- **VIA SELECTION-SCREEN**
The selection screen of the called executable program (report) appears. If you transfer values to the program using one or more of the other options, the corresponding input fields in the selections screen are filled. The user can change these values. By default, the system does not display a selection screen after SUBMIT.
- **USING SELECTION-SET <var>**
This option tells the system to start the called program with the variant <var> (see [Pre-Setting Selections Using Variants \[Page 864\]](#)).
- **WITH <sel> <criterion>**
Use this option to fill individual elements <sel> of the selection screen (selection tables and parameters). Use one of the elements <criterion>:
 - **<op> <f> [SIGN <s>]**, for single value selection
If <sel> is a selection criterion, use <op> to fill the OPTION field, <f> to fill the LOW field, and <s> to fill the SIGN field of the selection table <sel> in the called program (see [Selection Tables \[Page 816\]](#)).
 - If <sel> is a parameter, use any operator for <op>. You always fill parameter <sel> with <f>.
 - **[NOT] BETWEEN <f₁> AND <f₂> [SIGN <s>]**, for interval selection
You transfer <f₁> into the LOW field, <f₂> into the HIGH field, and <s> into the SIGN field of the selection table <sel> in the called report. If you omit the NOT option, the system fills the value BT into the OPTION field; if you use NOT, the system fills NB into the OPTION field (see [Selection Tables \[Page 816\]](#)).
 - **IN <seltab>**, transferring a selection table
You fill the selection table <sel> in the called executable program (report) with the values of Table <seltab> of the calling report. Table <seltab> must have the

Filling the Selection Screen of the Called Executable Program (Report)

structure of a selection table. Use the RANGES statement to create such a table (see [RANGES \[Page 818\]](#)).

- WITH FREE SELECTION <freesel>, user dialog for dynamic selections

To use this option, both calling and called programs must be connected to a logical database that supports dynamic selections. In the calling program, use the function modules FREE_SELECTIONS_INIT and FREE_SELECTIONS_DIALOG. They allow the user to enter dynamic selections on a selection screen. One export parameter of these function modules has structure RSDS_TEXPR from the RSDS type group. Transfer the values of this export parameter by means of the internal table <freesel> of the same structure to the called report.

- WITH SELECTION-TABLE <rspar>, dynamic transfer of values

First, create an internal table <rspar>, using the Dictionary structure RSPARAMS. The table then consists of the following six fields:

- SELNAME (type C, length 8) for the name of the selection criterion or parameter
- KIND (type C, length 1) for the selection type (S for selection criterion, P for parameter)
- SIGN, OPTION, LOW, HIGH as in a normal selection table (see [Selection Tables \[Page 816\]](#)), except that LOW and HIGH both have type C and length 45.

Within the calling report, you can dynamically fill this table with any values desired for the selection screen of the called executable program (report). If the name of a selection criterion appears more than once, the criterion occupies several lines of the selection tables in the called program. If the name of a parameter appears more than once, the system uses the last value. Note that LOW and HIGH have type C, so that the system executes type conversions to the criteria of the called program. This is important for date fields, for example. Therefore, check your programs using the VIA SELECTION-SCREEN option.

Except WITH SELECTION-TABLE, you can use any of the above options several times and in any combination within a SUBMIT statement. Especially the WITH <sel> option can be used several times for one single criterion <sel>. In the called program, the system appends the corresponding lines to the selection tables used. For parameters, it uses the last value specified. The only combination possible for the WITH SELECTION-TABLE option is USING SELECTION-SET.

During interactive events, you can use the above options of the SUBMIT statement to fill the selection screen of the called executable program (report) with data from the HIDE area of the selected line. This allows you to by-pass the restriction that you cannot use logical databases during interactive events. Divide the different GET statements among different programs and, after the user selected a line, call these reports using SUBMIT and passing the corresponding values.

If the input fields of selection screens are connected to SPA/GPA parameters (see [Using Default Values from SAP Memory \[Page 812\]](#)), you can use the SPA/GPA technique to transfer data to the selection screens. For more information in this technique, see [Transferring SPA/GPA Parameters to Transactions \[Page 1123\]](#).

Filling the Selection Screen of the Called Executable Program (Report)

The following executable program (report) creates a selection screen containing the parameter PARAMET and the selection criterion SELECTO:

```
REPORT SAPMZTS1.
DATA NUMBER TYPE I.
PARAMETERS  PARAMET(14).
SELECT-OPTIONS SELECTO FOR NUMBER.
```

The executable program (report) SAPMZTS1 is called by the following program, using different selection criteria:

```
REPORT SAPMZTST NO STANDARD PAGE HEADING.

DATA: INT TYPE I,
      RSPAR LIKE RSPARAMS OCCURS 10 WITH HEADER LINE.

RANGES SELTAB FOR INT.

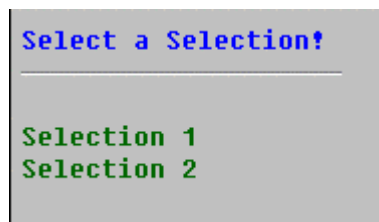
WRITE: 'Select a Selection!',
      / '-----'.
SKIP.

FORMAT HOTSPOT COLOR 5 INVERSE ON.
WRITE: 'Selection 1',
      / 'Selection 2'.

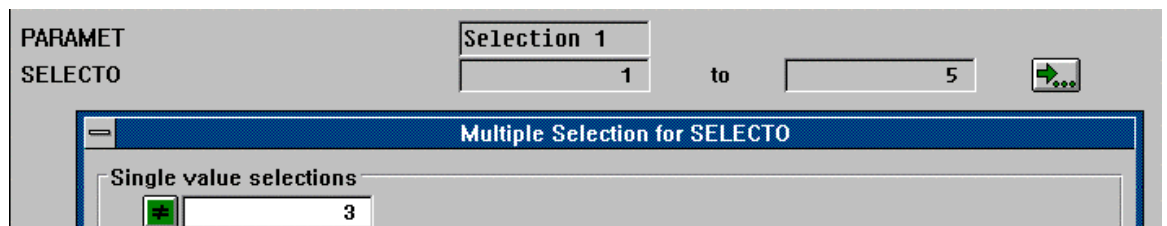
AT LINE-SELECTION.
CASE SY-LILLI.
  WHEN 4.
    SELTAB-SIGN = 'I'. SELTAB-OPTION = 'BT'.
    SELTAB-LOW = 1. SELTAB-HIGH = 5.
    APPEND SELTAB.
    SUBMIT SAPMZTS1 VIA SELECTION-SCREEN
      WITH PARAMET EQ 'Selection 1'
      WITH SELECTO IN SELTAB
      WITH SELECTO NE 3
      AND RETURN.
  WHEN 5.
    RSPAR-SELNAME = 'SELECTO'. RSPAR-KIND = 'S'.
    RSPAR-SIGN = 'E'. RSPAR-OPTION = 'BT'.
    RSPAR-LOW = 14. RSPAR-HIGH = 17.
    APPEND RSPAR.
    RSPAR-SELNAME = 'PARAMET'. RSPAR-KIND = 'P'.
    RSPAR-LOW = 'Selection 2'.
    APPEND RSPAR.
    RSPAR-SELNAME = 'SELECTO'. RSPAR-KIND = 'S'.
    RSPAR-SIGN = 'I'. RSPAR-OPTION = 'GT'.
    RSPAR-LOW = 10.
    APPEND RSPAR.
    SUBMIT SAPMZTS1 VIA SELECTION-SCREEN
      WITH SELECTION-TABLE RSPAR
      AND RETURN.
ENDCASE.
```

After executing the report, the following basic list appears:

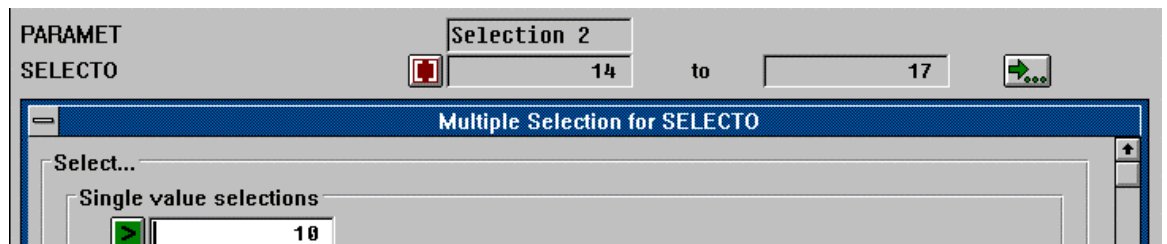
Filling the Selection Screen of the Called Executable Program (Report)



After clicking once on the first hotspot, the selection screen of SAPMZTS1 looks like this:



After clicking once on the second hotspot, the selection screen of SAPMZTS1 appears like this:



For both calls of SAPMZTS1, the system transfers values that lead to two-line selection tables SELECTO. The second line appears in the respective dialog window *Multiple Selection for SELECTO*. Without the VIA SELECTION-SCREEN option of the SUBMIT statement, PARAMET and SELECTO would be filled accordingly in SAPMZTS1, but they would not be displayed.

Calling Transactions

To call transactions from within an executable program (report), ABAP offers two possibilities.

If you do not want to return to the calling program after terminating the transaction, use:

Syntax

LEAVE TO TRANSACTION <tcod> [AND SKIP FIRST SCREEN].

This statement ends the executable program (report) and starts transaction <tcod>.

If you want to return to the calling program after terminating the transaction, use:

Syntax

CALL TRANSACTION <tcod> [AND SKIP FIRST SCREEN] [USING <itab>].

This statement saves the data of the executable program (report) and starts transaction <tcod>. At the end of the transaction, the system returns to the statement following the call in the calling report.

You can use a variable to specify transaction <tcod>. This allows you to call transactions statically as well as dynamically.

Use the AND SKIP FIRST SCREEN option to suppress the display of the transaction's initial screen.

If you use the AND SKIP FIRST SCREEN option, the system displays the screen of the transaction, which is registered in the initial screen's attribute *Next screen* in the Screen Painter. Therefore, if for the initial screen its own number is registered as *Next screen*, you cannot skip it.

Furthermore, the AND SKIP FIRST SCREEN option works only if all mandatory fields on the transaction's initial screen are filled completely and correctly with input values from SPA/GPA parameters. For information on the SPA/GPA technique, see:

[Transferring SPA/GPA Parameters to Transactions \[Page 1123\]](#)

Use the USING <itab> option of the CALL TRANSACTION statement to transfer an internal table <itab> to the called transaction that has the format of a batch input table. For more information on batch input, see the documentation [BC Basis Programming Interfaces \[Ext.\]](#).

Transferring SPA/GPA Parameters to Transactions

Transferring SPA/GPA Parameters to Transactions

To fill the input fields of a called transaction with data from the calling program, you can use the SPA/GPA technique. SPA/GPA parameters are values that the system stores in the global, user-related SAP memory. You use the SAP memory to transfer values between programs beyond the borders of transactions. A user can access the values stored in the SAP memory during one terminal session for all modes used in parallel.

Usually, the input fields on the initial screen of a transaction are connected to SPA/GPA parameters. If you fill these parameters from within your program before calling the transaction, the system fills the input fields with the corresponding values.

To fill an SPA/GPA parameter, use:

Syntax

SET PARAMETER ID <pid> FIELD <f>.

This statement saves the contents of field <f> under the ID <pid> in the SAP memory. Use three characters for the ID <pid>. If the ID <pid> exists, this statement overwrites the value previously stored there.

If the ID <pid> does not exist, double-click on <pid> in the ABAP Editor to create a new parameter object.

To read an SPA/GPA parameter into an ABAP program, use:

Syntax

GET PARAMETER ID <pid> FIELD <f>.

This statement fills the value stored under the ID <pid> into the variable <f>. If the system does not find a value for <pid> in the SAP memory, it sets SY-SUBRC to 4, otherwise to 0. To transfer parameters to a called program, you do not need this statement.

When calling transactions, you must know which SPA/GPA parameters are connected to the input fields on the transaction's initial screen. To find this out, start the transaction, position the cursor on the individual input fields, choose *Help*, and on the dialog window that appears, choose *Technical Information*. The dialog window *Technical Information* appears, showing the appropriate SPA/GPA parameter in the field *Parameter ID*.

The technical information for the first input field of the booking transaction TCG2 looks like this:

Transferring SPA/GPA Parameters to Transactions

The screenshot shows a SAP R/3 Help window titled 'Help - SAP R/3'. The main content area displays information for the field 'Airline'. It includes the label 'Airline carrier Id' and a 'Definition' stating 'This field contains the ID of the airline carrier.' To the left of this information is a form with input fields for 'Airline', 'Connection number', 'Date of flight', and 'Booking number'. Below the main content area is a 'Technical Information' dialog box. This dialog box contains several sections: 'Screen data' (Program name: SAPMTGG2, Screen number: 0100), 'GUI data' (Program: SAPMTGG2, Status: MEN), 'Field data' (Table name: SBOOK, Field name: CARRID, Data element: S_CARR_ID, DE supplement: 0, Parameter ID: CAR), and 'Field description for batch input' (Scrn field: SBOOK-CARRID).

The SPA/GPA parameter for the input field *Company* has the ID CAR. Use this method to find the IDs CON, DAY, and BOK for the other input fields.

The following executable program (report) is connected to the logical database F1S and calls Transaction TCG2:

REPORT SAPMZTST NO STANDARD PAGE HEADING.

TABLES SBOOK.

START-OF-SELECTION.

WRITE: 'Select a booking',
/ '-----'.

SKIP.

GET SBOOK.

WRITE: SBOOK-CARRID, SBOOK-CONNID,
SBOOK-FLDATE, SBOOK-BOOKID.

HIDE: SBOOK-CARRID, SBOOK-CONNID,
SBOOK-FLDATE, SBOOK-BOOKID.

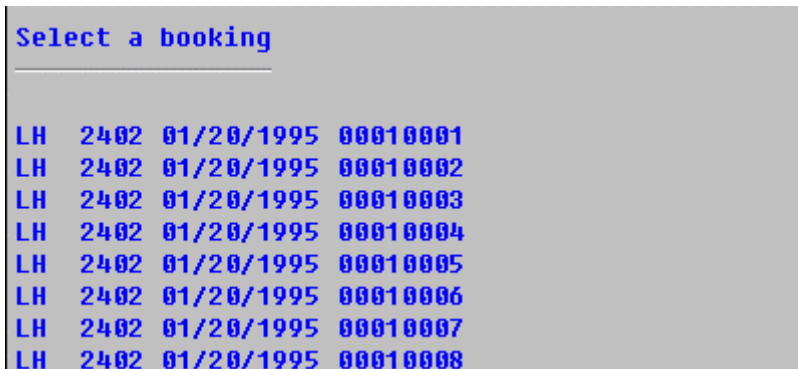
AT LINE-SELECTION.

SET PARAMETER ID: 'CAR' FIELD SBOOK-CARRID,
'CON' FIELD SBOOK-CONNID,
'DAY' FIELD SBOOK-FLDATE,
'BOK' FIELD SBOOK-BOOKID.

CALL TRANSACTION 'TCG2'.

Transferring SPA/GPA Parameters to Transactions

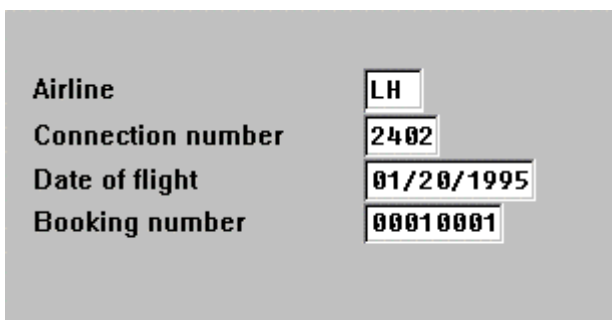
The basic list of the program shows fields from the database table SBOOK according to the user entries on the selection screen. These data are also stored in the HIDE areas of each line.



The screenshot shows a window titled "Select a booking" with a list of flight bookings. Each line contains the airline code, connection number, date of flight, and booking number.

	Airline	Connection number	Date of flight	Booking number
LH	2402	01/20/1995	00010001	
LH	2402	01/20/1995	00010002	
LH	2402	01/20/1995	00010003	
LH	2402	01/20/1995	00010004	
LH	2402	01/20/1995	00010005	
LH	2402	01/20/1995	00010006	
LH	2402	01/20/1995	00010007	
LH	2402	01/20/1995	00010008	

If the user selects a line of booking data by double-clicking, the system triggers the AT LINE-SELECTION event and takes the data stored in the HIDE area to fill them into the SPA/GPA parameters of the initial screen of Transaction TGC2. Then it calls the transaction. Since you do not suppress the initial screen using AND SKIP FIRST SCREEN, the initial screen may appear as follows:



The screenshot shows the initial screen of Transaction TGC2 with the following fields filled in:

Airline	LH
Connection number	2402
Date of flight	01/20/1995
Booking number	00010001

If you would use the AND SKIP FIRST SCREEN option with the CALL TRANSACTION statement, the second screen would appear immediately, since all mandatory fields of the first screen are filled.

Printing Lists

You use lists to output structured, formatted data. By default, the system sends a list (basic list and secondary lists) to the output screen after creating it. This section describes how to send lists to the SAP spool system instead of the output screen.

Within ABAP, sending a list to the SAP spool system is generally called 'printing lists'. This, however, must not necessarily mean that the list is actually printed on a printer. You can use the spool system also to store a list temporarily. And you can use the spool system to store lists in an optical archive instead of print them.

ABAP offers two possibilities to print a list:

You can print a list **after** and **while** creating it.

[Printing a List after Creating it \[Page 1127\]](#)

[Printing a List while Creating it \[Page 1129\]](#)

Additional Information about this Section

For information on the SAP spool system, see the documentation [BC Printing Guide \[Ext.\]](#).

For information on optical archiving of lists, see the documentation [SAP ArchiveLink \[Ext.\]](#) and the section "Optical Archiving of ABAP Lists" in the documentation [SAP ArchiveLink - Scenarios in Applications \[Ext.\]](#).

Printing a List after Creating it

Printing a List after Creating it

When printing a list after creating it, you do not use any of the print-specific statements described in the topics below to send the list from within the program to the SAP spool system.

By default, the system sends the completed list to the output screen. If the *Print* function (function code PRI) in the status of the list's user interface has been activated, the user can send the screen list to the SAP spool system by choosing *Print* (see [Printing the Output List \[Page 947\]](#)). On the dialog window *Print Screen List*, the system asks for the print parameters (see [Print Parameters \[Page 1130\]](#)). To modify the pre-settings of this screen, see [Print Parameters -Pre-setting Values \[Page 1135\]](#).

Printing a list after creating it provides several problems:

- You format a list, which you display on the output screen, for screen display and not for print output. For the reasons stated below, you cannot always use the display format for printing:
 - A list on an output screen usually consists of one single page (see notes in [Determining the Page Length \[Page 958\]](#)). When printing, the system 'cuts' this logical page into several physical pages whose format depends on the print parameters specified. The system places the page header on each of these print pages. If the page header contains a page number, this number is the same on all pages (SY-PAGNO). This prohibits numbering the printed pages consecutively.
 - If the list contains page breaks programmed using NEW-PAGE (see [Unconditional Page Break \[Page 964\]](#)), these page breaks are not adapted to the format of the print pages, which may lead to further automatic page breaks. For a print page created by an automatic page break, the system uses the same page header as for the previous page, since only NEW-PAGE increases the SY-PAGNO system field.
 - If the list consists of several pages due to the LINE-COUNT option in the REPORT or NEW-PAGE statement (see [Lists with Several Pages \[Page 962\]](#)), you can either not print the list at all, since the page length specified exceeds the maximum page length of a print page, or you do not make full use of the physical print page.
 - You can set the width of a list on an output screen to any value between 1 and 255 columns (see [Determining the List Width \[Page 956\]](#)). This list width is not adapted to a printer format. For example, a normal printer prints lists wider than 132 columns with the same small letter size as lists of 255 columns because there is no print format in between.
- When creating a list for screen output, you cannot include print control statements into the list (see [Print Control \[Page 1152\]](#)).
- At the end of each print page, you cannot output footer lines defined in the program. Instead, mark *Footer* on the dialog window *Print Screen List*. The system then reserves one line on each page for the system-defined footer line.
- Screen lists do not contain any index information for optical archiving. You can specify such index information for optical archiving only when printing the list during its creation (see [Indexing Print Lists for Optical Archiving \[Page 1159\]](#)).

The printout of a completed list from the output screen is a hardcopy of the screen rather than a real program-controlled printout. Use this method for testing purposes only, or for lists whose

Printing a List after Creating it

formats are acceptable to the printer. For complex lists (for example, lists containing extensive page headers that shall not appear on each print page), use print statements within your program (see [Printing a List while Creating it \[Page 1129\]](#)).

If you want to offer the user the possibility of starting a program-controlled print process from the output screen, use the methods of interactive reporting (see [Interactive Lists \[Page 1030\]](#)). For example, you first create a list for the output screen. Use a self-defined user interface in which you replace function code PRI with a self-defined function code. During the AT USER-COMMAND event, re-create the list for the spool system (see [Printing a List while Creating it \[Page 1129\]](#)).

Printing a List while Creating it

Printing a List while Creating it

When you print a list while creating it, you receive best print output, since the system formats the list according to the requirements of the printer. The system sets list width and page length according to the print format. This prevents lines from being wider than the print format in use. Page breaks occur at the end of a physical print page.

The program must know the print format before it starts creating the list. The print format is part of the print parameters. Print parameters are set either interactively by the user or from within the program.

[Print Parameters \[Page 1130\]](#)

ABAP offers the following possibilities to print a list while creating it:

- If your program displays a selection screen, the user can choose *Execute + print* on the selection screen.

[Executing and Printing \[Page 1143\]](#)

- You can start print output from within your program using the NEW-PAGE PRINT ON statement.

[Printing from within the Program \[Page 1146\]](#)

- You can call an executable program (report) using the SUBMIT... TO SAP-SPOOL statement.

[Printing Lists of Called Executable Programs \(Reports\) \[Page 1150\]](#)

- You can include a report into a background job using the function module JOB_SUBMIT. For more information on background jobs and on function module JOB_SUBMIT, see the documentation [BC Basis-Programming Interfaces \[Ext.\]](#).

When printing a list while creating it, you can manipulate the print format:

[Print Control \[Page 1152\]](#)

When printing a list while creating it, the system sends each completed page to the spool system and then deletes it. The length of a printed list, therefore, is restricted only by the capacity of the spool system. In contrast to lists for display, the system does not store list levels when printing. Since an entire list in printing never exists, you cannot refer to the contents of previous pages.

Print Parameters

You must set print parameters before the printing process starts.

When printing lists **after** creating them, the system uses the print format specified in the print parameters to split the completed list and fit it onto the print pages, truncating it if necessary.

When printing lists **while** creating them, the system uses the print format to actually format the list in the program.

Print parameters are set either interactively by the user or from within the program.

The topics below contain:

[Print Parameters- Overview \[Page 1131\]](#)

[Print Parameters -Pre-setting Values \[Page 1135\]](#)

[Setting Print Parameters from within the Program \[Page 1137\]](#)

Print Parameters- Overview

Print Parameters- Overview

For each print process, the spool system needs a complete and consistent set of print parameters. In ABAP, the fields strings of structure PRI_PARAMS (ABAP Dictionary) represent a set of print parameters.

When passing the print parameters interactively, the system displays the following dialog window after starting the report, asking for the most important print parameters.

The table below shows which input fields of the *Print List Output* dialog window conform to which PRI_PARAMS components.

Input field	Component	Description
<i>Output device</i>	PDEST	Name of a printer or fax (pre-set from user defaults)
<i>Number of copies</i>	PRCOP	Number of printed copies (pre-set with: 1)
<i>Name</i>	PLIST	Name of the spool request. Set this only if you do not want to print immediately (pre-set with report name (SY-REPID) including the first three characters of the user name (SY-

Print Parameters- Overview

		UNAME)).
<i>Title</i>	PRTXT	Description text of the spool request. This text appears on the standard cover page. PRTXT also replaces the name PLIST in the list of print requests (<i>System</i> → <i>Services</i> → <i>Print requests</i>).
<i>Authorization</i>	PRBER	Authorization for the spool request. Only users with this authorization can view the contents of the request.
<i>Print immed.</i>	PRIMM	If you mark this field, the system sends the spool request to the <i>Output device</i> immediately after completing it. (pre-set from user defaults)
<i>Delete after print</i>	PRREL	If you mark this field, the system deletes the spool request immediately after outputting it on the <i>Output device</i> . Otherwise, it deletes the request after the <i>Retention period</i> expired (pre-set with from user defaults).
<i>New spool request</i>	PRNEW	If you mark this field, the system creates a new spool request. Otherwise, the system tries to append this spool request to a request that is not yet completed. In this case, the <i>Name</i> , <i>Output device</i> , <i>Number of copies</i> , and <i>Format</i> must be the same.
<i>Retention period</i>	PEXPI	Number of days for which the system holds the spool request before deleting it. (pre-set with: 8)
<i>Archiving mode</i>	ARMOD	Specify the archiving mode. Click on the possible entries button to choose <i>Print</i> , <i>Archive</i> , and <i>Print and archive</i> (ARMOD is 1, 2, or 3). (pre-set with: <i>Print</i>)
<i>SAP cover sheet</i>	PRSAP	If this field contains 'X', the system creates a standard cover page containing several data. If it contains 'D', it depends on the setting of the output device whether a cover page is printed or not. If the field is empty, the system does not create a cover page. (pre-set with: 'D')
<i>Recipient</i>	PRREC	Specify the recipient name for the <i>SAP cover sheet</i> (pre-set with: name of the user)
<i>Department</i>	PRABT	Specify the department name for the <i>SAP cover sheet</i>

Print Parameters- Overview

		(pre-set with value from user address)
<i>Lines</i>	LINCT	Number of list lines. This field has the same effect as the LINE-COUNT option of the REPORT statement. You cannot specify 0 (unlimited number of lines) for printing. The maximum number in this field depends on the contents of the <i>Format</i> field. (pre-set internally)
<i>Columns</i>	LINSZ	Number of characters per list line. This field has the same effect as the LINE-SIZE option of the REPORT statement. The maximum number in this field depends on the contents of the <i>Format</i> field. (pre-set internally)
<i>Format</i>	PAART	This field actually determines the page format of the output. Depending on the printer connected, you can set different formats with different maximum page length and width values in this field. (pre-set internally)

The *Print List Output* dialog window checks the input values for consistency and completeness. You cannot print if the print parameters are inconsistent (for example, using output formats the specified output device does not support).

The print parameters LINCT and LINSZ **do not** overwrite the LINE-COUNT and LINE-SIZE options of the REPORT or NEW-PAGE statements. If you use these options in your program, the values you specify there fill the components LINCT and LINSZ. The corresponding input fields in the *Print List Output* dialog window no longer accept input then. If the values you specify exceed the maximum values determined in the *Format* field, you cannot print the list.

In addition to the print parameters, there are archiving parameters. However, you must specify them only if optical archiving is turned on (archiving mode is *Archive* or *Print and archive*). In ABAP, field strings with the same structure as ARC_PARAMS (ABAP Dictionary) represent a set of archiving parameters. If optical archiving is turned on and the print parameters are set interactively, another dialog window *Archive Parameters* appears, on which the user must set the most important archiving parameters:

The table below shows which input fields of the *Archive Parameters* dialog window conform to which ARC_PARAMS components.

Input field	Component	Description
<i>Object type</i>	SAP_OBJECT	Object type of the SAP object
<i>Document type</i>	AR_OBJECT	Document type of the archiving object
<i>Information</i>	INFO	Information short form of the archiving request
<i>Text</i>	ARCTEXT	Description text of the archiving request

You cannot archive a list unless the entries you make in this dialog window are consistent and complete. For optical archiving it can be a good idea to write index information into the list in order support a later search in the archived list. This is only possible for printing lists during their creation (see [Indexing Print Lists for Optical Archiving \[Page 1159\]](#)).

To set print and archiving parameters from within the program, you must use the function module GET_PRINT_PARAMETERS (see [Setting Print Parameters from within the Program \[Page 1137\]](#)). The system does not accept any values directly assigned to the print and archiving parameters sets. If you assign values directly and use these values afterwards, a runtime error occurs.

Print Parameters -Pre-setting Values

Print Parameters -Pre-setting Values

The *Print List Output* dialog window **always** appears after the user chooses

- *Execute + print* on the selection screen.
- *Print* on the list interface.

You cannot suppress the dialog window for these user actions.

However, you can pre-set values for the *Print List Output* dialog window from within the program. Use the function module SET_PRINT_PARAMETERS. This function module has no export parameters and **only** takes effect for the printing of lists that was triggered by one of the above user actions.

For *Execute + print* on the selection screen, you must call SET_PRINT_PARAMETERS during the AT SELECTION-SCREEN event (or earlier). For *Print* on the list interface, you must call the function module before sending the list to the output screen (or earlier).

The table below shows how the import parameters of SET_PRINT_PARAMETERS are related to the print and archiving parameters:

Import parameters	Parameters	Description
IN_PARAMETERS	PRI_PARAMS	Entire set
IN_ARCHIVE_PARAMETERS	ARC_PARAMS	Entire set
ARCHIVE_MODE	PRI_PARAMS-ARMOD	Archiving mode
AUTHORITY	PRI_PARAMS-PRBER	Authorization
COPIES	PRI_PARAMS-PRCOP	Number of copies
COVER_PAGE	PRI_PARAMS-PRBIG	Selection cover page
DATA_SET	PRI_PARAMS-PRDSN	Spool file
DEPARTMENT	PRI_PARAMS-PRABT	Department name
DESTINATION	PRI_PARAMS-PDEST	Output device
EXPIRATION	PRI_PARAMS-PEXPI	Spool retention time
IMMEDIATELY	PRI_PARAMS-PRIMM	Print immediately
LAYOUT	PRI_PARAMS-PAART	Page layout
LINE_COUNT	PRI_PARAMS-LINCT	Lines per page
LINE_SIZE	PRI_PARAMS-LINSZ	Columns per line
LIST_NAME	PRI_PARAMS-PLIST	Name of spool request
LIST_TEXT	PRI_PARAMS-PRTXT	Description text
NEW_LIST_ID	PRI_PARAMS-PRNEW	New spool request
RECEIVER	PRI_PARAMS-PRREC	Recipient
RELEASE	PRI_PARAMS-PRREL	Delete after output
SAP_COVER_PAGE	PRI_PARAMS-PRSAP	SAP cover page

Print Parameters -Pre-setting Values

TYPE	PRI_PARAMS-PTYPE	Type of spool request
FOOT_LINE	PRI_PARAMS-FOOTL	Output footer line
ARCHIVE_ID	ARC_PARAMS-ARCHIV_ID	Target archive
ARCHIVE_INFO	ARC_PARAMS-INFO	Information
ARCHIVE_TEXT	ARC_PARAMS-ARCTEXT	Description text
AR_OBJECT	ARC_PARAMS-AR_OBJECT	Document type
SAP_OBJECT	ARC_PARAMS-SAP_OBJECT	Object type

To see which input fields in the dialog windows *Print List Output* and *Archive Parameters* conform to these parameters, see the tables in [Print Parameters- Overview \[Page 1131\]](#).

To the parameters IN_PARAMETERS and IN_ARCHIVE_PARAMETERS, you must assign field strings with the structures PRI_PARAMS and ARC_PARAMS, respectively. These field strings must either be initial or contain the results of the function module GET_PRINT_PARAMETERS (see [Setting Print Parameters from within the Program \[Page 1137\]](#)).

The system uses the FOOT_LINE parameter only, if the user chooses *Print* on the list interface. If this parameter equals 'X', the system outputs one system-defined footer line on each page.

To include the function module into your program, use *Edit → Insert statement...CALL FUNCTION* in the ABAP Editor.

For an example showing how to use SET_PRINT_PARAMETERS, see [Executing and Printing \[Page 1143\]](#).

Setting Print Parameters from within the Program

Setting Print Parameters from within the Program

If you use the print statements

- NEW-PAGE PRINT ON
- SUBMIT... TO SAP-SPOOL
- CALL FUNCTION 'JOB-SUBMIT'

for printing, you set the print parameters from within the program, using the corresponding options of the print statements. You can choose between allowing or suppressing a user dialog via the *Print List Output* dialog window.

To ensure that the parameters are sent to the spool system properly and completely, you should always transfer the entire parameter set with the print statements. To create a parameter set, use the function module GET_PRINT_PARAMETERS. The following topics describe this function module:

[GET_PRINT_PARAMETERS - Overview \[Page 1138\]](#)

[Import Parameters of GET_PRINT_PARAMETERS \[Page 1139\]](#)

[Export Parameters of GET_PRINT_PARAMETERS \[Page 1140\]](#)

[Exception Parameters of GET_PRINT_PARAMETERS \[Page 1141\]](#)

[How to Use GET_PRINT_PARAMETERS \[Page 1142\]](#)

GET_PRINT_PARAMETERS - Overview

The function module GET_PRINT_PARAMETERS has the following tasks:

- Creating a complete set of print and archiving parameters.
The individual print and archiving parameters are tightly interconnected and must be complete. For example, for each output device, you must specify the layout format, which, in turn, requires the numbers of lines and columns to be set. Or, when setting the archiving mode *Archive* or *Print and archive*, the archiving parameters must be set.
- Uncoupling the user dialog from the actual print statements.
The print statements (NEW-PAGE PRINT ON, SUBMIT <rep> TO SAP-SPOOL) support a user dialog, but with the disadvantage of not offering the *Back* function. After starting the print process using a print statement, the system cannot go back to before the print statement. The user can end the process only using *End* which terminates the entire program.

GET_PRINT_PARAMETERS executes the following functions:

- You use the import parameters to set the print and archiving parameters. The function module receives any required values not set via import parameters from the system. These values correspond to the pre-set values on the *Print List Output* dialog window, some of which are set in the user master record.
- The function module, by default, displays the *Print List Output* dialog window for a user dialog. Here, the user can overwrite the fields filled with import parameters or pre-set values.
- The function module automatically sets dependent values. If you set an import parameter, for example, for a certain layout, the system automatically sets the dependent parameters such as lines and columns instead of requesting them as import parameters.
- The function module supplies you with complete sets of print and archiving parameters as export parameters. You can transfer these export parameters to the spool system, using the options of the print statements. These parameter sets are either completely filled or completely empty.

Import Parameters of GET_PRINT_PARAMETERS

Import Parameters of GET_PRINT_PARAMETERS

The function module GET_PRINT_PARAMETERS has the same import parameters as the function module SET_PRINT_PARAMETERS (see [Print Parameters -Pre-setting Values \[Page 1135\]](#)), however, with the following exceptions:

GET_PRINT_PARAMETERS has no import parameter FOOT_LINE, since you need this parameter only if the user chooses *Print* on the output screen of the list.

GET_PRINT_PARAMETERS has the following additional import parameters:

- **MODE**

The following values influence the functionality of the module:

MODE	Effect
PARAMS	This is the default. The user can choose <i>Print</i> or <i>Cancel</i> on the dialog window.
PARAMSEL	The dialog window contains the additional checkbox <i>Selection cover page</i> . If the user fills this field (print parameter PRBIG), the system includes a cover page into the output that contains the selections of the selection screen.
DISPLAY	The print parameters in the dialog window are display-only.
CURRENT	Uses the function module to determine the print parameters during the current print process (after a print statement). These values correspond to the parameters set for printing. If no print process is in progress, the system uses the pre-set values.
BATCH	Uses the function module to determine the print parameters for a background job. The executable program (report) to be started must be specified in the import parameter REPORT. If the program's REPORT statement contains the options LINE-COUNT and LINE-SIZE, the system uses them as pre-settings in the dialog window. Instead of the <i>Print</i> pushbutton, the system offers the <i>Save</i> pushbutton in the dialog window.

- **REPORT**

The value contained in REPORT always influences the value pre-set for the name in the spool request (component PLIST) which, otherwise, is determined by the SY-REPID system field. This value is itself overwritten by the import parameter LIST_NAME (if used).

If MODE is set to 'BATCH', the value in REPORT specifies the name of the report you want to start as a background job. GET_PRINT_PARAMETERS determines the print parameters for that report and not for the current report.

- **NO_DIALOG**

determines whether to display the dialog window. If NO_DIALOG contains 'X', the system suppresses the dialog.

Export Parameters of GET_PRINT_PARAMETERS

The function module GET_PRINT_PARAMETERS has the following export parameters:

- **OUT_PARAMETERS**
This parameter either contains a complete set of print parameters or it is empty (see VALID).
- **OUT_ARCHIVE_PARAMETERS**
This parameter either contains a complete set of archiving parameters or it is empty (see VALID).
- **VALID**
This parameter shows whether the sets OUT_PARAMETERS and OUT_ARCHIVE_PARAMETERS are completely filled or empty. If VALID contains 'X', the parameter sets are complete. You can transfer them to the spool system. If VALID contains SPACE, the parameter sets are empty. VALID is set to SPACE if the user cancels the user dialog. Therefore, after a user dialog, always check VALID. If no user dialog occurs, VALID contains 'X'.

Exception Parameters of GET_PRINT_PARAMETERS

Exception Parameters of GET_PRINT_PARAMETERS

The function module GET_PRINT_PARAMETERS has the following exception parameters:

- **ARCHIVE_INFO_NOT_FOUND**
The archiving data specified are not consistent or the archive specified does not exist in the system.
- **INVALID_PRINT_PARAMS, INVALID_ARCHIVE_PARAMS**
The set of print or archiving parameters is invalid. You created an invalid parameter set by directly assigning values to the individual components of the parameter field strings and by using these structures to fill the import parameter IN_PARAMETERS or IN_ARCHIVE_PARAMETERS. The parameter field strings must always be the result of a previous call of GET_PRINT_PARAMETERS. Invalid import parameters, such as lines or columns set to 0, also create invalid parameter sets.

How to Use GET_PRINT_PARAMETERS

The function module GET_PRINT_PARAMETERS is the only method allowed by ABAP to assign values to the print and archiving parameter sets. Transferring parameter sets filled using GET_PRINT_PARAMETERS to the spool system prevents the program from being ended abnormally. This is of special importance for background processing. However, you must ensure that the export parameter VALID is unequal to SPACE and that no exception occurred.

Note that for GET_PRINT_PARAMETERS, it is most important that the parameter sets are complete and that the system can therefore execute the print request.

GET_PRINT_PARAMETERS does not execute a complete consistency check as does the *Print List Output* dialog window. It provides consistency only where it is needed to execute the print request. Inconsistent entries are partly ignored, partly replaced. For example, you can

- set values using the import parameters LINE_SIZE, LINE_COUNT that do not go with the LAYOUT parameter. In a user dialog, the system finds this inconsistency. Without user dialog, these values may cause truncated printouts.
- set an invalid value in the import parameter DESTINATION and, at the same time, set IMMEDIATELY to 'X'. In this case, the function module replaces the output device by a default value (LP01) and sets the component PRIMM to SPACE. This causes the spool system to store the request using the settings that go with the default printer.

To include the function module into your program, use *Edit → Insert statement...CALL FUNCTION* in the ABAP Editor.

It is reasonable to call the function module GET_PRINT_PARAMETERS several times in succession. For example, you can use GET_PRINT_PARAMETERS at the beginning of the program to trigger a user dialog, prompting the user for the basic setting. You then use the export parameters OUT_PARAMETERS and OUT_ARCHIVE_PARAMETERS as import parameters for further calls of the function module, in which you modify certain parameters from within the program (for example, to print wide lists in landscape format and narrow lists in portrait format).

For examples using GET_PRINT_PARAMETERS, see [Printing from within the Program \[Page 1146\]](#) and [Printing Lists of Called Executable Programs \(Reports\) \[Page 1150\]](#).

Executing and Printing

The easiest way of printing a list while creating it, is for the user to choose *Execute + print* on the report's selection screen. The user can choose between displaying the list on the screen (choosing *Execute*) or printing it directly without displaying it (choosing *Execute + print*).

If the user chooses *Execute + print* on the selection screen of the report, the system displays the *Print List Output* dialog window before creating the list. The user enters the print parameters. You can use the function module SET_PRINT_PARAMETERS to pre-set values for this dialog window (see [Print Parameters -Pre-setting Values \[Page 1135\]](#)).

Consequently, you must program the list in such a ways that it can both be displayed and printed. Therefore, in the REPORT statement, do not specify the page width wider than 132 characters (LINE-SIZE option) and better omit setting the page length (LINE-COUNT option).

Using *Execute + print*, the user can print only the basic list of the report. To print secondary lists that you create during interactive events on the displayed list, use NEW-PAGE PRINT ON (see [Printing from within the Program \[Page 1146\]](#)).

```
REPORT SAPMZTST NO STANDARD PAGE HEADING LINE-COUNT 0(2).
```

```
PARAMETERS P TYPE I.
```

```
INITIALIZATION.
```

```
CALL FUNCTION 'SET_PRINT_PARAMETERS'
```

```
EXPORTING
```

```
  ARCHIVE_MODE = '3'
```

```
  COPIES       = '5'
```

```
  DEPARTMENT   = 'BASIS'
```

```
  DESTINATION  = 'LT50'
```

```
  EXPIRATION   = ''
```

```
  IMMEDIATELY  = 'X'
```

```
  LAYOUT       = 'X_65_132'
```

```
  LINE_COUNT   = 54
```

```
  LINE_SIZE    = 20
```

```
  LIST_NAME    = 'Test'
```

```
  LIST_TEXT    = 'Test for User"s Guide'
```

```
  NEW_LIST_ID  = 'X'
```

```
  RECEIVER     = 'KELLERH'
```

```
  RELEASE      = ''
```

```
  SAP_COVER_PAGE = 'X'
```

```
START-OF-SELECTION.
```

```
DO P TIMES.
```

```
WRITE / SY-INDEX.
```

```
ENDDO.
```

```
TOP-OF-PAGE.
```

```
WRITE: 'Page', SY-PAGNO.
```

```
ULINE.
```

END-OF-PAGE.

ULINE.

WRITE: 'End of', SY-PAGNO.

After executing this program, the user can enter a value for parameter P on the selection screen (for example 100) and choose *Execute + print*. The system then displays this dialog window:

The function module SET_PRINT_PARAMETERS fills the input fields with pre-set values. Due to the function module, the field *Lines* is input-enabled, even though the REPORT statement contains the LINE-COUNT option. The option, in this case, is needed to reserve space for two footer lines.

After choosing *Print* on the *Print List Output* dialog window, the system displays the *Archive Parameters* dialog window, since the import parameter ARCHIV_MODE sets the archiving mode to *Print and archive*.

If the user enters 100 for the parameter P on the selection screen, the system creates an SAP cover page and two print pages that look like this.

First page:

Page 1

1

Executing and Printing

```
      2
      3
.....
     49
     50
-----
End of  1
Second page:
Page  2
-----
     58
     59
     60
.....
     99
    100
-----
End of  2
```

On each page, you can output up to 54 lines, including page header and footer lines. Note that the system triggers page breaks and creates page headers and footers exactly as described in [Creating Complex Lists \[Page 938\]](#).

If the user chooses *Execute* on the selection screen instead of *Execute + print*, the system displays the list as one page and without page footer on the output screen.

Printing from within the Program

To start the printing process from within the program while creating a list, use the NEW-PAGE statement with the PRINT ON option:

Syntax

```
NEW-PAGE PRINT ON [NEW-SECTION]
    [<params> | PARAMETERS <pripar>]
    [ARCHIVE PARAMETERS <arcpar>]
    [NO DIALOG].
```

This statement causes all subsequent output to be placed on a new page (see [Unconditional Page Break \[Page 964\]](#)) and the system interprets any output statements following NEW-PAGE PRINT ON as print statements. In other words, starting from NEW-PAGE PRINT ON, the system no longer creates the list for display but for the spool system.

If you use the NEW-PAGE PRINT ON statement without the NEW-SECTION option while already creating a list for the spool system, the statement is of no effect.

If you use the NEW-SECTION option, you reset pagination (SY-PAGNO system field) to 1. If the system already creates a list for the spool system, NEW-SECTION may have two effects:

- If the print parameters specified match the parameters of the currently created list and the print parameter PRNEW equals SPACE, the system does not create a new spool request.
- If the print parameters specified do not match the parameters of the currently created list or the print parameter PRNEW is unequal to SPACE, the system closes the current spool request and creates a new spool request.

The other options determine the print parameters (see below).

The end of a processing block (events during data retrieval or interactive events) automatically ends the print process. To explicitly end creating a list for the spool system, use the PRINT OFF option of the NEW-PAGE statement:

Syntax

```
NEW-PAGE PRINT OFF.
```

This statement creates a page break and sends the last page to the spool system. Any output statements following this statement appear in the list on the output screen.

Determining Print Parameters

To determine the print parameters for printed output following the NEW-PAGE PRINT ON statement, use the options of the statement.

You can use several options <params> to specify each print parameter (for example DESTINATION <dest>). The keyword documentation explains each option. Use the NO DIALOG option to tell the system whether to display or suppress the *Print List Output* dialog window. This method of setting print parameters is disadvantageous, since the system does not check whether the parameters specified are complete. Incomplete print parameters are detected only if you use the *Print List Output* dialog window. However, for background jobs, this is not possible. If the print parameters are incomplete and you use the NO DIALOG option, the system sends a warning after the syntax check, but it does not stop processing. This may cause unpredictable results when executing the program.

Printing from within the Program

SAP, therefore, recommends not to use the <params> options. Use the PARAMETERS option instead, and the ARCHIVE PARAMETERS option if necessary. To create the corresponding arguments <pripar> and <arcpa>, use the export parameters of the function module GET_PRINT_PARAMETERS (see [Setting Print Parameters from within the Program \[Page 1137\]](#)). This is the only method that guarantees completely set parameters and an executable print request. Since the function module GET_PRINT_PARAMETERS has its own user dialog, always use the NO DIALOG option in the NEW-PAGE PRINT ON statement.

```
REPORT SAPMZTST NO STANDARD PAGE HEADING.
```

```
DATA: VAL,  
      PRIPAR LIKE PRI_PARAMS,  
      ARCPAR LIKE ARC_PARAMS,  
      LAY(16), LINES TYPE I, ROWS TYPE I.
```

```
CALL FUNCTION 'GET_PRINT_PARAMETERS'  
  IMPORTING  
    OUT_PARAMETERS = PRIPAR  
    OUT_ARCHIVE_PARAMETERS = ARCPAR  
    VALID = VAL  
  EXCEPTIONS  
    ARCHIVE_INFO_NOT_FOUND = 1  
    INVALID_PRINT_PARAMS = 2  
    INVALID_ARCHIVE_PARAMS = 3  
    OTHERS = 4.
```

```
IF VAL <> SPACE AND SY-SUBRC = 0.  
  SET PF-STATUS 'PRINT'.  
  WRITE ' Select a format!'.  
ENDIF.
```

```
TOP-OF-PAGE DURING LINE-SELECTION.  
WRITE: 'Page', SY-PAGNO.  
ULINE.
```

```
AT USER-COMMAND.  
CASE SY-UCOMM.  
  WHEN 'PORT'.  
    LAY = 'X_65_80'.  
    LINES = 60.  
    ROWS = 55.  
    PERFORM FORMAT.  
  WHEN 'LAND'.  
    LAY = 'X_65_132'.  
    LINES = 60.  
    ROWS = 110.  
    PERFORM FORMAT.  
ENDCASE.
```

```
FORM FORMAT.  
CALL FUNCTION 'GET_PRINT_PARAMETERS'  
  EXPORTING  
    IN_ARCHIVE_PARAMETERS = ARCPAR  
    IN_PARAMETERS = PRIPAR  
    LAYOUT = LAY
```

```

LINE_COUNT    = LINES
LINE_SIZE     = ROWS
NO_DIALOG     = 'X'
IMPORTING
OUT_ARCHIVE_PARAMETERS = ARCPAR
OUT_PARAMETERS    = PRIPAR
VALID            = VAL
EXCEPTIONS
ARCHIVE_INFO_NOT_FOUND = 1
INVALID_PRINT_PARAMS  = 2
INVALID_ARCHIVE_PARAMS = 3
OTHERS              = 4.
IF VAL <> SPACE AND SY-SUBRC = 0.
  PERFORM LIST.
ENDIF.
ENDFORM.

FORM LIST.
NEW-PAGE PRINT ON
NEW-SECTION
PARAMETERS PRIPAR
ARCHIVE PARAMETERS ARCPAR
NO DIALOG.
DO 440 TIMES.
  WRITE (3) SY-INDEX.
ENDDO.
ENDFORM.

```

This program immediately calls the function module GET_PRINT_PARAMETERS without passing import parameters. On the *Print List Output* dialog window, the user can enter the print and archiving parameters for this program. The system passes these parameters, using the export parameters of the function module, to the field strings PRIPAR and ARCPAR. To guarantee that the parameters are complete and consistent, the program executes the user dialog via the dialog window and checks the return value of VALID.

After completing the dialog, the system displays the following basic list:



In the status PRINT of the basic list, the function codes PORT and LAND are assigned to the function keys F5 and F6, and to two pushbuttons of the application toolbar (see [Defining Individual User Interfaces \[Page 1051\]](#)). If the user chooses one of these functions, the AT USER-COMMAND event occurs, assigning to the variables LAY, LINES, and ROWS the values for portrait or landscape output format and calling the subroutine FORMAT.

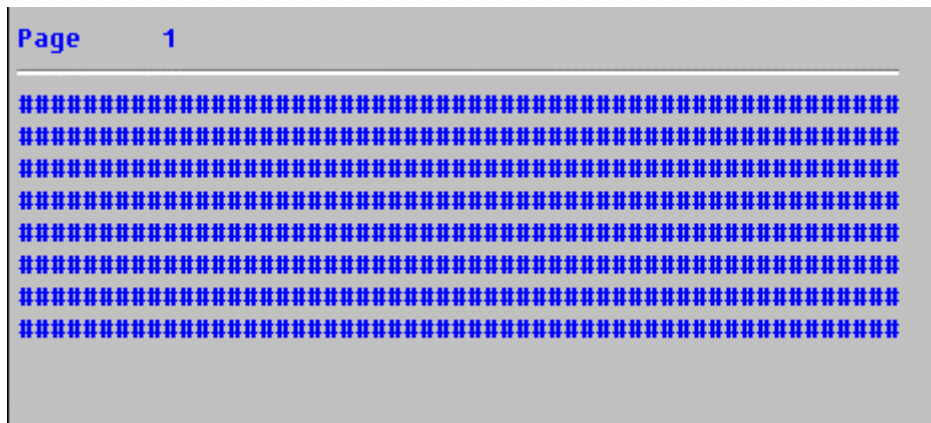
The subroutine FORMAT calls the function module GET_PRINT_PARAMETERS, passing the previously determined parameters PRIPAR and ARCPAR as import parameters. To the import parameters LAYOUT, LINE_COUNT, and LINE_SIZE, the program assigned the values stored in LAY, LINES, and ROWS. No user dialog occurs. The system returns the parameters to the field strings PRIPAR and

Printing from within the Program

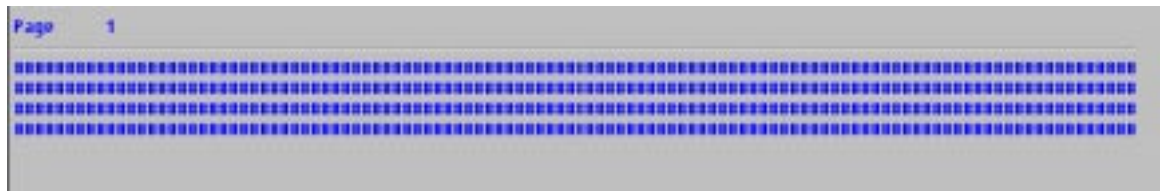
ARCPAR. The function of the subroutine call is to set the components PAART, LINCT, and LINSZ of the structure PRIPAR with new values.

After checking the parameters for completeness and consistency, the program calls the subroutine LIST. This subroutine sends a list to the spool system using NEW-PAGE PRINT ON, thereby determining the print and archiving parameters using PRIPAR and ARCPAR. A user dialog is not necessary, since all settings required were made by GET_PRINT_PARAMETERS.

To view the stored spool requests, the user can choose *System* → *Services* → *Print requests*. After choosing *PORTRAIT*, the spool request may look like this:



After choosing *LANDSCAPE*, however, the spool request looks like this:



Printing Lists of Called Executable Programs (Reports)

To send the output of reports you call from within your program using SUBMIT to the spool system, you must include the TO SAP-SPOOL option into the SUBMIT statement:

Syntax

```
SUBMIT <rep> TO SAP-SPOOL
    [<params>|SPOOL PARAMETERS <pripar>]
    [ARCHIVE PARAMETERS <arcpa>]
    [WITHOUT SPOOL DYNPRO].
```

For the description of the SUBMIT statement, see [Calling Executable Programs \(Reports\) \[Page 1113\]](#). Using the TO SAP-SPOOL option causes the system to format the lists of the called executable program (report) for print output while creating them and to send them to the spool system. Use the other options to determine the print parameters.

Determining Print Parameters

To determine the print parameters, proceed as described with the NEW-PAGE PRINT ON statement (see [Printing from within the Program \[Page 1146\]](#)):

You may set each print parameter individually using one of the <params> options (see keyword documentation) or use the SUBMIT statement to execute a user dialog; however, to determine the print parameters, use the function module GET_PRINT_PARAMETERS only (see [Setting Print Parameters from within the Program \[Page 1137\]](#)). The function module GET_PRINT_PARAMETERS uncouples the user dialog from the SUBMIT statement and guarantees a complete parameter set even without executing the user dialog. To determine the parameters, use only the options SPOOL PARAMETERS and ARCHIVE PARAMETERS and, to suppress the user dialog of the SUBMIT statement, use the WITHOUT SPOOL DYNPRO option.

The following executable program (report) is connected to the logical database F1S:

```
REPORT SAPMZTS1.
TABLES SPFLI.
GET SPFLI.
NEW-LINE.
WRITE: SPFLI-MANDT, SPFLI-CARRID, SPFLI-CONNID,
SPFLI-CITYFROM, SPFLI-AIRPFROM, SPFLI-CITYTO,
SPFLI-AIRPTO, SPFLI-FLTIME, SPFLI-DEPTIME, SPFLI-ARRTIME,
SPFLI-DISTANCE, SPFLI-DISTID, SPFLI-FLTYPE.
```

The following program calls SAPMZTS1 and sends the output to the spool system:

```
REPORT SAPMZTST NO STANDARD PAGE HEADING.
DATA: VAL,
      PRIPAR LIKE PRI_PARAMS,
      ARCPAR LIKE ARC_PARAMS.
CALL FUNCTION 'GET_PRINT_PARAMETERS'
  EXPORTING
    LAYOUT      = 'X_65_132'
```

Printing Lists of Called Executable Programs (Reports)

```

LINE_COUNT    = 65
LINE_SIZE     = 132
IMPORTING
OUT_PARAMETERS = PRIPAR
OUT_ARCHIVE_PARAMETERS = ARCPAR
VALID         = VAL
EXCEPTIONS
ARCHIVE_INFO_NOT_FOUND = 1
INVALID_PRINT_PARAMS   = 2
INVALID_ARCHIVE_PARAMS = 3
OTHERS                = 4.

```

```

IF VAL <> SPACE AND SY-SUBRC = 0.
  SUBMIT SAPMZTS1 TO SAP-SPOOL
  SPOOL PARAMETERS PRIPAR
  ARCHIVE PARAMETERS ARCPAR
  WITHOUT SPOOL DYNPRO.
ENDIF.

```

After starting the program, the function module GET_PRINT_PARAMETERS triggers a user dialog, displaying the area *Output format* of the *Print List Output* dialog window filled with the values from the import parameters:

Output format	
Lines	65
Columns	132
Format	X_65_132
At least 65 rows by 132 columns	

After the user has entered and confirmed the print parameters, the program calls SAPMZTS1, passing the export parameters of GET_PRINT_PARAMETERS as print and archiving parameters. SAPMZTS1 creates neither a screen display nor a user dialog. It sends the created list directly to the spool system. The user can view the stored spool request choosing *System* → *Services* → *Print requests*. Using the output format specified above, the spool request may look like this:

002 AA	0017 NEW YORK	JFK SAN FRANCISCO	SFO	06:01:00	13:30:00	16:31:00	2.572	MLS
002 AA	0064 SAN FRANCISCO	SFO NEW YORK	JFK	05:21:00	09:00:00	17:21:00	2.572	MLS
002 DL	1699 NEW YORK	JFK SAN FRANCISCO	SFO	06:22:00	17:15:00	20:37:00	2.572	MLS
002 DL	1984 SAN FRANCISCO	SFO NEW YORK	JFK	05:25:00	10:00:00	18:25:00	2.572	MLS
002 LH	0400 FRANKFURT	FRA NEW YORK	JFK	08:24:00	10:10:00	11:34:00	6.162	KM
002 LH	0402 FRANKFURT	FRA NEW YORK	JFK	01:35:00	13:30:00	15:05:00	6.162	KM
002 LH	0454 FRANKFURT	FRA SAN FRANCISCO	SFO	12:20:00	10:10:00	12:30:00	9.096	KM
002 LH	0455 SAN FRANCISCO	SFO FRANKFURT	FRA	13:30:00	15:00:00	10:30:00	9.096	KM
002 LH	2402 FRANKFURT	FRA BERLIN	SXF	01:05:00	10:30:00	11:35:00	430	KM
002 LH	2407 BERLIN	TXL FRANKFURT	FRA	01:05:00	07:10:00	08:15:00	430	KM
002 LH	2415 BERLIN	SXF FRANKFURT	FRA	01:05:00	09:25:00	10:30:00	430	KM
002 LH	2436 FRANKFURT	FRA BERLIN	THF	01:05:00	17:30:00	18:35:00	430	KM
002 LH	2462 FRANKFURT	FRA BERLIN	TXL	01:05:00	06:30:00	07:35:00	430	KM
002 LH	2463 BERLIN	SXF FRANKFURT	FRA	01:05:00	21:25:00	22:30:00	430	KM
002 LH	3577 ROM	FCO FRANKFURT	FRA	02:00:00	07:05:00	09:05:00	957	KM
002 SQ	0026 FRANKFURT	FRA NEW YORK	JFK	08:20:00	08:30:00	09:50:00	3.851	MLS
002 UA	0007 NEW YORK	JFK SAN FRANCISCO	SFO	06:10:00	14:45:00	17:55:00	2.572	MLS
002 UA	0941 FRANKFURT	FRA SAN FRANCISCO	SFO	12:36:00	14:30:00	21:06:00	5.685	MLS
002 UA	3504 SAN FRANCISCO	SFO FRANKFURT	FRA	13:35:00	15:00:00	10:30:00	5.685	MLS

Print Control

You can manipulate the output of a list during the print process from within the executable program (report). The statements SET MARGIN and PRINT-CONTROL, described in the topics below, take effect only if the list is directly sent to the spool. The statements do not affect a list that is displayed on the screen and printed from there using *List → Print*.

[Determining Left and Upper Margins \[Page 1153\]](#)

[Determining the Print Format \[Page 1155\]](#)

[Indexing Print Lists for Optical Archiving \[Page 1159\]](#)

Determining Left and Upper Margins

Determining Left and Upper Margins

To determine the size of the left and of the upper margin of a print page, use this statement:

Syntax

SET MARGIN <x> [<y>].

This statement causes the current print page to be sent to the spool system with <x> columns of space on the left page margin and, if specified, with <y> lines of space on the upper page margin.

The statement sets the contents of the system fields SY-MACOL and SY-MAROW to <x> and <y>. For printouts, these system fields determine the number of columns on the left margin and the number of lines on the upper margin.

The values you set apply to all subsequent pages, until you use another SET MARGIN statement. If you specify more than one SET MARGIN statement on one page, the system always uses the last one.

The following executable program (report) is connected to the logical database F1S.

```
REPORT SAPMZTST LINE-SIZE 60.
```

```
TABLES SPFLI.
```

```
SET MARGIN 5 3.
```

```
GET SPFLI.
```

```
WRITE: / SPFLI-CARRID, SPFLI-CONNID, SPFLI-CITYFROM,  
       SPFLI-AIRPFROM, SPFLI-CITYTO, SPFLI-AIRPTO.
```

If, after starting the report, the user chooses *Execute* on the selection screen, the following list appears on the output screen:

SPFLI				1
AA	0017	NEW YORK	JFK SAN FRANCISCO	SFO
AA	0064	SAN FRANCISCO	SFO NEW YORK	JFK
DL	1699	NEW YORK	JFK SAN FRANCISCO	SFO
DL	1984	SAN FRANCISCO	SFO NEW YORK	JFK
LH	0588	FRANKFURT	FRA NEW YORK	JFK

The SET MARGIN statement has no effect on the display.

If, after starting the report, the user chooses *Execute + print* on the selection screen, the list is printed while it is created. The user can view the stored spool request using *System → Services → Print requests*:

Determining Left and Upper Margins

1996/03/12			SPFLI	1
AA	0017	NEW YORK	JFK SAN FRANCISCO	SFO
AA	0064	SAN FRANCISCO	SFO NEW YORK	JFK
DL	1699	NEW YORK	JFK SAN FRANCISCO	SFO
DL	1984	SAN FRANCISCO	SFO NEW YORK	JFK
LH	0100	FRANKFURT	FRF NEW YORK	JFK

The list is shifted to the right by five columns and to the bottom by three lines.

Determining the Print Format

Determining the Print Format

To determine the print format, use the PRINT-CONTROL statement:

Syntax

PRINT-CONTROL <formats> [LINE <lin>] [POSITION <col>].

Without using the options LINE and POSITION, this statement sets the print format specified in <formats> for all characters that are printed starting from the current output position (system fields SY-COLNO and SY-LINNO). The LINE option sets the print format to start from line <lin>. The POSITION option sets the print format to start from column <pos>.

In <formats>, you can specify several different print formats. The system converts the values into a printer-independent code, the so-called print-control code. When printing, the system translates the print-control code to printer-specific control characters of the selected printer.

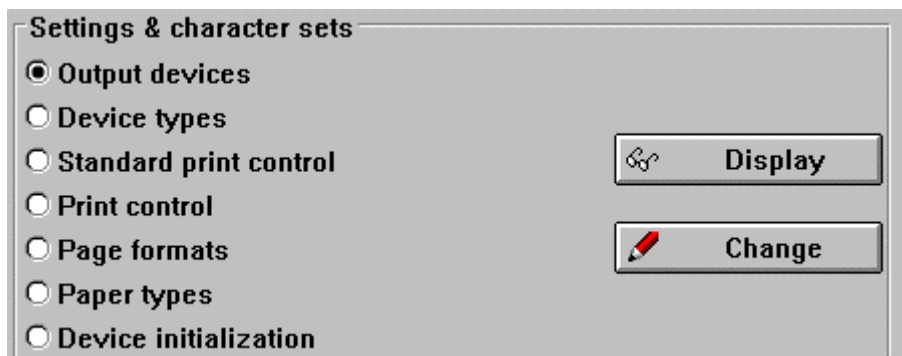
The table below lists the valid <formats> options and the corresponding print-control codes:

<formats>	Code	Description
CPI <cpi>	CI<cpi>	Characters per inch
LPI <lpi>	LI<lpi>	Lines per inch
COLOR BLACK	CO001	Color black
COLOR RED	CO002	Color red
COLOR BLUE	CO003	Color blue
COLOR GREEN	CO004	Color green
COLOR YELLOW	CO005	Color yellow
COLOR PINK	CO006	Color pink
LEFT MARGIN <lfm>	LM<lfm>	Space from the left margin
FONT <fnt>	FO<fnt>	Font
FUNCTION <code>	<code>	For directly specifying a code

There are many more print-control codes than <formats> options. Therefore, you can specify any print-control code directly using the FUNCTION option.

Use the print formats only to set formats that are either not possible or not reasonable when formatting output for the output screen (for example, size specifications or fonts). For any other formats, use the methods described in [Formatting Options \[Page 896\]](#) or [Formatting Output \[Page 995\]](#). These formats automatically apply for printing as well as for display (provided the specified printer supports them).

To find the codes supported by a certain printer, choose *Tools* → *Administration* → *Spool* → *Spool administration*. This takes you to the screen *Spool Administration* (transaction SPAD).



Choose the individual components to receive the following information:

Choose	To get
<i>Output devices</i>	a list of the installed printers, including the device types
<i>Device types</i>	a description of the device types
<i>Standard print control</i>	a list of the print-control codes
<i>Print control</i>	an assignment of printer-specific control characters to the print-control codes for each device type
<i>Page formats</i>	a list of page formats
<i>Paper types</i>	a list of valid formats
<i>Device initialization</i>	an assignment of formats for each device type

Display *Print control* to find the print-control codes available for the printer you want to use. From the displayed list, select the desired device type. A section from this table (T02DD) for a Post Script printer may look like this:

Determining the Print Format

Info

Device type: **POSTSCRIPT**
 Name: **PostScript-Printer**

PrCtl	V	H	D	Control char. seq.
COL7H	1	X		'292073686F77706172742073616F666660a28') showpart saoff\n(
COL7N	1	X		'292073686F77706172742073616F666660a28') showpart saoff\n(
COL7U	1	X		'292073686F77706172742073616F666660a28') showpart saoff\n(
S0000	1	X		
S4001	1	X		
S4004	1	X		
S<<<<	1	X		
S>>>>	1	X		
SABLD	1	X		'292073686F7770617274207361626c640a28') showpart sabld\n(
SAOFF	1	X		'292073686F77706172742073616F666660a28') showpart saoff\n(
SAULN	1	X		'292073686F7770617274207361756c6e0a28') showpart sauln\n(

Spool Administration: Additional Info on Print Control SAULN

Print control: **S....**

Comment:

S.... Print control for SAPscript

Begin underline

On the right, you can see whether printer-specific control characters are maintained for a print-control code. Choose *Info* to display additional information for the individual codes in a dialog window.

For detailed information on spool administration, see the documentation [BC Printing Guide \[Ext.\]](#).

The following Executable program (report) is connected to the logical database F1S.

REPORT SAPMZTST LINE-SIZE 60.

TABLES SPFLI.

PRINT-CONTROL FUNCTION: 'SABLD' LINE 1,
 'SAOFF' LINE 2,
 'SAULN' LINE 3.

GET SPFLI.

WRITE: / SPFLI-CARRID, SPFLI-CONNID, SPFLI-CITYFROM,
SPFLI-AIRPFROM, SPFLI-CITYTO, SPFLI-AIRPTO.

If in table T02DD, the printer control characters for the print-control codes SABLD, SAOFF, and SAULN are defined as in the figure above, the system formats the output as follows:

```
1996/03/13      SPFLI      1
-----
AA 0017 NEW YORK      JFK SAN FRANCISCO SFO
AA 0064 SAN FRANCISCO SFO NEW YORK      JFK
DL 1699 NEW YORK      JFK SAN FRANCISCO SFO
DL 1984 SAN FRANCISCO SFO NEW YORK      JFK
LH 0400 FRANKFURT     FRA NEW YORK      JFK
```

....

The print format for the first line is set to bold, using the print-control code SABLD. The print-control code SAOFF turns off bold style, starting from the second line. Using the print-control code SAULN, the system underlines all lines starting from line 3.

Indexing Print Lists for Optical Archiving

Indexing Print Lists for Optical Archiving

To include index lines for optical archiving into a list, use the PRINT-CONTROL statement as follows:

Syntax

PRINT-CONTROL INDEX-LINE <f>.

This statement writes the contents of field <f> into an index line after finishing the current print line. The system stores this index line in the spool file. The index line is not printed on a printer.

During optical archiving, the spool system divides the list into a data file and a description file. Data files contain the actual data information (print lists). Description files contain the index information.

The ArchiveLink Viewer provides the function *Attribute Search* to search for index lines in archived lists. This additional search can be important for performance when searching in very long lists. In order to enable the function *Edit → Attribute Search* in the ArchiveLink Viewer, you must insert specific index lines <f> that follow a special naming convention (for more information about this, see the topic "Indexing Print Lists" in the documentation [SAP ArchiveLink \[Ext.\]](#)).

```
REPORT BCARC001.
PARAMETERS NUMBER TYPE I.
DATA: INDEX TYPE I, SQUARE TYPE I, NUMB TYPE I, NUM(4),
      DKEY(100), DAIN(100).
DKEY = 'DKEYIndex'.
DKEY+44 = '0'.
DKEY+47 = '3'.
PRINT-CONTROL INDEX-LINE DKEY.
CLEAR DKEY.
DKEY = 'DKEYNumber'.
DKEY+44 = '3'.
DKEY+47 = '4'.
PRINT-CONTROL INDEX-LINE DKEY.
INDEX = 0.
DO NUMBER TIMES.
  INDEX = INDEX + 1.
  IF INDEX = 100.
    NUMB = SY-INDEX / 100.
    WRITE NUMB TO NUM LEFT-JUSTIFIED.
    CONCATENATE 'DAIN' 'IDX' NUM INTO DAIN.
    PRINT-CONTROL INDEX-LINE DAIN.
    INDEX = 0.
  ENDIF.
  SQUARE = SY-INDEX ** 2.
  WRITE: / SY-INDEX, SQUARE.
ENDDO.
```

Indexing Print Lists for Optical Archiving

By using a DO loop, this report writes a list of square numbers. The user can specify the number of loop passes in the input field NUMBER on the selection screen. All hundred lines, the PRINT-CONTROL statement creates a index line for optical archiving (DAIN line). Two DKEY lines at the beginning of the list define the structure of the DAIN lines (see the topic "Indexing Print Lists" in the documentation [SAP ArchiveLink \[Ext.\]](#)).

If the user chooses *Execute + print* on the selection screen and stores the print request in the spool system, the user can then view the request using *System → Services → Print requests*.

The beginning of the list looks as follows:

DKEYIndex	0	3	
DKEYNumber	3	4	
11/11/1996	Test for optical archiving		1
Number	Square		
1	1		
2	4		
3	9		
4	16		
5	25		
6	36		
7	49		
8	64		
9	81		
10	100		

The first two lines are DKEY index lines. They define the structure of the following DAIN index lines.

Around line one hundred, the list looks as follows:

94	8.836
95	9.025
96	9.216
97	9.409
98	9.604
99	9.801
DAINIDX1	
100	10.000
101	10.201
102	10.404
103	10.609
104	10.816
105	11.025
106	11.236
107	11.449
108	11.664

All hundred lines, a DAIN index line is inserted. The first DAIN index has the name 'Index'. It starts at position 0, has a length of 3, and contains the value 'IDX'. The second DAIN index has the name 'Number'. It starts at position 3, has a length of 4 and contains the value 1 in front of line 100 (it contains the value 2 in front of line 200 etc.).

Indexing Print Lists for Optical Archiving

Please refer to the section "Optical Archiving of ABAP Lists" in the documentation [SAP ArchiveLink Scenarios in Applications \[Ext.\]](#) for information about optical archiving of this list, retrieving the archived list, and using the *Attribute Search* function in the ArchiveLink Viewer.

Switching Between Dialog Screens and List Display

Often you want to produce a list from within your transaction. There are two ways you can do this.

- Submit a separate executable program (report)
Use the SUBMIT statement to start a separate executable program (report) directly from the transaction. Using the SUBMIT statement is described in [Submitting Executable Programs \(Reports\) \[Page 1334\]](#).
- Produce the list from your module pool using LEAVE TO LIST-PROCESSING

The LEAVE TO LIST-PROCESSING statement is the statement you use to produce a list from a module pool. This statement lets you switch from dialog-mode into list-mode within a dialog program. You can then code the list-processing logic you need right in the module pool.

At runtime, when a LEAVE TO LIST-PROCESSING statement is executed, the module pool still retains control of execution. The transaction's data area is completely available to the report-processing code, so no parameters must be passed back and forth.

When you branch into list-mode, you can produce lists using all the ABAP features available for interactive lists. This is particularly useful during PROCESS ON VALUE-REQUEST or PROCESS ON HELP-REQUEST processing, when you want to customize the display of field help or possible values.

See the following for information:

[Using LEAVE TO LIST-PROCESSING \[Page 1163\]](#)

[Using GUI Statuses in List-Mode \[Page 1165\]](#)

[Returning to Dialog Mode \[Page 1167\]](#)

[Returning to a Different Screen \[Page 1168\]](#)

For a concrete example of how to branch into list-mode, see transaction TZ70 (in development class SDWA, which is delivered with your system) and the discussion of it in this chapter:

[Example Transaction: Branching to List-Mode \[Page 1169\]](#)

Using LEAVE TO LIST-PROCESSING

Using LEAVE TO LIST-PROCESSING

To use LEAVE TO LIST-PROCESSING, place the statement in the code where you want list-mode processing to begin. This statement does two things:

- Switches to list-mode processing
From this point on, you can code the statements needed to issue and control the list (WRITE, ULINE, SKIP and so on). All the standard report-related events and features are available: AT LINE-SELECTION, function keys, basic and detail list levels, and so on.
- Sets the standard list output to be the “following screen” for the current screen
The system notes that the standard list-output should follow the display of the current screen. Depending on your application, you may want both displays, or you can inhibit the current screen and replace it with the list output.

Very often, it is cleaner to enclose all code for the list processing in a single subroutine. Example transaction TZ70 does this:

**** ABAP module and form: ****

```
MODULE PREPARE_LIST OUTPUT.
  LEAVE TO LIST-PROCESSING AND RETURN TO SCREEN 0.
  PERFORM EDIT_LIST.
  LEAVE SCREEN.
ENDMODULE.

FORM EDIT_LIST.
  SET PF-STATUS 'LIST'.
  SET TITLEBAR 'LST' WITH SFLIGHT-CONNID SFLIGHT-CARRID.

  NEW-PAGE LINE-SIZE 72.
  SELECT * FROM SFLIGHT WHERE   CARRID = SFLIGHT-CARRID
                                AND CONNID = SFLIGHT-CONNID.

  WRITE: /   SY-VLINE NO-GAP,
           SFLIGHT-FLDATE   COLOR 4 INTENSIFIED OFF NO-GAP,
           SY-VLINE NO-GAP,
           SFLIGHT-PRICE    COLOR 2 INTENSIFIED OFF NO-GAP,
           .....

ENDFORM.
```

How List-Mode in Dialog-Mode Works

At runtime, the module pool retains control of execution. You can code list-mode logic in PBO or PAI for the current screen. The choice depends on whether you want the list output to follow the current screen, or to replace it. In either case, the list appears when processing for the current screen terminates. (Screen processing terminates when control reaches either a LEAVE SCREEN statement or the end of PAI.)

- To display the list output *in addition to* the current screen:
Place the LEAVE TO LIST-PROCESSING logic at the end of PAI. Coded this way, the program can respond to requests for list output within the current PAI processing. On return from the list display, the system repeats processing for the current screen, starting with the beginning of PBO.

Using LEAVE TO LIST-PROCESSING

- To display the list output *instead of* the current screen:

Code the LEAVE TO LIST-PROCESSING logic in the PBO, and follow it with LEAVE SCREEN. This tells the system to display the list without displaying the current screen. PAI processing for the current screen is not executed.

For more information, see:

[Processing Screens in the Background \[Page 722\]](#)

[Example Transaction: Branching to List-Mode \[Page 1169\]](#)

Using GUI Statuses in List-Mode

Using GUI Statuses in List-Mode

You can use the standard GUI status for lists, or you can define your own interface. If you use the standard list status, the system automatically implements many GUI functions for you. For example:

- two return functions (BACK, CANCEL)
- the *Print* and *Search* functions
- the standard scrolling keys (P+, P++, P-, P--)

You do not need to program these functions in your own code. You can set the standard GUI status for lists using the command

SET PF-STATUS SPACE.

If you define your own status, you must do two things. First, activate the desired functions explicitly in the GUI status (by entering their function codes in the relevant pushbuttons and function codes). Second, in your ABAP module, code the logic for those functions not handled automatically by the system. This includes the F21-F24 keys, although the P-, P-, P+, and P++ functions are automatically defined.

Example transaction TZ70 defines its own status (named LIST). Since this status has type *List in dialog box*, only pushbuttons can be defined. (A menu bar with menus is not needed). The *Back* and *Cancel* exit functions (BACK und RW) appear automatically as pushbuttons, because the system implements these functions itself.



For a list of all functions in list-mode that are handled automatically by the system, do the following:

1. In the Menu Painter worksheet, select *Utilities* → *Explanations* → *Norms/suggestions*.
2. In the resulting popup, double-click on the *List functions* element of the hierarchy.

For information on interactive lists, see [Interactive Lists \[Page 1030\]](#).

Returning to Dialog Mode

Returning to Dialog Mode

There are two ways to return from list-mode to dialog-mode. In both cases, the place your program returns to is, by default, the screen that started list processing. (This is the screen containing the LEAVE TO LIST-PROCESSING statement.)

If you want to return to a different screen, see [Returning to a Different Screen \[Page 1168\]](#).

Your program continues to run in list-mode until one of the following occurs:

- The system reaches a LEAVE LIST-PROCESSING statement in your code.
The LEAVE LIST-PROCESSING statement returns control to the dialog screen. On return, the system re-starts processing at the beginning of PBO.
- The user requests BACK or CANCEL from the basic-list level of the executable program.
If the user exits the list using the BACK or CANCEL icons, you do not need to program an explicit LEAVE LIST-PROCESSING. BACK or CANCEL are standard return functions that are automatically handled by the system. When the user presses one of these, the system returns to the screen containing the LEAVE TO LIST-PROCESSING. Here the system re-starts PBO processing screen.

(Remember that the EXIT function, unlike BACK or CANCEL, is *not* handled by the system. If you want to offer the EXIT function, you must implement it yourself. In this case, implement it using a LEAVE LIST-PROCESSING statement.)

Example transaction TZ70 returns from list-mode using the second method. For screen 200, where the transaction branches into list mode, no exit functions are coded, since BACK and CANCEL are offered automatically by the system. For screen 100 however, where there is no list processing, the program must implement its own exit functions (BACK, EXIT, and CANCEL):

```
MODULE EXIT_0100 INPUT.  
  CASE OK_CODE.  
    WHEN 'CANC'.  
      CLEAR OK_CODE.  
      SET SCREEN 0. LEAVE SCREEN.  
    WHEN 'EXIT'.  
      CLEAR OK_CODE.  
      SET SCREEN 0. LEAVE SCREEN.  
    WHEN 'BACK'.  
      CLEAR OK_CODE.  
      SET SCREEN 0. LEAVE SCREEN.  
  ENDCASE.  
ENDMODULE.
```

Returning to a Different Screen

When returning to dialog-mode, your program can also re-route the user to a screen different from the one that started the list. To do this, use the keywords `AND RETURN TO SCREEN` when you first branch to list-mode:

```
LEAVE TO LIST-PROCESSING AND RETURN TO SCREEN 100.
```

With this statement, whenever the program returns to dialog mode (either because the user exits from a list, or because a `LEAVE LIST-PROCESSING` is executed), the system resumes processing at the PBO for the screen requested (here screen 100).

Transaction TZ70 uses `AND RETURN TO SCREEN 0` in the `PREPARE_LIST` module:

```
** PROCESSING FOR SCREEN 200**  
MODULE PREPARE_LIST OUTPUT.  
  LEAVE TO LIST-PROCESSING AND RETURN TO SCREEN 0.  
  PERFORM EDIT_LIST.  
  LEAVE SCREEN.  
ENDMODULE.
```

Here, returning to screen 0 closes the `CALL` mode started when screen 200 was triggered with `CALL SCREEN`. The system returns control to the last screen in the previous call mode (that is, in the PBO for screen 100).

Example Transaction: Branching to List-Mode

Example Transaction: Branching to List-Mode

Transaction TZ70 demonstrates one way to perform list processing inside a transaction.:

1. Screen 100 PAI: CALL SCREEN 200.
2. Screen 200 PBO: leave to list-processing, generate list, and leave screen 200.
 - No display of Screen 200 (inhibited by LEAVE SCREEN statement)
3. Screen 200 PAI: no coding needed (inhibited by LEAVE SCREEN)
 - Display of list output occurs:

Airline carrier	AA
Flight number	17

Flights for flight number 0017 company AA					
Date	Ticket price	Curr	Jet	Capacity	Occupied
30.01.1995	689,66	USD	747-400	660	10
01.02.1995	689,66	USD	747-400	660	20
01.06.1995	689,66	USD	747-400	660	38
04.06.1995	689,66	USD	747-400	660	38

- Return to PBO for screen 100 (because user presses either *Cancel* or *End*)
4. Screen 100 PBO: re-execute PBO...

Screen Flow Logic

The relevant parts of the flow logic for the two screens are:

```

*-----*
* Screen 100: Flow Logic (PAI only)                *
*&-----*
PROCESS AFTER INPUT.
  MODULE EXIT_0100 AT EXIT-COMMAND.
  MODULE USER_COMMAND_0100.

*-----*
* Screen 200: Flow Logic                            *
*&-----*
PROCESS BEFORE OUTPUT.
  MODULE PREPARE_LIST.
  
```

Example Transaction: Branching to List-Mode

*
PROCESS AFTER INPUT.

ABAP Code

The main PAI module for screen 100 calls screen 200, which will appear as a separate dialog box (popup):

```
*&-----*
*&   Module  USER_COMMAND_0100  INPUT
*&-----*
MODULE USER_COMMAND_0100 INPUT.
  CASE OK_CODE.
    WHEN SPACE.
      <...Select flight info for the requested flight..>
      CLEAR OK_CODE.
      CALL SCREEN 200 STARTING AT 10 5 ENDING AT 80 15.
    ENDCASE.
  ENDMODULE.
```

The PBO module for screen 200 sets up the branch to list-mode:

```
*&-----*
*&   Module  PREPARE_LIST  OUTPUT
*&-----*
MODULE PREPARE_LIST OUTPUT.
  LEAVE TO LIST-PROCESSING AND RETURN TO SCREEN 0.
  PERFORM EDIT_LIST.
  LEAVE SCREEN.
ENDMODULE.
```

The subroutine for putting out the list contains the following statements (shown partially):

```
*&-----*
*&   Form  EDIT_LIST
*&-----*
FORM EDIT_LIST.
  SET PF-STATUS 'LIST'.
  SET TITLEBAR 'LST' WITH SFLIGHT-CONNID SFLIGHT-CARRID.

  NEW-PAGE LINE-SIZE 72.
  SELECT * FROM SFLIGHT WHERE   CARRID = SFLIGHT-CARRID
                                AND   CONNID = SFLIGHT-CONNID.

  WRITE: /      SY-VLINE NO-GAP,
                                SFLIGHT-FLDATE   COLOR 4 INTENSIFIED OFF NO-GAP,
*                                .....
  ENDSELECT.
  IF SY-SUBRC = 0. ULINE. ENDIF.
ENDFORM.
```

The list-mode event TOP_OF_PAGE is coded for the list display:

```
TOP-OF-PAGE.
  ULINE.
  WRITE: /      SY-VLINE NO-GAP,
                                .....

```

Messages

Messages

Messages are another means of communicating with the user. Their principal use is to notify users of errors.

This section describes message handling on the different types of screen.

Handling Errors and Messages

When a user enters screen input, the transaction must check the input's validity before using it. The SAP System provides error-handling features that simplify field-checking as much as possible. These features include keywords for programming error-handling, as well as aspects of the dialog-processing runtime environment:

- automatic field checks (performed by the system)
Some field checks are performed automatically by the system, based on information stored in the ABAP Dictionary.
- the FIELD and CHAIN statements (in the flow logic language)
The FIELD and CHAIN flow logic statements let you program your own field checks. FIELD and CHAIN tell the system which fields you are checking, and whether the system should perform checks in the flow logic or call an ABAP module. If errors are found, the system enters an error dialog with the user.
- the MESSAGE statement (in ABAP)
The MESSAGE statement (in ABAP) lets you put out messages from an ABAP program. An ABAP program notifies the system of errors by putting out an error message or warning. In response, the system enters its error dialog with the user.
- an error dialog (performed by the system)
Errors can be detected either by the system or by an ABAP module. In either case, when an error is found, the system automatically re-displays the screen and puts out a message.

Errors are most often field-specific. In the re-display, the field (or fields) causing the error is input-enabled, and all other fields are disabled. The system places the cursor in the error field, and requires the user to re-enter the input. The field checks are then repeated.

The following topics provide information:

[Introduction to Error Processing \[Page 1173\]](#)

[Checking Screen Fields for Validity \[Page 1176\]](#)

[Making Module Calls Conditionally \[Page 1182\]](#)

[Issuing Messages \[Page 1186\]](#)

[Example Transaction: Checking Field Input \[Page 1192\]](#)

Introduction to Error Processing

Introduction to Error Processing

In normal dialog processing, the transaction progresses from one screen to the next. However, if an error occurs, the system redisplay the screen where the error occurred. A message is displayed, and if the error involved field input, the field is input-enabled. (All other fields retain fixed values.) How does this look to the user, and how does your program tell the ABAP processor that a re-display is necessary?

Look at transaction TZ31 for an example of error-handling. TZ31 (development class SDWA) is a small transaction for displaying and updating flight information. The transaction lets the system perform automatic field checking, but also contains its own logic to conduct additional error-checks.

Normally, when you use the transaction, you enter airline and flight identifiers and press ENTER. The system then displays all field details in update-mode. To make your changes, you type in the new information and save.

What happens when your input is wrong? Suppose you just pressed ENTER without typing in any of the required information. (Fields with "?" in them are required-input fields.) The system checks for this automatically, and sends you a message:

Change Flight Data

Flight data Edit Goto System Help

Other flight

Airline carrier ?
Flight number ?

Flight data

From city
Dep. airport
Destination
Dest. airport
Flight time 00:00:00
Departure 00:00:00
Arrival 00:00:00
Distance
Dist. in

Flight type

☒ Sched. flight
☐ Charter flight

E: Required entry not made

TZ31 also checks for things the system neglects. For instance, what if when updating the display, you enter an airport code that doesn't exist? The program sends you a message:

Other flight

Airline carrier ab
Flight number 17

Flight data

Introduction to Error Processing



E: Entry AB does not exist (please check your entry)

In this screen, only the airport field can be changed. All other input is fixed. When you then correct the airport and re-enter, the transaction continues with its other processing.

By experimenting with TZ31, you will see that several kinds of field checks take place. Some are handled automatically by the system, and some by the program:

- Do the fields that require input have input in them? (**automatic**)
The *Airline* and *Flight-number* fields have the required-input attribute in the Screen Painter. The system automatically checks that these fields get input from the user.
- Does the airline entered exist? (**automatic**)
In the Screen Painter, the *Airline* field is declared as the table field SPFLI-CARRID. In the Dictionary, the CARRID field has a foreign key relationship with check table SCARR. As a result, the system automatically checks that all input to SPFLI-CARRID is contained in SCARR.
- Does the flight-number exist for this airline? (**ABAP**)
An ABAP module (CHECK_FLIGHT) in transaction TZ31 checks that the flight number entered exists for the given airline.
- Departure/arrival cities: do they exist? (**automatic**)
The *Departure city* field (SPFLI-CITYFROM) is a foreign key, with check table SGEOCITY. The system automatically checks that the input to this field is found in the SGEOCITY table.
- Departure/arrival airports: do they exist? (**ABAP**)
The ABAP modules check that the airports entered exist.

The rest of this chapter tells you how to program error-handling:

[Checking Screen Fields for Validity \[Page 1176\]](#)

[Issuing Messages \[Page 1186\]](#)

For a discussion of how transaction TZ31 is implemented, see.

[Example Transaction: Checking Field Input \[Page 1192\]](#)

If you want to play with transaction TZ31 in the system, remember that you can use the Data Browser to find out valid flight numbers for a given airline code. To access the Data Browser, get into the Object Browser (in the Workbench) and select *Environment* → *Data Browser* → *Table contents*.

Checking Screen Fields for Validity

The R/3 System offers various methods for checking screen fields:

- Automatic field checks, performed by the system
The automatic field checks are the easiest to use, when they are suitable. The system makes these checks based on field information you can store in the ABAP Dictionary.
- Checks performed in the screen flow logic
You can specify some field checks in the flow logic for a screen. You do this with the flow logic statement `FIELD...VALUES...`, which specifies a list of possible values for a screen field. The system checks screen field input against these values without even entering an ABAP module. As a result, you can design and test a screen without having any ABAP code available.
- Checks performed in ABAP
When the automatic field checks and the `FIELD...VALUES...` flow logic statement are not flexible enough, you can code ABAP modules to perform field-checking. To trigger a call to the module, you code a `FIELD...MODULE...` flow logic statement in your screen flow logic.

See the following topics for more explanation:

[Understanding Automatic Field Checks \[Page 1177\]](#)

[Checking Fields in the Screen Flow Logic \[Page 1178\]](#)

[Checking Fields in ABAP \[Page 1179\]](#)

Understanding Automatic Field Checks

Understanding Automatic Field Checks

The system automatically carries out certain validity checks on screen fields. These checks are performed after the user enters input, and before any PAI processing has begun. The types of automatic checks performed are:

- Required input

In the Screen Painter, you can set a field's required-input (*Req.entry*) attribute. When you do, the system requires the user to enter input into the field before entering PAI processing.

- Proper data format

In the Screen Painter, each field has a data format. This format limits the kinds of input that can be valid. For example, a DATS field (a date field) is an 8-character string in YYYYMMDD format. All characters must be numbers. The substrings MM and DD must be less than or equal to 12 and 31 respectively. For the given month value entered, the system also checks that the day value is valid.

When the user enters input that does not match the data format requirements, the system re-displays the screen until the input is corrected.

- Valid value for the field

In the Dictionary, there are two ways to restrict field values to an allowable set. The field can have a **foreign key relationship** with another table, or its domain can specify a **fixed-value list** for the field. For foreign key fields, the system checks that the user's input value can be found in the related check table. For fields with defined fixed-value lists, the system ensures that the user's input value is one of the values in the list.

For foreign-key fields, you can store a predefined error message with the field.

If any field fails a validity test, the system re-displays the screen and requires the user to change his input. Once re-entered, all automatic checks are repeated for fields in which the user enters new input.

To extend the automatic field checks, you can define field checks of your own. For how to do this, see:

[Checking a Single Field \[Page 1180\]](#)

[Checking Multiple Fields \[Page 1181\]](#)

Sometimes you want to code processing that should happen *before* the system carries out any automatic checks. This processing usually handles exit requests. (For example, when the user wants to return from a transaction, there is no reason to make him enter input in required fields.) To code logic that should execute before the system performs its automatic field checks, use the flow logic command MODULE... AT EXIT-COMMAND. For information, see [Avoiding Automatic Field Checks \[Page 1185\]](#).

Checking Fields in the Screen Flow Logic

Use the FIELD...VALUES flow logic statement to check field values in screen flow logic. With this statement, you specify a list of possible values, and the system compares the field input against this list. Syntax for the statement is:

```
FIELD <screen field> VALUES (<value-list>)
```

The <value-list> can include individual values or value intervals. Surround all values with single quotes. For example:

```
FIELD SBOOK-CARRID VALUES ('AA', 'LH', 'US').
```

```
FIELD SBOOK-CONNID VALUES (BETWEEN '1' AND '10', NOT '3').
```

If the value entered by the user is not equal to one of the values in the list (or fit within a specified interval), the system redisplay the screen for new input.

The following table shows all possible formats for the <value-list>.

<i>Comparison</i>	<i>Syntax</i>
Single value	('<value>')
Complement of single values	(NOT '<value>')
Several single values, or complements	('<value1>', '<value2>', NOT '<value3>')
Value interval	(BETWEEN '<value>' AND '<value>')
Everything outside a certain interval	(NOT BETWEEN '<value>' AND '<value>')

To use the FIELD...VALUES option, the fields for comparison must have data type CHAR or NUMC. Also remember that all values given in single quotes should be capitalized.

Checking Fields in ABAP

Checking Fields in ABAP

To program field checks in ABAP, you use the FIELD and CHAIN flow logic language statements. The following form of the FIELD statement lets you call ABAP modules that make field checks:

FIELD <field> MODULE <module>.

The FIELD statement may contain more than one MODULE call:

FIELD <field>:	MODULE <module1>,
	MODULE <module2>.

You can also specify more than one field in a FIELD statement. This is especially useful when you want to chain field-checks together with the CHAIN statement. For example, both forms of the FIELD statement below are allowed:

```
CHAIN.  
  FIELD <field1>.  
  FIELD <field2>.  
  FIELD: <field3>, <field4>, ... <fieldn>.  
  MODULE <module1>.  
  MODULE <module2>.  
ENDCHAIN.
```

When an ABAP module finds an error, it puts out an error message or warning to notify the user. Issuing these messages alerts the system that an error dialog is needed. The system redisplay the screen, requiring the user to enter a new value for the erroneous field. All other fields are input-disabled. With the CHAIN statement, if an error is found, the screen is re-displayed with all fields in the chain input-enabled.

The following topics provide information:

[Checking a Single Field \[Page 1180\]](#)

[Checking Multiple Fields \[Page 1181\]](#)

[Making Module Calls Conditionally \[Page 1182\]](#)

[Avoiding Automatic Field Checks \[Page 1185\]](#)

Checking a Single Field

The FIELD...MODULE flow logic statement lets you tie validity-checking to particular screen fields. FIELD...MODULE triggers calls to ABAP modules that check particular fields and send error messages if errors are found. In this case, the system re-displays the screen, disabling input for all fields but the ones specified.

To notify the system that an error has been found, the ABAP module must put out either an error message (type E) or a warning (type W). For example:

**** **Screen flow logic:** ****

```
PROCESS AFTER INPUT.  
  FIELD SPFLI-AIRPFROM  
    MODULE CHECK_FR_AIRPORT.
```

**** **ABAP module:** ****

```
MODULE CHECK_FR_AIRPORT INPUT.  
  SELECT SINGLE * FROM SAIRPORT  
    WHERE ID = SPFLI-AIRPFROM.  
  IF SY-SUBRC NE 0. MESSAGE E003 WITH SPFLI-AIRPFROM. ENDIF.  
ENDMODULE
```

The error message causes the system to redisplay the screen and require new input from the user. In the re-display, only field SPFLI-AIRPFROM can take new input. All other fields are input-disabled.

For more information about error messages, see [Issuing Messages \[Page 1186\]](#).

How the FIELD statement transfers data

The system transfers data from the screen to the ABAP program only once per screen display. Normally this happens at the beginning of PAI processing, after the system has performed its automatic field checking.

However there are exceptions to this. Transfer of data for fields mentioned in a FIELD statement is delayed until execution actually reaches the FIELD statement. If the screen field occurs in more than one FIELD statement, its value is transferred only at the first FIELD statement in which the field occurs.

The time of transfer is important because you should not use a field in an ABAP module before it has been transferred from the screen. If you do, the ABAP field will contain the value it had before the user entered screen input. This value can be either the value left over from the last screen, or even an invalid value.

Using a field in multiple FIELD statements

Sometimes you need to specify the same field in multiple FIELD statements. This makes error processing slightly more complex. When a module finds an error, the system re-displays the screen and restarts PAI processing with the corrected input. However, the the system cannot simply restart with the FIELD statement containing the module. The field in error may also have occurred in some earlier FIELD or CHAIN statement. The system has a special procedure for determining where to restart processing in this case. For more information, see [Restarting PAI after an error dialog \[Page 1189\]](#).

Checking Multiple Fields

Checking Multiple Fields

Sometimes you want to check several fields as a group. To do this, include the fields in a FIELD statement, and enclose everything in a CHAIN-ENDCHAIN block. A CHAIN statement is used in example transaction TZ31:

**** Screen flow logic: ****

```
CHAIN.
  FIELD: SPFLI-CARRID, SPFLI-CONNID.
  MODULE CHECK_FLIGHT.
ENDCHAIN.
```

**** ABAP module: ****

```
MODULE CHECK_FLIGHT INPUT.
  SELECT SINGLE * FROM SPFLI
    WHERE CARRID = SPFLI-CARRID
    AND CONNID = SPFLI-CONNID.
  IF SY-SUBRC NE 0.
    MESSAGE E005 WITH SPFLI-CARRID SPFLI-CONNID.
  ENDIF.
* .....
ENDMODULE.
```

In a chain block, all fields are mutually dependent. When an error is found inside a chain, the screen is re-displayed, and all fields found anywhere in the chain are input-enabled. All non-chain fields remain disabled. After the user re-enters his input (to one of the chain fields), PAI is restarted and all statements in the chain are re-executed as a unit.

Chains can include any other permissible flow logic-language statements. Also, chains can include more than one FIELD statement. In general, all FIELD statements should occur at the beginning of the CHAIN block.

```
CHAIN.
  FIELD: A, B, C.
  FIELD: D, E, F.
  MODULE X.
  MODULE Y.
ENDCHAIN.
```

It is allowed, but not really sensible, to add a MODULE statement onto a FIELD statement contained in a CHAIN block:

```
CHAIN.
  FIELD f1.
  FIELD: f2, f3 MODULE m1.      "(No period after f3)
  MODULE m2.
ENDCHAIN.
```

If module m finds an error, it opens *all* chain fields for input in the re-display, not just f2 and f3. Using FIELD...MODULE in this way is only sensible when you are using one of the AT- or ON-conditions. For more information, see [Conditional CHAIN Statements \[Page 1184\]](#).

Making Module Calls Conditionally

You can place conditions on the module calls you make from a screen's flow logic. For example, you can specify that a module should only be called if a given field has a non-initial value in it:

FIELD X MODULE CHECK_FIELDX ON INPUT.

With the conditional forms of the FIELD statement, you can prevent unnecessary module calls. Particularly when you are updating table entries, conditional calls can improve performance substantially. The following topics provide information:

[Conditional FIELD Statements \[Page 1183\]](#)

[Conditional CHAIN Statements \[Page 1184\]](#)

[Avoiding Automatic Field Checks \[Page 1185\]](#)

Conditional FIELD Statements

The FIELD...MODULE flow logic statement becomes conditional when you add ON- and AT-conditions. Use the following conditions to specify when the module should be called:

- ON INPUT

The ABAP module is called only if the field contains a value other than its initial value. This initial value is determined by the field's data type: blanks for character fields, zeroes for numerics. If the user changes a field value back to its initial value, ON INPUT does not trigger a call. (Contrast this with the ON REQUEST call, which does trigger the in this case.)

- ON REQUEST

The ABAP module is called only if the user has entered a value in the field value since the last screen display. The value counts as changed even if the user simply types in the value that was already there.

In general, the ON REQUEST condition is triggered through any form of "manual input". The system considers the following ways of setting a field as manual input:

- Actual user input
- SET PARAMETER field input (both manual and automatic)
- HOLD DATA field input
- Parameter input for a parameter transaction (CALL TRANSACTION...USING)
- Global fields used to customize your system (these specify automatic settings for fields for certain fields)

Any of these fulfill the ON REQUEST condition and would trigger the module call.

- ON *-INPUT

The ABAP module is called if the user has entered a "*" in the first character of the field, and the field has the attribute **-entry* in the Screen Painter. You can use this option in exceptional cases where you want to check only fields with certain kinds of input.

Some of these conditions apply to FIELD statements alone, and others to FIELD statements inside CHAIN blocks. In particular, the ON- and AT-conditions have a special meaning in FIELD statements that contain multiple fields, but are not enclosed in a CHAIN block. See [Checking Multiple Fields \[Page 1181\]](#) for details.

Conditional CHAIN Statements

To make the module calls in a CHAIN conditional, there are two options:

- ON CHAIN-INPUT

Similar to ON INPUT. The ABAP module is called if *any one* of the fields in the chain contains a value other than its initial value (blanks or nulls).

- ON CHAIN-REQUEST

This condition functions just like ON REQUEST, but the ABAP module is called if *any one* of the fields in the chain changes value.

For example:

```
CHAIN.  
  FIELD: A, B, C.  
  FIELD: D, E, F.  
  MODULE X ON CHAIN-INPUT.  
  MODULE Y.  
ENDCHAIN.
```

Here, module X is called if any of the fields A, B, C, D, E, F has a value different from its initial value. Module Y however is always called. If Y finds an error, *all* six fields are re-opened for input during the error dialog.

To restrict a condition to a particular field, connect the MODULE statement to the relevant FIELD statement.

```
CHAIN.  
  FIELD: A, B, C MODULE X ON INPUT.  
ENDCHAIN.
```

In this example, module X is called only when the *last* field in the list (C) contains a non-initial value. However, if X finds an error, all three fields (A, B, C) are re-opened for input in the error dialog.

Sometimes you want to make a call depend on several fields in a chain, but not others. For clarity, it is simplest to break up the chain you are using and create separate chains for separate field combinations. In each case, you use ON CHAIN-INPUT or ON CHAIN-REQUEST. For example:

```
CHAIN.  
  FIELD: A, B, C MODULE X ON CHAIN-REQUEST.  
ENDCHAIN.  
CHAIN.  
  FIELD: A, B, D, E MODULE Y ON CHAIN-REQUEST.  
ENDCHAIN.
```


Avoiding Automatic Field Checks

Avoiding Automatic Field Checks

Sometimes you want the system to carry out certain processing logic before it performs automatic field checks. For example, if the user wants to exit from the screen, there is no reason to make him or her enter data in the required-input fields.

The flow logic keywords AT EXIT-COMMAND are a special addition to the MODULE statement in the flow logic. AT EXIT-COMMAND lets you call a module before the system executes the automatic field checks:

**** **Screen flow logic:** ****

```
PROCESS AFTER INPUT.  
  MODULE EXIT AT EXIT-COMMAND.
```

To use AT EXIT-COMMAND, you must assign a function type of E to the relevant function in the Menu Painter or Screen Painter. In the Screen Painter, call up the attributes for the desired pushbutton, and set the attribute *FctType* to E. In the Menu Painter, select *Goto* → *Function list*, and enter E in the *Type* column for each function code that should behave as an exit command.

Once you have defined a function as type E, you can use the AT EXIT-COMMAND option to tell the system to process all ABAP modules associated with this function before carrying out any field checks. The AT EXIT-COMMAND event will only be triggered when the user activates a function defined as a type E function.

**** **ABAP module:** ****

```
MODULE EXIT INPUT.  
  CASE OK_CODE.  
    WHEN 'NEW'.  
      CLEAR: SPFLI, OK_CODE.  
      LEAVE SCREEN.  
    WHEN 'CANC'.  
      CLEAR OK_CODE.  
      SET SCREEN 0. LEAVE SCREEN.  
  ENDCASE.  
ENDMODULE.
```

Normally, the MODULE...AT EXIT-COMMAND statement is intended for processing the exit commands BACK, EXIT and CANCEL. The ABAP module you code to process these commands should contain statements to exit from the screen or transaction, (for example LEAVE TO SCREEN 0).

If you do not terminate the screen or transaction in the AT EXIT-COMMAND module, the system continues flow logic processing as usual: first it carries out automatic field checks, then processes the PAI statements in sequential order.

Issuing Messages

An ABAP module lets the system know that an error has occurred by issuing error or warning messages. The message causes the system to enter an error dialog with the user. (The error dialog redisplay the screen and, in the case of an error messages, requires new input.)

Messages are pre-defined texts stored as development objects in the system. The text may contain variable sub-strings you can replace with real texts at runtime. Each message belongs to a message class.

To send messages in a module pool, you must:

- send the message itself (with MESSAGE) and specify the message class in the PROGRAM statement
- if necessary, create the message class and message in the system

The following sections provide information on using messages in ABAP:

[Sending a Message \[Page 1187\]](#)

[Creating a Message Class \[Page 1190\]](#)

[Creating Messages \[Page 1191\]](#)

Sending a Message

Sending a Message

In general, you send messages using the ABAP statement MESSAGE, and use the message type to signify the type of the error. For example, in the following:

```
IF SY-SUBRC NE 0.  
  MESSAGE E001.  
ENDIF.
```

the message number is 001, and the E is the message type (an error).

You can put five different message types (E, W, I, A, S) on the front of a message number. For example, with message number 001, you could specify:

E001	sends message 001 as a error
W001	sends message 001 as a warning
I001	sends message 001 as an information message
A001	sends message 001 as an abnormal termination message (A=abend)
S001	sends message 001 as a success message

When you put out a message, the error processing that takes place depends on the message type and context. For more information, see [Message Processing in Dialog-Mode \[Page 1188\]](#).

Specifying the Message Class

All messages belong to a message class. So if you issue messages in a program, you must also specify a message class. You do this in the PROGRAM statement in your TOP-module:

```
PROGRAM SAPMTZ31 MESSAGE-ID A&.
```

The MESSAGE-ID parameter specifies the two-character code (either alphabetic or numeric) for the message class. This class holds for all messages put out in the module pool. If you need to issue messages from other classes beside the one specified in the PROGRAM statement, give the different class after the message number:

```
MESSAGE E123(ZZ).
```

Remember though that user-defined development objects (such as a message class) should begin with Y or Z.

Adding to the Message Text

Message texts may contain up to four variable texts. When you create the message, you use an & in the text to stand for the variable:

```
Flight &1 &2 does not exist.
```

At runtime, the system replaces the strings &1 and &2 with texts provided in the MESSAGE statement. To specify the replacement texts, use the WITH parameter:

```
MESSAGE E005 WITH SPFLI-CARRID SPFLI-CONNID.
```

If the two variables above contain flight number and airline, the user gets the message:

```
Flight AA 177 does not exist.
```

Message Processing in Dialog-Mode

In normal dialog processing, a transaction works from screen to screen, processing PBO and PAI events for each.

When a message is issued, screen processing takes a different course, depending on the type of message. We've seen that for an error message, the system redisplay the screen without repeating PBO processing. In the re-display, a message appears in the status line (bottom of screen) and all fields except those causing the error are input-disabled.

In general, the message type determines the screen on which the message appears, and how the dialog proceeds. The differences are:

<i>Message Type</i>	<i>Message appears on</i>	<i>Meaning</i>
Error	Current screen	All screen fields mentioned in FIELD statements are input-enabled. The user must enter a new value. The system then restarts PAI processing for the screen using the new values.
Warning	Current screen	All screen fields mentioned in FIELD statements are input-enabled. However, the user need not enter new values. If new values are entered, the system restarts PAI processing for the screen using the new values. If no new values are entered (the user just presses ENTER), the system resumes PAI processing immediately after the MESSAGE statement
Information	Popup screen	The current screen is interrupted, and the system automatically displays the information message in a popup.
Success	Following screen	The success message is displayed after the PBO processing for the following screen. Success messages thus have no effect on processing of the screen in which they are generated
Abend	Current screen	When the user presses ENTER, the current process is interrupted. The system returns the user to the SAP main menu.

With error messages and warnings, the system restarts PAI processing for the screen after re-displaying it. However, it may not repeat all PAI processing. For information, see [Restarting PAI after an error dialog \[Page 1189\]](#).

Restarting PAI after an error dialog

Restarting PAI after an error dialog

When an ABAP module called with FIELD or CHAIN issues a warning or error message, the system redisplay the screen, gets new input from the user, and then restarts PAI processing. Where this restart takes place (that is, where in the PAI event) may not be immediately obvious. All FIELD or CHAIN statements must be repeated if they share fields with the FIELD or CHAIN statement that found the error. The system determines where to restart using the following steps:

1. In the flow logic, determine which fields:
 - are specified in the CHAIN or FIELD statement that raised the error AND
 - were updated by the user during the last screen display
2. Search through the PAI event for the first FIELD statement that mentions any of the fields in Step 1.
3. Restart processing at that FIELD statement, or if it occurs in a CHAIN, at the beginning of the CHAIN statement.

For example, in the following PAI event:

PROCESS AFTER INPUT.

```
FIELD A MODULE M.  
FIELD B MODULE N.
```

```
CHAIN.  
FIELD: A, B, C.  
FIELD: D, E, A.  
MODULE P.  
MODULE Q.  
ENDCHAIN.
```

```
CHAIN.  
FIELD: F.  
MODULE R.  
ENDCHAIN.
```

```
CHAIN.  
FIELD: D.  
MODULE S.           ==> ERROR !!!  
ENDCHAIN.
```

Module S raises an error. After re-displaying the screen, the system re-starts processing with module P.

Creating a Message Class

You can create a message class from two places in the system:

- from an *Object class* object list (in the Object Browser)
- from an ABAP module (in the ABAP editor)

The most natural method is from the ABAP module containing the PROGRAM statement.

1. Enter the MESSAGE-ID parameter value:
PROGRAM SAPMTZ31 MESSAGE-ID A&.
2. Double-click on the message-id name (here "A&").

The system responds with a dialog window asking whether you want to create the message class. Press Yes. You jump to the message-class screen.

The screenshot shows the 'Display Message Class' dialog box. The 'Message class' field contains 'A&'. The 'Development class' field contains 'SDWA'. The 'Last changed by' field contains 'KEHRERW'. The 'Change date' field contains '03.01.1996'. The 'Last changed at' field contains '12:07:17'. The 'Attributes' section shows 'Pflegesprache' set to 'D'. The 'Responsible' field contains 'HILLENBRAND'. The 'Short text' field contains 'Documentation: Dialog Programming - Examples'.

3. Enter a *Short text* description and press the Save icon.
The message class is created.

4. Press *Messages* to enter actual message texts.

See [Creating Messages \[Page 1191\]](#) for how to enter message texts.

Creating Messages

Creating Messages

There are several ways to jump to a message class to enter message texts. The most natural method is:

1. Code the MESSAGE statement in your program, using the message number you need.
MESSAGE E001.
If you don't know the next free message number, just enter any number.
2. Double-click on the message number.
The system jumps to the list of messages for the message class. Because message classes can be used by multiple users, it is always better to maintain single messages at a time. Thus, only the message number you requested is in update-mode.
If the message number you entered already exists in the system, you branch to it in display mode. In this case:
 - Press *Next free number* to jump to the next free message number.
 - Press *Individual maint.* to edit it.
3. Type in the message text and press the save icon.

Short text	Documentation: Dialog Programming - Examples
Person responsible	HILLENBRAND
Last changed by	KEHRERW
Date	03.01.1996

Message number		Self-explanator
001	Flight &1 &2 changed	<input type="checkbox"/>
002	Unable to save flight &1 &2	<input type="checkbox"/>
003	Departure airport &1 is unknown	<input type="checkbox"/>
004	Destination airport &1 is unknown	<input type="checkbox"/>
005	Flight &1 &2 not created	<input type="checkbox"/>
006	Flight &1 &2 has no flight data	<input type="checkbox"/>
007	Internal error when reading screen data	<input type="checkbox"/>
008	Unable to display possible entries	<input type="checkbox"/>
009	You are not authorized to display flight data	<input type="checkbox"/>
010	You are not authorized to change flight data	<input type="checkbox"/>

4. Return to your code and correct the message number, if necessary.

If you want to enter several message texts all at once, use the *Maintain all* button to convert all messages to update-mode. Then you don't need to save each message text individually. But remember that you block other users from updating the message class when you do this.

Example Transaction: Checking Field Input

As an example of user-programmed field-checking, we have been using transaction TZ31, one of the example transactions delivered with every system. A quick look at the flow logic and ABAP code for the program demonstrates how the various checks are sewn into the program.

Screen Flow Logic

The screen flow logic (PAI only) for transaction TZ31 looks as follows.

```
*-----*
* Screen 100: Flow Logic                               *
*&-----*
PROCESS AFTER INPUT.
  MODULE EXIT AT EXIT-COMMAND.
*
CHAIN.
  FIELD: SPFLI-CARRID, SPFLI-CONNID.
    MODULE CHECK_FLIGHT ON CHAIN-REQUEST.
ENDCHAIN.
*
  FIELD SPFLI-AIRPFROM
    MODULE CHECK_FR_AIRPORT ON REQUEST.
  FIELD SPFLI-AIRPTO
    MODULE CHECK_TO_AIRPORT ON REQUEST.
*
  MODULE USER_COMMAND_0100.
```

The flow logic is organized so that:

- The statement `MODULE...AT EXIT-COMMAND` handles “exit commands” and occurs at the beginning of PAI. Exit-commands involve processing that must happen *before* the system performs its automatic field checks.
- All `FIELD` and `CHAIN` statements come before any logic that processes user commands. The statements here show that the ABAP modules for checking field values only execute if the `ON REQUEST` condition is fulfilled.

ABAP Code

The following are the TZ31 modules relevant to field-checking. The first, the `EXIT` module, handles the exit-commands, which require no screen input from the user. This includes the `NEW` and `CANC` functions: the `SET SCREEN` and `LEAVE SCREEN` statements cause either the screen or the transaction to terminate. These statements are described in [Controlling the Screen Flow \[Page 712\]](#).

```
*&-----*
*&  Module EXIT INPUT
*&-----*
MODULE EXIT INPUT.
  CASE OK_CODE.
    WHEN 'NEW'.
      CLEAR: SPFLI, OK_CODE.
      LEAVE SCREEN.
    WHEN 'CANC'.
      CLEAR OK_CODE.
```


Example Transaction: Checking Field Input

```

    SET SCREEN 0. LEAVE SCREEN.
  ENDCASE.
ENDMODULE.

```

The CHECK_FLIGHT module makes sure the flight-number exists for the given airline. If not, the program puts out a message. Note that the SELECT statement alone cannot fill the *Regular* and *Charter* screen fields, since they are represented by a single field in the database. So the module must fill these screen fields explicitly.

```

*&-----*
*&   Module CHECK_FLIGHT INPUT
*&-----*
MODULE CHECK_FLIGHT INPUT.
  SELECT SINGLE * FROM SPFLI
    WHERE CARRID = SPFLI-CARRID
    AND CONNID  = SPFLI-CONNID.

  IF SY-SUBRC NE 0.
    MESSAGE E005 WITH SPFLI-CARRID SPFLI-CONNID.
  ENDIF.
  CLEAR: CHARTER, REGULAR.
  IF SPFLI-FLTYPE = SPACE. REGULAR = 'X'.
  ELSE.                CHARTER = 'X'.
  ENDIF.
ENDMODULE

```

Field-checking for the airport fields is as follows.

```

*&-----*
*&   Module CHECK_AIRPORT INPUT
*&-----*
MODULE CHECK_FR_AIRPORT INPUT.
  SELECT SINGLE * FROM SAIRPORT
    WHERE ID = SPFLI-AIRPFROM.
  IF SY-SUBRC NE 0. MESSAGE E003 WITH SPFLI-AIRPFROM. ENDIF.
ENDMODULE.

```

After all field-checking has executed, the program can safely use the screen input to process user commands. The module USER_COMMAND_0100 (not shown here) updates the database and puts out a message (either an abort or success), depending on results. Neither of these message types causes the screen processor to enter an error dialog.

Messages on Selection Screens

On selection screens, messages are processed in the [AT SELECTION-SCREEN \[Page 1216\]](#) event. They are described in the documentation for the event.

Messages in Lists

Messages in Lists

ABAP allows you to react to incorrect or doubtful user input by displaying messages that influence the program flow depending on how serious the error was. Handling messages is mainly a topic of dialog programming where it is described in detail (see [Handling Errors and Messages \[Page 1172\]](#)). This topic only explains the influence messages have on interactive list processing.

You store and maintain messages in Table T100. Messages are sorted by language, by a two-character ID, and by a three-digit number. You can assign different message types to each message you output. The influence of a message on the program flow depends on the message type.

In your program, use the MESSAGE statement to output messages statically or dynamically and to determine the message type.

Use the MESSAGE-ID option of the REPORT or PROGRAM statement, if you want to use messages of a certain ID statically in your program:

Syntax

REPORT <rep> MESSAGE-ID <id>.

Due to this statement, the report <rep> can use all messages stored in Table T100 under the ID <id>. If you specify message IDs dynamically, you can omit this option.

To specify a message number and type statically, use:

Syntax

MESSAGE <c><num> [WITH <f₁>... <f₄>].

This statement outputs the message stored in Table T100 under number <num>, that has the same ID <id> as in the REPORT statement, as message type <c>.

To specify a message ID, type, and number dynamically at runtime, use:

Syntax

MESSAGE ID <id> TYPE <c> NUMBER <num> [WITH <f₁>... <f₄>].

This statement outputs the message whose ID, number, and type are stored in the fields <id>, <num>, and <c>. For this statement, you do not need the MESSAGE-ID option in the REPORT statement.

A message can have five different types. These message types have the following effects during list processing:

- A (=Abend):
The system displays a message of this message type in a dialog window. After the user confirms the message using ENTER, the system terminates the entire transaction (for example SE38).
- E (=Error) or W (=Warning):
The system displays a message of this message type in the status line. After the user chooses ENTER, the system acts as follows:
 - While creating the basic list, the system terminates the report.

- While creating a secondary list, the system terminates the corresponding processing block and keeps displaying the previous list level.
- I (=Information):
The system displays a message of this message type in a dialog window. After the user chooses ENTER, the system resumes processing at the current program position.
- S (=Success):
The system displays a message of this message type on the output screen in the status line of the currently created list.

Ampersand characters '&' serve as placeholders within a message. If you use the WITH option, the system replaces the placeholders '&' in the message one after the other with the contents of the fields <f_i> and the numbered placeholders '&i' according to the number. To output a '&' character in the message, you must write '&&'.

To create, display, or change messages, simply double-click on the ID <id> or the message number in the ABAP Editor. If the object does not yet exist, the system asks you whether you want to create it.

You can easily include static MESSAGE statements into your program by choosing *Edit → Insert statement...* from the ABAP Editor. Select MESSAGE as statement:

MESSAGE ID **HB** Typ **E** Number **100**

With this method, you can either copy existing messages or create a new object (ID and number). In addition, the system includes the message text as comment into your program.

Suppose, the following messages are stored in Table T100 under the ID HB:

100 Example for Message on Lists

200 Level &1 not allowed

The following program uses these messages:

REPORT SAPMZTST MESSAGE-ID HB NO STANDARD PAGE HEADING.

WRITE 'Basic List'.

MESSAGE S100.

AT LINE-SELECTION.

IF SY-LSIND = 1.

MESSAGE ID 'HB' TYPE 'I' NUMBER 100.

ENDIF.

IF SY-LSIND = 2.

MESSAGE E200 WITH SY-LSIND.

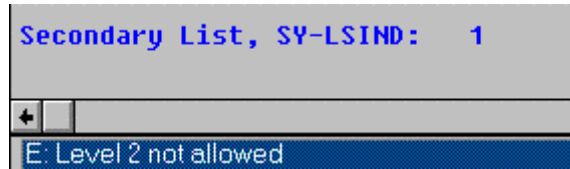
ENDIF.

WRITE: 'Secondary List, SY-LSIND:', SY-LSIND.

After executing the program, the system displays the basic list and the success message 100 in the status line. After selecting a line by double-clicking, the event AT LINE-SELECTION occurs. While the system creates the first secondary list, it

Messages in Lists

displays a dialog window with the information message 100. You cannot create the second secondary list, because the message 200 has message type E:



Running ABAP Programs

Directly Executable Programs

[Executable Programs \(Reports\) and Logical Databases \[Page 1200\]](#)

[Using Events to Control Program Flow \[Page 1208\]](#)

[Logical Databases - Attributes and Maintenance \[Page 1246\]](#)

Executable Programs (Reports) and Logical Databases

An executable program (report) has the program attribute *Type 1*.

Users can run executable reports either in the foreground, by entering the program name in Transaction SA38 (*System* → *Services* → *Reporting*), or as a background job. You do not need to assign a transaction code to an executable program (report), although you may if you wish, and you do not have to use the Screen Painter to create any screens for it.

When an executable program (report) is run, the program flow is controlled by an invisible system program (see [Starting a Program Directly \[Ext.\]](#)). The system program starts by calling a selection screen, then controls the database access to read the data, and the output of the data in a list. Executable programs (reports) are often used for reading data (reporting). This is further supported by linking executable programs with logical databases.

There are two ways of accessing data from database tables for analysis with executable programs (reports).

[Accessing Data with SELECT \[Page 1201\]](#)

[Accessing Data Using Logical Databases \[Page 1202\]](#)

The following topics contain

[Comparison of Access Methods \[Page 1204\]](#)

[Advantages of Logical Databases \[Page 1206\]](#)

[Controlling the Database Accesses from the Executable program \(Report\) \[Page 1207\]](#)

For information about the features and maintenance of logical databases, i.e. how to display, change or create logical databases (Transaction SLDB or SE36), see [Features and Maintenance of Logical Databases \[Page 1246\]](#).

Accessing Data with SELECT

The SELECT statement is part of SAP's Open SQL which is a subset of standard SQL. For further information about the SELECT statement, see [Reading Data from Database Tables \[Page 542\]](#).

You can read and analyze data from all database tables known to the SAP system by using the SELECT statement and its different clauses. An executable program which uses only SELECT statements is called an SQL report.

In an SQL report, there is no absolute need for external flow control by events. Provided no event keywords are used (see [Controlling the Flow of ABAP Programs by Events \[Page 1208\]](#)), you can treat an SQL report like a conventional sequentially processed program, since only a single processing block (START-OF-SELECTION) is called.

However, if you want to access data using logical databases, the use of events is unavoidable.

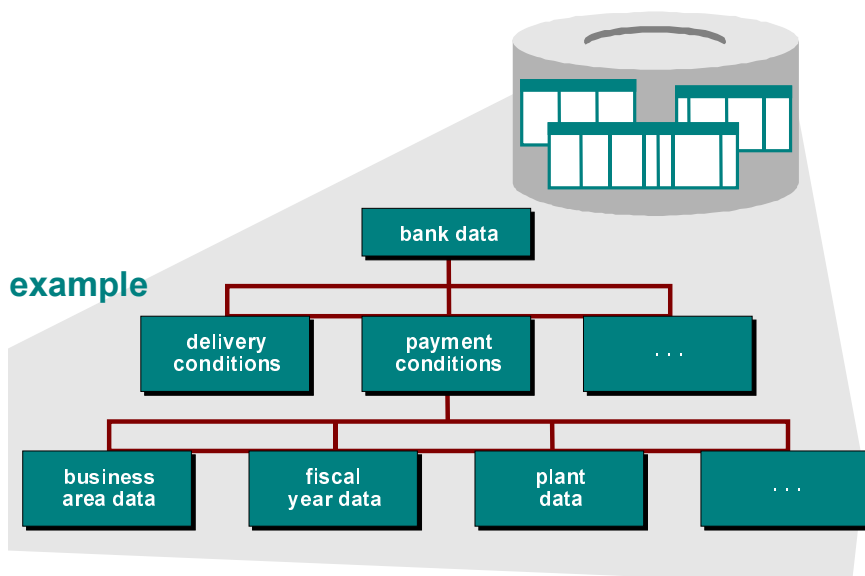
Accessing Data Using Logical Databases

Logical databases provide a method for accessing data in database tables which differs considerably from the SELECT statement.

A logical database is a special ABAP program which combines the contents of certain database tables. You can link a logical database to an ABAP report program as an attribute. The logical database then supplies the report program with a set of hierarchically structured table lines which can be taken from different database tables.

The term “logical database” refers not only to the program itself, but also to the dataset it supplies.

In the SAP system, many tables are linked by foreign key dependencies (for further information about this, see the documentation [BC - ABAP Dictionary \[Ext.\]](#)). Some of these dependencies form tree-like hierarchical structures. Using logical databases facilitates the process of reading database tables which form the components of these structures.



The above diagram shows how the SAP system represents enterprise structures. A logical database can read the lines of these tables one after the other into an executable program in a sequence which is normally defined by the hierarchical structure.

The ABAP Development Workbench includes a convenient tool for creating and displaying logical databases (either call Transaction SLDB or choose *Tools → ABAP Development Workbench → Development → Programming environ. → Logical databases*). To see the hierarchical structure of a logical database <ldb>, you can also type SHOW DATABASE <ldb> in the command field of the ABAP Editor. For detailed information about maintaining logical databases, see [Features and Maintenance of Logical Databases \[Page 1246\]](#).

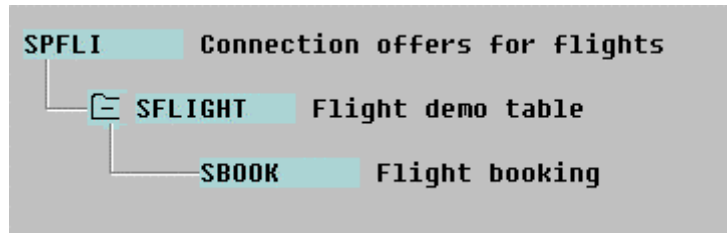
When you link a logical database to a report program in order to access data, the sequential program flow is no longer sufficient. Instead, you must program a sequence based on events (see diagram in [Programming Logical Expressions \[Page 235\]](#)). The logical database then provides the events for the external flow control of your report program. The most important event in connection with logical databases is GET (see [Events and their Event Keywords \[Page 1211\]](#)).

Accessing Data Using Logical Databases

You can also use SELECT statements in an executable program (report) linked to a logical database.

Comparison of Access Methods

The example below shows a comparison between a report program which uses SELECT statements and a report which uses a logical databases. All the examples in this section use the logical database F1S. You must specify this name in the program attributes (see [Maintain Program Attributes \[Page 74\]](#)). The structure of the logical database F1S is as follows (editor command SHOW DATABASE F1S):



The following table compares two report programs which both read data from the hierarchically structured database tables SPFLI, SFLIGHT, and SBOOK.

Report with SELECT statements	Report with logical databases
REPORT.....	REPORT.....
TABLES: SPFLI, SFLIGHT, SBOOK.	TABLES SPFLI, SFLIGHT,SBOOK.
SELECT * FROM SPFLI WHERE..	GET SPFLI.
<processing block>	<processing block>
SELECT * FROM SFLIGHT WHERE..	GET SFLIGHT.
<processing block>	<processing block>
SELECT * FROM SBOOK WHERE..	GET SBOOK.
<processing block>	<processing block>
ENDSELECT.	
ENDSELECT.	
ENDSELECT.	

The executable program (report) in the left column reads the data with nested SELECT loops. The program in the right column uses the logical database F1S. Note that in both cases, the database tables must be declared with the TABLES statement in the program.

The executable program (report) in the right column uses a logical database program which reads the data from the database tables with SELECT statements, like the executable program (report) in the left column does. The main structure of the logical database program is as follows:

Logical database program
REPORT SAPDB...
TABLES: SPFLI, SFLIGHT, SBOOK.
SELECT * FROM SPFLI WHERE..
SELECT * FROM SFLIGHT WHERE..
SELECT * FROM SBOOK WHERE..

Comparison of Access Methods

.....
ENDSELECT.
ENDSELECT.
ENDSELECT.

In general, a logical database program reads the data from the database tables with SELECT statements. This means that the code containing the SELECT statements is rolled out from the executable program (report) to the logical database program.

For information about the precise structure of the logical database program, see [Database Program of a Logical Database \[Page 1254\]](#).

Advantages of Logical Databases

Using logical databases saves you from having to program the data retrieval from the database tables. In your executable program (report), you do not have to define how the information should be retrieved, but only how it should be presented on the screen. In the processing blocks after the GET events, you only have to specify the statements for analyzing the data and writing the results to the screen.

After each table line is transferred, a GET event occurs and the executable program (report) is able to process it by activating the appropriate processing block. If you do not specify a logical database in the program attributes, the GET events never occur.

When you are working with logical databases, you do not have to program a selection screen for user input since this is created automatically (see [Standard Selection Screens and Logical Databases \[Page 797\]](#)). If you are only working with SELECT statements, however, you have to program the selection screen yourself (see [Working with Selection Screens \[Page 795\]](#)).

A report can only work with one logical database, but each logical database can be used by several reports. This offers considerable advantages over integrating the database accesses with SELECT statements into each executable program (report). It means that you only have to code identical access paths once. The same applies to coding authorization checks.

When using logical databases, most reports benefit from the following features:

- an easy-to-use standard user interface
- check functions which check that user input is complete, correct, and plausible
- meaningful data selection
- central authorization checks for database accesses
- good read access performance (for example, with views) while retaining the hierarchical data view determined by the application logic.

Even when using central logical databases, you have flexibility because you can

- create and design your own selection screen versions
- use particular event keywords in ABAP executable program (report)s to enable extensive user dialogs (e.g. for further authorization or plausibility checks).

Controlling the Database Accesses from the Executable program (Report)

Controlling the Database Accesses from the Executable program (Report)

The executable program (report) can influence the amount of data read by the logical database program. The data currently being processed can be dynamically checked at any time to ensure that it satisfies any selection criteria. If this is not the case, the logical database skips the subordinate parts of the hierarchy structure. If you do not select a particular table for processing, the system automatically ignores the relevant part of the tree.

If the system reads an entry from a logical database table, a GET event occurs. In the executable program (report), you can assign a processing block to the GET event. If, for example, the system reads an entry from table SPFLI, it executes the processing block assigned to GET SPFLI. You do not have to define a processing block for each table a report uses in order to address fields. The system reads the tables whether you define a processing block or not. If, however, database tables are designated for field selection in the logical database, the system behaves differently (see [GET <table> \[Page 1226\]](#)).

A processing block contains both the fields of the table just read and also the fields of all superior tables on the current access path. For example, in the event GET SBOOK., you can address the fields of tables SPFLI, SFLIGHT, and SBOOK. This is because the logical database program reads tables with nested SELECT loops (see [Comparison of Access Methods \[Page 1204\]](#)).

However, since you do not always need to access fields from all of the tables, it would waste valuable CPU time to read all of the tables belonging to a logical database each time. Therefore, the read depth depends on the GET statement corresponding to the tables located at the lowest level of the hierarchy.

Suppose you want to generate a list of flight connections. In this case, the system would only need to read fields from table SPFLI.

TABLES: SPFLI.

...

GET SPFLI.

The system does not read data from the table SBOOK.

To obtain the documents relevant to a booking, you need the tables SPFLI and SBOOK.

TABLES: SPFLI, SBOOK.

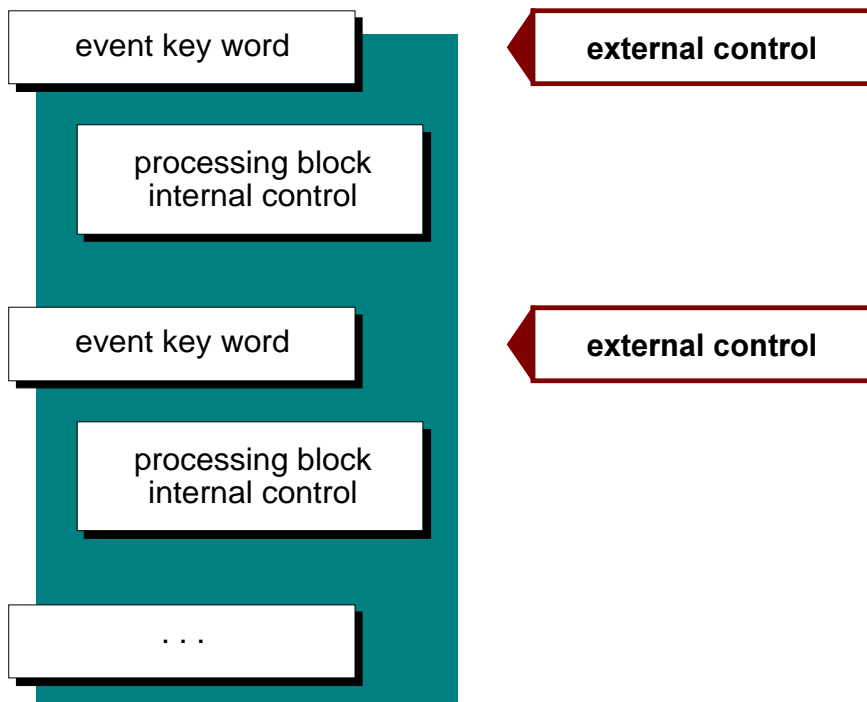
...

GET SBOOK.

The system also reads the table SFLIGHT because it is located on the access path of the table SBOOK. However, you can only access the data in SFLIGHT if you declare the appropriate table work area to the report with TABLES.

Controlling the Flow of ABAP Programs by Events

The program flow of an executable program (report) is controlled by an invisible system program dependent on external events. You can assign processing blocks to events in the executable program using event keywords.



A significant feature of executable ABAP programs (type 1 programs) is that they are made up of processing blocks for **events**. There are various event classes:

- When reading data from the database, the system program creates a series of events. The sequence of these events follows a fixed scheme and is partly defined by the used logical database.
- During the creation of output lists, events occur that can be used for list formatting.
- During the display of lists on the screen, the user can produce interactive list events for interactive reporting.

The interface between events and the program is provided by event keywords. The system always starts a processing block when the corresponding event occurs (see [Program Construction and Execution \[Ext.\]](#)).

The flow control inside a processing block (internal control) is described in [Controlling the Flow of an ABAP Program \[Page 234\]](#). This section describes the external control of an ABAP program and the possible events involved:

[Defining Processing Blocks \[Page 1209\]](#)

[Events and their Event Keywords \[Page 1211\]](#)

[Terminating Processing Blocks \[Page 1233\]](#)

Defining Processing Blocks

Defining Processing Blocks

You can define processing blocks in your ABAP program by using event keywords. The possible event keywords are described in [Events and their Event Keywords \[Page 1211\]](#).

All statements between two event keywords or between an event keyword and a FORM statement (see [Defining Subroutines \[Page 445\]](#)) form a processing block. When an event occurs, the system processes the processing block after the corresponding event keyword.

Each statement in an ABAP report program is part of a processing block or a subroutine.

Statements which do not follow an event keyword or a FORM-ENDFORM block are automatically part of the processing block of the default event START-OF-SELECTION (for more information about this event, see [START-OF-SELECTION \[Page 1225\]](#)). This has the following consequences:

- If you write statements between the REPORT or PROGRAM statements and the first event keyword or FORM statement, these statements are included in the START-OF-SELECTION processing block.
 - If no START-OF-SELECTION keyword is included in your program, these statements form the entire START-OF-SELECTION processing block.
 - If a START-OF-SELECTION keyword is included in your program, these statements are inserted at the beginning of this block.
- If you do not specify any event keywords in your program, all statements of the program before a FORM statement form the START-OF-SELECTION processing block.

All statements between an ENDFORM statement and an event keyword or between an ENDFORM statement and the end of the program form a processing block that is never processed. Do not place any statements there. Place all subroutines at the end of your program.

The order in which the processing blocks appear in the program is irrelevant because the sequence of processing is triggered exclusively by events. For reasons of readability, however, you should list processing blocks within a program in their order of execution.

Unlike operational or control statements, the system processes declarative statements during the generation of a program and not at runtime (for more information about declarative statements, see [Keywords \[Page 88\]](#)). They are processed regardless of their position in the program code and can belong to any processing block. For the sake of clarity, you should place all declarative keywords at the beginning of your program or subroutine.

```
REPORT SAPMZTST.  
WRITE / 'Statement 1'.  
FORM ROUTINE.  
  WRITE / 'Subroutine'.  
ENDFORM.
```

```
WRITE / 'Statement 2'.
PERFORM ROUTINE.
WRITE / 'Statement 3'.
```

The output looks as follows:

Statement 1

In this program, only the START-OF-SELECTION processing block is executed. This block consists of the first WRITE statement.

Now, insert a START-OF-SELECTION statement into the program as follows:

```
REPORT SAPMZTST.
WRITE / 'Statement 1'.
FORM ROUTINE.
  WRITE / 'Subroutine'.
ENDFORM.
START-OF-SELECTION.
  WRITE / 'Statement 2'.
  PERFORM ROUTINE.
  WRITE / 'Statement 3'.
```

The output is now as follows:

Statement 1

Statement 2

Subroutine

Statement 3

In this program, the START-OF-SELECTION processing block consists of all statements except the FORM-ENDFORM block. The following is a more readable form of this program:

```
REPORT SAPMZTST.
START-OF-SELECTION.
  WRITE / 'Statement 1'.
  WRITE / 'Statement 2'.
  PERFORM ROUTINE.
  WRITE / 'Statement 3'.
FORM ROUTINE.
  WRITE / 'Subroutine'.
ENDFORM.
```

In this form of the program, you could also omit the START-OF-SELECTION statement.

Events and their Event Keywords

Events and their Event Keywords

There are various groups of event keywords which control the flow of an ABAP program under certain circumstances.

Logical databases are the central point for the external flow control of a typical report program (see [Accessing Data Using Logical Databases \[Page 1202\]](#)). If a logical database is linked to a report program, it causes a selection screen to be displayed and determines how the system reads data from database tables. This leads to the sequence of events described in the following table. For further information about the order in which events occur, see [Logical Databases and Executable ABAP Programs \(Reports\) \[Page 1258\]](#).

The following events occur at runtime of a typical report program which uses logical databases:

Event keyword	Event
INITIALIZATION [Page 1213]	Point before the standard selection screen is displayed
AT SELECTION-SCREEN [Page 1216]	Point after processing user input on a selection screen while the selection screen is still active
START-OF-SELECTION [Page 1225]	Point after processing the standard selection screen
GET <table> [Page 1226]	Point at which the logical database offers a line of the database table <table>.
GET <table> LATE [Page 1230]	Point after processing all tables which are hierarchically subordinate to the database table <table> in the structure of the logical database.
END-OF-SELECTION [Page 1232]	Point after processing all lines offered by the logical database.

The following topics describe the processing blocks of these events.

Further events not connected to logical databases occur during the processing and display of the output list of an executable program (report). You can use these to format the output lists and to make executable programs (reports) interactive. These events are described in the appropriate sections.

The following events occur during the processing of the output list of an executable program (report):

Event keyword	Event
TOP-OF-PAGE	Point during list processing when a new page is started
END-OF-PAGE	Point during list processing when a page is ended

You can use these keywords to improve the layout of the output list. For a description, see [Creating Complex Lists \[Page 938\]](#).

The following events occur during the display of the output list of an executable program (report):

Events and their Event Keywords

Event keyword	Event
AT LINE-SELECTION	Point at which the user selects a line
AT USER-COMMAND	Point at which the user presses a function key or enters a command in the command field
AT PF<nn>	Point at which the user presses the function key with the function code PF<n>

You can use these keywords to write a program for interactive reporting. For further information, see [Interactive Lists \[Page 1030\]](#).

For a detailed description of all event keywords, see the keyword documentation.

INITIALIZATION

INITIALIZATION

When you start a program in which a standard selection screen is defined (either in the program itself or in the linked logical database program), the system normally processes this standard selection screen first. If you want to execute a processing block before the standard selection screen is processed, you can assign it to the event keyword **INITIALIZATION**.

In this block, you specify statements which initialize the standard selection screen, for example by changing the default values of parameters or selection criteria. This is the only possibility to change the default values of parameters or selection criteria defined in logical databases. For selection criteria, you should define at least the components **<seltab>-SIGN**, **<seltab>-OPTION**, **<seltab>-LOW** of the selection table **<seltab>** (see [Selection Tables \[Page 816\]](#)) by changing its header line and appending it to the table. Otherwise, parts of the selection criterion may be undefined.

You can find out the names of the internal fields you want to change either by examining the logical database **SAPDB<ldb>** itself (with the transaction **SLDB** or by choosing *Tools → ABAP Development Workbench → Development → Programming environ. → Logical databases*), or by retrieving the technical information of that field. To do this, select the input field on the selection screen and press **F1**. Then choose *Technical info* in the next dialog box. In the field **Scrn Field** of the following window, you then see the name of the field used in the program.


Suppose you have an executable program (report) which is linked to the logical database **F1S**:

REPORT SAPMZTST.

PARAMETERS FIRSTDAY LIKE SY-DATUM DEFAULT SY-DATUM.

TABLES SPFLI.

When you start this program, the following standard selection screen appears automatically:

Carrier ID	<input type="text"/>	To	<input type="text"/>	
From	<input type="text"/>	To	<input type="text"/>	
FIRSTDAY	<input type="text" value="01/08/1996"/>			

The selection criterion with the selection text *Carrier ID* and the parameters with the selection texts *From* and *To* (see [Selection Texts \[Page 154\]](#)) are defined in the logical database **F1S**. The parameter **FIRSTDAY** is defined in the program itself.

Now you select, for example, the first input field of *Carrier ID*, press **F1**, and then choose *Technical info* to find out the name of the selection table:

The screenshot shows the 'Technical Information' dialog box for the field 'CARRID'. The dialog is divided into several sections:

- Screen data:** Report (SAPMZTST), Program name (SAPMZTST), Screen number (1000).
- GUI data:** Program name (RSSYSTDB), Status (%_00).
- Field data:** Table name (SPFLI), Field name (CARRID), Data element (S_CARR_ID), DE supplement (0), Parameter ID (CAR).
- Field description for batch input:** Scrn field (CARRID-LOW).

At the bottom of the dialog, there are buttons for 'Navigate' (with a green checkmark icon) and a red 'X' icon. Below the dialog, there are buttons for 'Extended help' (with a green checkmark icon) and 'Technical info' (with a red 'X' icon).

In the field *Scrn field*, you see the name CARRID-LOW which is the component of the selection table corresponding to the selected input. From this, you see that the name of the selection criterion is CARRID. In the same procedure as described above, you find that the parameters of the input fields *From* and *To* are named CITY_FR and CITY_TO.

Now you can change the executable program (report) follows:

REPORT SAPMZTST.

PARAMETERS FIRSTDAY LIKE SY-DATUM DEFAULT SY-DATUM.

TABLES SPFLI.

INITIALIZATION.

CITY_FR = 'NEW YORK'.

CITY_TO = 'FRANKFURT'.

CARRID-SIGN = 'I'.

CARRID-OPTION = 'EQ'.


CARRID-LOW = 'AA'.

APPEND CARRID.

FIRSTDAY+6(2) = '01'.

After starting SAPMZTST, the standard selection screen appears as follows:

INITIALIZATION

Carrier ID	AA	To		
From	NEW YORK			
To	FRANKFURT			
FIRSTDAY	01/01/1996			

The default values of the selection criterion and all parameters are changed.

AT SELECTION-SCREEN

The event keyword AT SELECTION-SCREEN provides you with several possibilities to carry out processing blocks while the system is processing a selection screen. To react on different events, that can occur when the selection screen is processed, the keyword AT SELECTION-SCREEN has various options.

You can find out the screen number of the current screen from the SY-DYNNR system field.

If you specify the keyword without any option, the corresponding processing block is started after the system has finished processing the selection screen (PAI of the selection screen). If an error message is sent from this processing block, the system displays the selection screen again and all input fields can be changed. It is your task to supply an appropriate error message.

This method enables you, for example, to make input fields mandatory, although they were not defined with the OBLIGATORY option of the PARAMETERS or SELECT-OPTIONS statement in the logical database program.

You store and maintain messages in table T100. They are grouped by language, a two-character ID, and a three-digit number. From your program, you can send a message with different qualifications:

A	Abend; the current transaction is stopped
E	Error; the system waits for new input data
I	Information; after pressing ENTER, the system continues processing
S	Confirmation; the message appears on the next screen
W	Warning; you can change the input data or continue by pressing ENTER

You must specify the MESSAGE-ID behind the REPORT or PROGRAM statement of your program. You can include messages easily into your program via the ABAP Editor by choosing *Edit* → *Insert statement...* You can also change messages from there. For more information about message handling in executable programs (reports), see [Messages in Lists \[Page 1195\]](#).

The logical database F1S is attached to the following executable program.

```
REPORT SAPMZTST MESSAGE-ID HB.
```

```
TABLES SPFLI.
```

```
AT SELECTION-SCREEN.  
  IF CARRID-LOW IS INITIAL  
    OR CITY_FR IS INITIAL  
    OR CITY_TO IS INITIAL.  
    MESSAGE E000.  
  ENDIF.
```

This executable program (report) uses a message with the ID HB. After starting SAPMZTST, the selection screen displays as defined in the logical database F1S.

AT SELECTION-SCREEN

As long as the user does not enter a value into each input field, the following error message appears in the status bar of the screen.

E: at least one field is initial, please enter values for all fields

The message 000 was coded for this example with ID HB in table T100.

The options of the event keyword AT SELECTION-SCREEN enable you to create processing blocks for special events during the processing of the selection screen. These events are described in the following topics:

[Processing a Particular Input Field \[Page 1218\]](#)

[Processing Multiple Selection \[Page 1219\]](#)

[Creating a List of Input Values \[Page 1220\]](#)

[Creating Help for Input Fields \[Page 1221\]](#)

[Processing a Group of Radio Buttons \[Page 1222\]](#)

[Processing a Block of Input Fields \[Page 1223\]](#)

[PBO of the Selection Screen \[Page 1224\]](#)

These options are shortly introduced in the following topics. For more information, see the keyword documentation on AT SELECTION-SCREEN.

Processing a Particular Input Field

To start a processing block after a particular input field of the selection screen is processed, use the keyword `AT SELECTION-SCREEN` as follows:

Syntax:

`AT SELECTION-SCREEN ON <field>.`

The corresponding processing block is started when the system has processed the input field for the variable `<field>`. If an error message is sent from this processing block, the system displays the selection screen again, and only the input field for the variable `<field>` must be changed by the user.

The logical database F1S is attached to the following executable program (report):

```
REPORT SAPMZTST MESSAGE-ID HB.
```

```
TABLES SPFLI.
```

```
AT SELECTION-SCREEN ON CITY_FR.
```

```
  IF CITY_FR NE 'NEW YORK'.
```

```
    MESSAGE E010.
```

```
  ENDIF.
```

If the user does not insert "NEW YORK" in the field *From* on the selection screen, the following error message appears in the status bar of the screen



E: Please, insert NEWYORK for field FROM

until the user makes the correct input. The message 010 was coded for this example with ID HB in table T100.

Processing Multiple Selection

Processing Multiple Selection

After entering complex selections for a particular selection criterion into the *Multiple Selection* window of the selection screen and processing this window (see example in [Basic Form of the SELECT-OPTIONS Statement \[Page 821\]](#)), you can call a processing block. To do this, use the AT SELECTION-SCREEN statement as follows:

Syntax

AT SELECTION-SCREEN ON END OF <seltab>.

The corresponding processing block is started at the end of processing the *Multiple selection* window of selection criterion <seltab>. You can use this options to check the entries in the internal table <seltab>.

The logical database F1S is attached to the following executable program (report):

```
REPORT SAPMZTST MESSAGE-ID HB.
```

```
TABLES SPFLI.
```

```
AT SELECTION-SCREEN ON END OF CARRID.
```

```
  LOOP AT CARRID.
```

```
    IF CARRID-HIGH NE ' '.
```


```
      IF CARRID-LOW IS INITIAL.
```

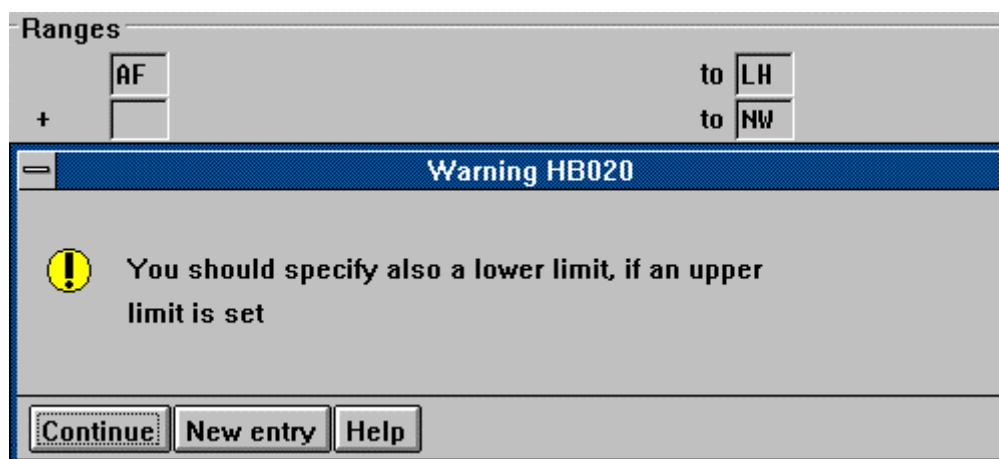
```
        MESSAGE W020.
```

```
      ENDIF.
```

```
    ENDIF.
```

```
  ENDLOOP.
```

If, after starting SAPMZTST, the user clicks the arrow icon  on the standard selection screen and then enters upper limits without corresponding lower limits for range selections in the *Multiple Selection* window, the following dialog box appears with a warning:



The message 020 was coded for this example with ID HB in table T100.

Creating a List of Input Values

You can create a list of possible input values for an input field on the selection screen by using the AT SELECTION-SCREEN statement as follows:

Syntax

AT SELECTION-SCREEN ON VALUE-REQUEST FOR <field>.

If you use this statement, the possible entries button automatically appears next to the input field for parameter or selection criterion <field> when it is selected on the selection screen.

You cannot use this statement in a logical database program.. You can use the VALUE-REQUEST option of the PARAMETERS and SELECT-OPTIONS statements with logical databases (see the keyword documentation).

You must program the list of proposed values for <field> inside the processing block of the AT SELECTION-SCREEN ON VALUE REQUEST statement. When the user clicks the possible entries button or presses F4, this list will display. How to code such a list is a subject of dialog programming and is described in [Programming Field- and Value-Help \[Page 787\]](#).

PARAMETERS FIELD(10).

AT SELECTION-SCREEN ON VALUE-REQUEST FOR FIELD.

The parameter would appear as follows:



If a list of proposed values is programmed for FIELD, then it will display if the user clicks the possible entries button.

Creating Help for Input Fields

Creating Help for Input Fields

You can create your own help for an input field on the selection screen by using the AT SELECTION-SCREEN statement as follows:

Syntax

AT SELECTION-SCREEN ON HELP-REQUEST FOR <field>.

If you use this statement, the help text will display when the user selects the input field of <field> on the selection screen and presses the F1 key.

You cannot use this statement in a logical database program. You use the HELP-REQUEST option of the PARAMETERS and SELECT-OPTIONS statements with logical databases (refer to the keyword documentation).

You must program your help text inside the processing block of the AT SELECTION-SCREEN ON HELP REQUEST statement. How to code such a help is a subject of dialog programming and is described in [Programming Field- and Value-Help \[Page 787\]](#).

Processing a Group of Radio Buttons

To start a processing block after a group of radio buttons is processed on the selection screen (see [Creating Radio Button Groups on the Selection Screen \[Page 811\]](#)), use the keyword AT SELECTION-SCREEN as follows:

Syntax:

AT SELECTION-SCREEN ON RADIOBUTTON GROUP <radi>.

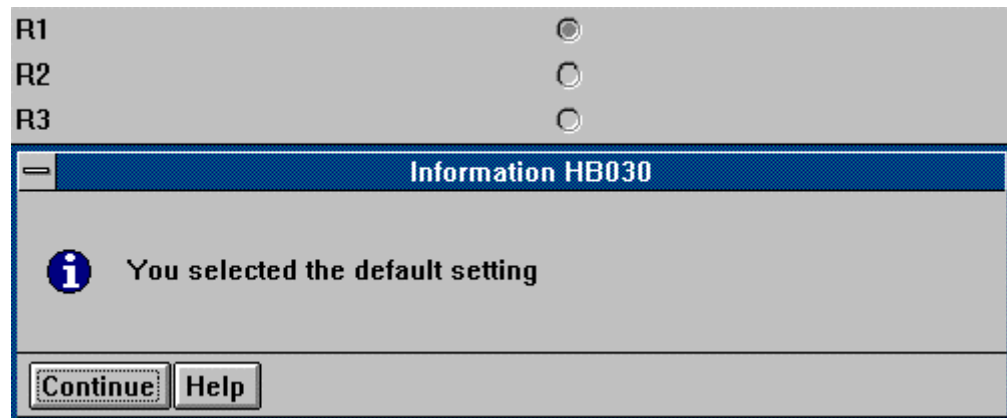
The corresponding processing block is started when the system has processed the radio button group <radi>. If an error message is sent from this processing block, the system displays the selection screen again, and only the input fields of the radio button group <radi> must be changed by the user.

```
REPORT SAPMZTST MESSAGE-ID HB.
```

```
PARAMETERS: R1 RADIOBUTTON GROUP RAD1 DEFAULT 'X',  
             R2 RADIOBUTTON GROUP RAD1,  
             R3 RADIOBUTTON GROUP RAD1.
```

```
AT SELECTION-SCREEN ON RADIOBUTTON GROUP RAD1.  
  IF R1 = 'X'.  
    MESSAGE I030.  
  ENDIF.
```

If the user does not change the radio buttons on the selection screen, the following information message appears:



For this example, the message 030 was coded with ID HB in table T100.

Processing a Block of Input Fields

Processing a Block of Input Fields

To start a processing block after a block of elements is processed on the selection screen (see [Creating Blocks of Elements \[Page 841\]](#)), use the keyword AT SELECTION-SCREEN as follows:

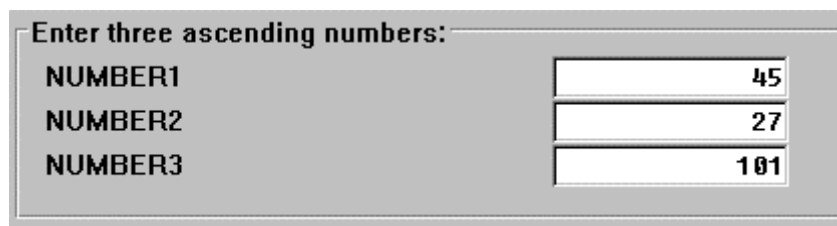
Syntax:

AT SELECTION-SCREEN ON BLOCK <block>.

The corresponding processing block is started when the system has processed the block of elements <block>. If an error message is sent from this processing block, the system displays the selection screen again, and only the input fields of the block <block> must be changed by the user.

```
REPORT SAPMZTST MESSAGE-ID HB.  
SELECTION-SCREEN BEGIN OF BLOCK PART1  
    WITH FRAME TITLE TEXT-001.  
    PARAMETERS: NUMBER1 TYPE I,  
               NUMBER2 TYPE I,  
               NUMBER3 TYPE I.  
SELECTION-SCREEN END OF BLOCK PART1.  
AT SELECTION-SCREEN ON BLOCK PART1.  
    IF NUMBER3 LT NUMBER2 OR  
       NUMBER3 LT NUMBER1 OR  
       NUMBER2 LT NUMBER1.  
       MESSAGE E040.  
    ENDIF.
```

After starting SAPMZTST, the user can enter numbers into the input fields of block PART1. If he does not enter the numbers in ascending order,



Enter three ascending numbers:	
NUMBER1	45
NUMBER2	27
NUMBER3	101

the following error message appears in the status bar:

E: Please, enter ascending numbers | E: Please, enter ascending numbers

For this example, the message 040 was coded with ID HB in table T100.

PBO of the Selection Screen

To start a processing block during the PBO of a selection screen for each ENTER, use the following AT SELECTION-SCREEN statement :

Syntax

AT SELECTION-SCREEN OUTPUT.

During these events, you can apply, for example, modifications to the fields on the selection screen. For further information and an example, see [Assigning Parameters to a Modification Group \[Page 814\]](#).

START-OF-SELECTION

START-OF-SELECTION

The event START-OF-SELECTION gives you the possibility of creating a processing block after processing the standard selection screen and before accessing database tables using a logical database. You can use this processing block, for example, to set the values of internal fields or to write informational statements onto the output screen.

At the START-OF-SELECTION event, also all statements are processed that are not attached to an event keyword except those that are written behind a FORM-ENDFORM block (for an example, see [Defining Processing Blocks \[Page 1209\]](#)).

GET <table>

The most important event for executable programs (reports) with an attached logical database is the moment at which the logical database program has read a line from a database table (see [Accessing Data Using Logical Databases \[Page 1202\]](#)). To start a processing block at this event, use the GET statement as follows:

Syntax

GET <table> [FIELDS <list>].

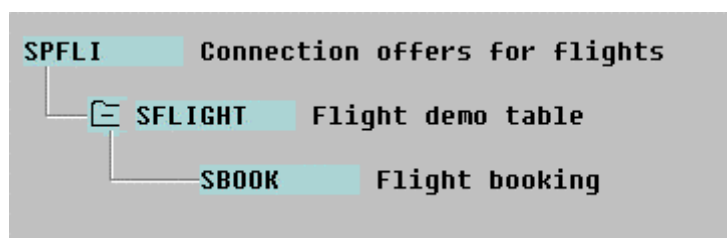
After this statement, you can work with the current line of the database table <table>. The data is provided in the table work area <table>.

The logical database reads **all** columns from **all** database tables which are not designated for field selection in the logical database and which are superior to <table> on the access path of the logical database (see [Controlling the Database Accesses from the Executable program \(Report\) \[Page 1207\]](#)). This is independent from whether you have specified a GET statement for these tables or not. However, you can access only the data of those tables that you have declared by using the TABLES statement in the program.

Performance can be much better for tables that are designated for field selection in the logical database (see [Editing Selections \[Page 1278\]](#)). If such tables are superior to <table> on the access path of the logical database and you do not specify a GET statement for them, the system reads the data of all columns only from those tables that are declared by using the TABLES statement in the program. From tables that are not declared by using the TABLES statement in the program, the system reads only the key fields (the logical database needs the key fields to build up the access path).

With the FIELDS option, you can explicitly specify the columns of the database table that should be read by the logical database. The FIELDS option is described in [Specifying Fields of the Database Table Explicitly \[Page 1229\]](#).

The logical database F1S is attached to the following executable program (report). F1S has the structure:



The code of the executable program (report) SAPMZTST is as follows:

```

REPORT SAPMZTST.

TABLES: SPFLI, SFLIGHT, SBOOK.

START-OF-SELECTION.
  WRITE 'Test Program for GET'.

GET SPFLI.
SKIP.
WRITE: / 'From:', SPFLI-CITYFROM,
       'To:', SPFLI-CITYTO.
  
```

GET <table>

```
GET SFLIGHT.  
SKIP.  
WRITE: / 'Carrid:', SFLIGHT-CARRID,  
        'Connid:', SFLIGHT-CONNID.  
ULINE.  
  
GET SBOOK.  
WRITE: / 'Fldate:', SFLIGHT-FLDATE,  
        'Bookid:', SBOOK-BOOKID,  
        'Luggweight', SBOOK-LUGGWEIGHT.  
ULINE.
```

Suppose, after starting SAPMZTST, the user fills the standard selection screen as follows:

Carrier ID	AA	To	UA	
From	Frankfurt			
To	Berlin			

Then, the first part of the output list appears as follows:

GET <table>

Test Program for GET			
From: FRANKFURT		To: BERLIN	
Carrid: LH Connid: 2402			
Fldate: 01/30/1995	Bookid: 00010001	Luggweight	40
Fldate: 01/30/1995	Bookid: 00010002	Luggweight	0
Fldate: 01/30/1995	Bookid: 00010003	Luggweight	0
Fldate: 01/30/1995	Bookid: 00010004	Luggweight	0
Fldate: 01/30/1995	Bookid: 00010005	Luggweight	50
Fldate: 01/30/1995	Bookid: 00010006	Luggweight	0
Fldate: 01/30/1995	Bookid: 00010007	Luggweight	0
Fldate: 01/30/1995	Bookid: 00010008	Luggweight	40
Fldate: 01/30/1995	Bookid: 00010009	Luggweight	0
Fldate: 01/30/1995	Bookid: 00010010	Luggweight	0
Carrid: LH Connid: 2402			
Fldate: 02/01/1995	Bookid: 00010001	Luggweight	0
Fldate: 02/01/1995	Bookid: 00010002	Luggweight	50
Fldate: 02/01/1995	Bookid: 00010003	Luggweight	0

Note that the table work area SPFLI is used in the processing block behind GET SBOOK.

Specifying Fields of the Database Table Explicitly

Specifying Fields of the Database Table Explicitly

To specify with which fields of the database table you want to work during a GET event, use the FIELDS option of the GET statement as follows:

Syntax

GET <table> [LATE] FIELDS <f₁> <f₂>...

With the FIELDS option, the logical database program reads only the fields <f₁> <f₂>... and the key fields from the database table <table>. The usage of the FIELDS option can result in **relevant performance improvements**.

The logical database linked to the executable program (report) must designate the database <table> for field selections (see [Editing Selections \[Page 1278\]](#)).

All fields of the database table <table> that are not key fields and are not listed after FIELDS, are not read by the logical database. The contents of the corresponding components of the table work area <table> are undefined during the GET event with the FIELDS option. They are also undefined during GET events of database tables that are inferior to <table> in the hierarchy of the logical database. Do not address these undefined fields and do not call external subroutines, that work with these fields.

Assume, that the logical database F1S designates the database tables SFLIGHT and SBOOK for field selection. Then, the following program can be written:

TABLES: SFLIGHT, SBOOK.

GET SFLIGHT FIELDS CARRID CONNID.

GET SBOOK FIELDS BOOKID.

GET SFLIGHT LATE FIELDS PLANETYPE.

In this example, the system reads the fields

- MANDT, CARRID, CONNID, FLDATE, and PLANETYPE from SFLIGHT
- MANDT, CARRID, CONNID, FLDATE, and BOOKID from SBOOK

Note that the system reads the fields MANDT and FLDATE from SFLIGHT, because MANDT and FLDATE are key fields of this table.

In this example only the key fields are read from SBOOK.

GET <table> LATE

To start a processing block at the moment after the system has processed all database tables of a logical database that are hierarchically inferior to a specific database table, use the event keyword GET as follows:

Syntax

GET <table> LATE [FIELDS <list>].

In analogy to executable programs (reports) that use only SELECT statements (see table in [Comparison of Access Methods \[Page 1204\]](#)), the processing block of a GET <table> LATE statement would appear directly before the ENDSELECT statement in the SELECT loop for the database table <table>.

The FIELDS option functions as with the GET <table> event and is explained in [Specifying Fields of the Database Table Explicitly \[Page 1229\]](#).

The logical database F1S is connected to the following executable program (report).

```
REPORT SAPMZTST.
```

```
TABLES: SPFLI, SFLIGHT, SBOOK.
```

```
DATA WEIGHT TYPE I VALUE 0.
```

```
START-OF-SELECTION.
```

```
  WRITE 'Test Program for GET <table> LATE'.
```

```
GET SPFLI.
```

```
  SKIP.
```

```
  WRITE: / 'From:', SPFLI-CITYFROM,
```

```
         'To:', SPFLI-CITYTO,
```

```
         'Connid:', SPFLI-CONNID.
```

```
  ULINE.
```

```
GET SFLIGHT.
```

```
  SKIP.
```

```
  WRITE: / 'Date:', SFLIGHT-FLDATE.
```

```
GET SBOOK.
```

```
  WEIGHT = WEIGHT + SBOOK-LUGGWEIGHT.
```

```
GET SFLIGHT LATE.
```

```
  WRITE: / 'Total luggage weight =', WEIGHT.
```

```
  ULINE.
```

```
  WEIGHT = 0.
```

With the same selection as in the example of [GET <table> \[Page 1226\]](#), the first part of the output screen appears as follows:

GET <table> LATE

Test Program for GET <table> LATE		
From: FRANKFURT	To: BERLIN	Connid: 2402
Date: 01/30/1995		
Total luggage weight =	130	
Date: 02/01/1995		
Total luggage weight =	270	
Date: 06/01/1995		
Total luggage weight =	500	
Date: 06/04/1995		
Total luggage weight =	490	
From: FRANKFURT	To: BERLIN	Connid: 2436
Date: 01/20/1995		
Total luggage weight =	140	
Date: 01/22/1995		
Total luggage weight =	270	

The total luggage weight is calculated for each flight at the event GET SBOOK. It is written onto the screen and reset at the event GET SFLIGHT LATE.

END-OF-SELECTION

To define a processing block after the system has read and processed all database tables of a logical database, use the keyword END-OF-SELECTION.

The logical database F1S is connected to the following executable program (report):

```
REPORT SAPMZTST.  
TABLES SBOOK.  
DATA NUMBER TYPE I VALUE 0.  
START-OF-SELECTION.  
  WRITE 'Test Program for END-OF-SELECTION'.  
  SKIP.  
  WRITE: / CITY_FR, CITY_TO.  
GET SBOOK.  
  NUMBER = NUMBER + 1.  
END-OF-SELECTION.  
  WRITE: / 'Total number of bookings:', NUMBER.
```

With the same selection as in the example of [GET <table> \[Page 1226\]](#), the output appears as follows:

```
Test Program for END-OF-SELECTION  
  
FRANKFURT          BERLIN  
Total number of bookings:      318
```

The total numbers of bookings from Frankfurt to Berlin is calculated at the event GET SBOOK and written onto the screen at the event END-OF-SELECTION.

Terminating Processing Blocks

ABAP provides several statements to leave processing blocks. Depending on the current event, different statements can branch the program flow to different processing blocks. You can distinguish between statements that leave processing blocks unconditionally or conditionally.

The following topics describe

[Leaving Processing Blocks Unconditionally \[Page 1234\]](#)

[Leaving Processing Blocks Conditionally \[Page 1238\]](#)

If you access database tables using logical databases, the termination of the processing blocks after the GET statements has some special features. Two further topics describe

[Leaving GET Events Unconditionally \[Page 1240\]](#)

[Leaving GET Events Conditionally \[Page 1244\]](#)

Leaving Processing Blocks Unconditionally

You can leave processing blocks unconditionally by

[Branching to END-OF-SELECTION \[Page 1235\]](#)

[Branching to the Output Screen \[Page 1236\]](#)

[Leaving AT Events \[Page 1237\]](#)

Branching to END-OF-SELECTION

Branching to END-OF-SELECTION

You can leave any processing block immediately and branch to the END-OF-SELECTION processing block by using the STOP statement as follows:

Syntax

STOP.

After a STOP statement, the system processes the END-OF-SELECTION processing block and stops the program.

The logical database F1S is attached to the following executable program (report):

```
REPORT SAPMZTST.  
TABLES: SPFLI, SFLIGHT, SBOOK.  
START-OF-SELECTION.  
  WRITE 'Test Program for STOP'.  
GET SBOOK.  
  WRITE: 'Bookid', SBOOK-BOOKID.  
  STOP.  
END-OF-SELECTION.  
  WRITE: / 'End of selection'.
```

The output appears as follows:

Test Program for STOP

Bookid 00010001

End of selection

After reading the first line from SBOOK, the system processes the END-OF-SELECTION statement block directly.

Branching to the Output Screen

You can leave any processing block (except of AT events) immediately and branch to the output screen by using the EXIT statement as follows:

Syntax

EXIT.

After the EXIT statement, the system displays the output list and stops the program. It does not process the END-OF-SELECTION processing block.

The EXIT statement functions differently in processing blocks of AT events (see [Leaving AT Events \[Page 1237\]](#)).

For more information about how the EXIT statement functions inside loops, see [Terminating Loops \[Page 256\]](#).

The logical database F1S is attached to the following executable program (report):

```
REPORT SAPMZTST.

TABLES: SPFLI, SFLIGHT, SBOOK.

START-OF-SELECTION.
  WRITE 'Test Program for EXIT'.

GET SBOOK.
  WRITE: 'Bookid', SBOOK-BOOKID.
  EXIT.

END-OF-SELECTION.
  WRITE: / 'End of selection'.
```

The output appears as follows:

Test Program for EXIT

Bookid 00010001

After reading the first line of SBOOK, the system processes the output list directly, but not the END-OF-SELECTION statement block.

Leaving AT Events

Leaving AT Events

If you use the EXIT statement in the processing blocks of AT events (all events that have the event keyword starting with an AT, see [Events and their Event Keywords \[Page 1211\]](#)), the system leaves this processing block and branches to the processing block of the next occurring event.

In the processing blocks of all other events, EXIT branches to the output screen.

For more information about how the EXIT statement functions inside loops, see [Terminating Loops \[Page 256\]](#).

The logical database F1S is attached to the following executable program (report):

```
REPORT SAPMZTST.
TABLES: SPFLI, SFLIGHT, SBOOK.
DATA FLAG.
AT SELECTION-SCREEN.
  IF CARRID-LOW IS INITIAL.
    FLAG = 'X'.
    EXIT.
  ENDIF.
  .....
START-OF-SELECTION.
  IF FLAG = 'X'.
    WRITE / 'No selection for CARRID made'.
    EXIT.
  ENDIF.
GET SPFLI.
  .....
GET SFLIGHT.
  .....
GET SBOOK.
  .....
END-OF-SELECTION.
  WRITE / 'End of Selection'.
```

If the user does not enter a value in the *Carrier ID* field, the output appears as follows:

No selection for CARRID made

Note that the EXIT statement is context sensitive. After the first EXIT statement, the next event, namely the START-OF-SELECTION block, is processed. After the second EXIT statement, the output list is displayed.

Leaving Processing Blocks Conditionally

You can leave any processing block conditionally by using two variants of the CHECK statement as follows:

Syntax

CHECK <condition>.

If the condition in the CHECK statement is **false**, the system leaves the processing block and branches to the processing block of the next occurring event. For <condition>, you can use any logical expression described in [Programming Logical Expressions \[Page 235\]](#).

Syntax

CHECK <seltab>.

If the contents of the table work area of the database table, to which the selection table <seltab> is attached, does not match the condition in the selection table <seltab> (see [Using Selection Tables with the CHECK Statement in GET Events \[Page 862\]](#)), the system leaves the processing block.

The CHECK statement is context sensitive. For the definition of the next occurring event during GET events, see [Leaving GET Events Conditionally \[Page 1244\]](#).

For more information about how the CHECK statement functions inside loops, see [Terminating Loops \[Page 256\]](#).

The logical database F1S is attached to the following executable program (report):

```
REPORT SAPMZTST.
TABLES: SPFLI, SFLIGHT, SBOOK.
START-OF-SELECTION.
  CHECK CITY_FR NE ''.
  WRITE: / 'Selected City-From:', CITY_FR.
  ULINE.
  CHECK CITY_TO NE ''.
  WRITE: / 'Selected City-To:', CITY_TO.
  ULINE.
GET SFLIGHT.
  WRITE: / 'Connid:', SFLIGHT-CONNID,
         'Carrid:', SFLIGHT-CARRID,
         'Fldate:', SFLIGHT-FLDATE.
```

Assume the following selection:

From	Frankfurt
To	

The first part of the output screen appears as follows:

Leaving Processing Blocks Conditionally

Selected City-From: FRANKFURT		
Connid: 0400	Carrid: LH	Fldate: 01/30/1995
Connid: 0400	Carrid: LH	Fldate: 02/01/1995
Connid: 0400	Carrid: LH	Fldate: 06/01/1995
Connid: 0400	Carrid: LH	Fldate: 06/04/1995
Connid: 0402	Carrid: LH	Fldate: 01/20/1995
Connid: 0402	Carrid: LH	Fldate: 01/22/1995
Connid: 0402	Carrid: LH	Fldate: 05/22/1995

After the second CHECK statement, the system left the START-OF-SELECTION processing block and branched to the GET SFLIGHT event.

Leaving GET Events Unconditionally

To leave the processing block of a GET event unconditionally, you have four possibilities:

[Branching to END-OF-SELECTION \[Page 1235\]](#)

[Branching to the Output Screen \[Page 1236\]](#)

[Branching to the Next Line of the Current Database Table \[Page 1241\]](#)

[Branching to the Next Line of a Superior Database Table \[Page 1243\]](#)

Branching to the Next Line of the Current Database Table

Branching to the Next Line of the Current Database Table

To leave the processing block of a GET statement and to process the next GET event at the same hierarchical level of the logical database, use the REJECT statement as follows:

Syntax

REJECT.

After this statement, the system processes the next GET event of the same database table immediately. This means that it starts the same processing block with a new line from the current table.

The REJECT statement is not context sensitive. It also has the same effect within loops and subroutines.

The logical database F1S is connected to the following executable program (report).

```
REPORT SAPMZTST.

TABLES: SPFLI, SFLIGHT, SBOOK.

GET SFLIGHT.
  SKIP.
  WRITE: / 'Connid:', SFLIGHT-CONNID,
         'Carrid:', SFLIGHT-CARRID,
         'Fldate:', SFLIGHT-FLDATE.
  ULINE.

GET SBOOK.
  IF SBOOK-BOOKID GT 00010002.
    REJECT.
  ENDIF.
  WRITE: / 'Bookid:', SBOOK-BOOKID.
```

With the same selection as in the example of [GET <table> \[Page 1226\]](#), the first part of the output screen appears as follows:

Branching to the Next Line of the Current Database Table

```
Connid: 2402 Carrid: LH Fldate: 01/30/1995
Bookid: 00010001
Bookid: 00010002
Connid: 2402 Carrid: LH Fldate: 02/01/1995
Bookid: 00010001
Bookid: 00010002
Connid: 2402 Carrid: LH Fldate: 06/01/1995
Bookid: 00010001
Bookid: 00010002
```

Only the first two bookings are displayed for each flight. Note however, that every booking is read by the system. For an example of how to improve the performance, see the example in [Branching to the Next Line of a Superior Database Table \[Page 1243\]](#).

Branching to the Next Line of a Superior Database Table

Branching to the Next Line of a Superior Database Table

To leave the processing block of a GET statement and to process the next GET event of a superior hierarchical level of the logical database, use the REJECT statement as follows:

Syntax

REJECT <dbtab>.

After this statement, the system processes the next GET event of the database table <dbtab> immediately. The database table <dbtab> must be at a superior level in the hierarchical structure in comparison to the current database table.

The logical database F1S is connected to the following executable program (report).

```
REPORT SAPMZTST.

TABLES: SPFLI, SFLIGHT, SBOOK.

GET SFLIGHT.
SKIP.
WRITE: / 'Connid:', SFLIGHT-CONNID,
       'Carrid:', SFLIGHT-CARRID,
       'Fldate:', SFLIGHT-FLDATE.
ULINE.

GET SBOOK.
IF SBOOK-BOOKID GT 00010002.
  REJECT 'SFLIGHT'.
ENDIF.
WRITE: / 'Bookid:', SBOOK-BOOKID.
```

With the same selection as in the example of [GET <table> \[Page 1226\]](#), the output is the same as for the example in [Branching to the Next Line of the Current Database Table \[Page 1241\]](#). Note, however, that in the present example, only the displayed lines are read from SBOOK because, after processing the REJECT statement, the system branches to GET SFLIGHT.

Leaving GET Events Conditionally

To leave GET events conditionally, you can use the CHECK statement. After leaving a GET event with the CHECK statement, the system processes the next GET event at the same hierarchical level of the logical database. It reads the next line of the current table. Database tables that are positioned lower in the hierarchical order of the attached logical database are not processed. Leaving a GET event conditionally with the CHECK statement works in the same way as leaving it unconditionally with the REJECT statement as described in [Branching to the Next Line of the Current Database Table \[Page 1241\]](#).

Inside GET events, you can use two variants of the CHECK statement:

Syntax

CHECK <condition>.

If the condition in the CHECK statement is **false**, the system leaves the processing block (see [Leaving Processing Blocks Conditionally \[Page 1238\]](#)).

Syntax

CHECK SELECT-OPTIONS.

If the line read from the current database table does not match the conditions in all selection tables that are connected to the database table (see [Using Selection Tables with the CHECK Statement in GET Events \[Page 862\]](#)), the system leaves the processing block.

The logical database F1S is connected to the following executable program (report).

```
REPORT SAPMZTST.
TABLES: SPFLI, SFLIGHT, SBOOK.
GET SFLIGHT.
  CHECK SFLIGHT-CARRID EQ 'LH'.
  WRITE: / 'Connid:', SFLIGHT-CONNID,
        'Carrid:', SFLIGHT-CARRID,
        'Fldate:', SFLIGHT-FLDATE.
GET SBOOK.
  CHECK SBOOK-BOOKID LT 00010003.
  WRITE: / 'Bookid:', SBOOK-BOOKID.
GET SFLIGHT LATE.
  ULINE.
```

Without filling any values into the fields of the standard selection screen, the first part of the output screen appears as follows:

Leaving GET Events Conditionally

```
Connid: 0400 Carrid: LH  Fldate: 01/30/1995
Bookid: 00010001
Bookid: 00010002

Connid: 0400 Carrid: LH  Fldate: 02/01/1995
Bookid: 00010001
Bookid: 00010002

Connid: 0400 Carrid: LH  Fldate: 06/01/1995
Bookid: 00010001
Bookid: 00010002
```

In the example above, all lines of the table SFLIGHT and, if SFLIGHT-CARRID is "LH", all lines of the table SBOOK, must be read. For performance reasons, do not program a selection as in this example, but do use the selections of the logical database as much as possible.

Features and Maintenance of Logical Databases

Logical databases are a means by which ABAP reports read and process data. **Every** ABAP executable program (report) is linked to a logical database specified in the program attributes.

A logical database has a three-character name (for example KDF) where the last letter denotes the application. If you do not specify a name for the logical database when defining the program attributes, the system uses a standard database which controls the formatting of the selection screen but does not read any data.

In this section, you learn more about

[Features of Logical Databases \[Page 1247\]](#)

[Creating and Maintaining Logical Databases \[Page 1269\]](#)

Other Sections Connected with Logical Databases

For a general introduction to database access through logical databases, see the section

[Standard Selection Screens and Logical Databases \[Page 797\]](#)

For information about how to use logical databases in connection with selection screens in reports, see the section

[Controlling the Flow of ABAP Programs Using Events \[Page 1208\]](#)

Features of Logical Databases

Logical databases facilitate efficient database accesses and provide a user interface which is easy to use and generate.

The following sections describe

[Tasks of Logical Databases \[Page 1248\]](#)

[Basic Features of Logical Databases \[Page 1249\]](#)

[Authorization Checks with Logical Databases \[Page 1262\]](#)

[Performance Aspects of Logical Databases \[Page 1263\]](#)

For an example of a logical database, see under

[Example of a Logical Database \[Page 1265\]](#)

Tasks of Logical Databases

Logical databases allow you to program several different tasks centrally. For instance, the layout of the user interface and the database accesses are coded centrally in the logical database in order to save the application logic of the executable program (report) from having to deal with the technical details. You use logical databases to perform the following tasks:

- If several executable programs (reports) read the same data, you can code the read accesses in a single logical database. For the individual executable programs (reports), you then no longer need to know the exact structure of the database tables involved (particularly the foreign key dependencies). However, you can be sure that the entries will be retrieved in the correct order when the GET event is executed.
- If you want to use the same user interface for several executable programs (reports), you can implement this easily with the selection screens of logical databases. To achieve the necessary flexibility, you can generate your own selection screen versions.
- Authorization checks for central (and sensitive) data are coded centrally in logical databases so that they are not affected by individual executable programs (reports).
- If you want to improve response times, logical databases permit you to take a number of measures to achieve this (e.g. using views instead of nested SELECT statements). These become immediately effective in all executable programs (reports) concerned and save you from having to modify their source code.

Basic Features of Logical Databases

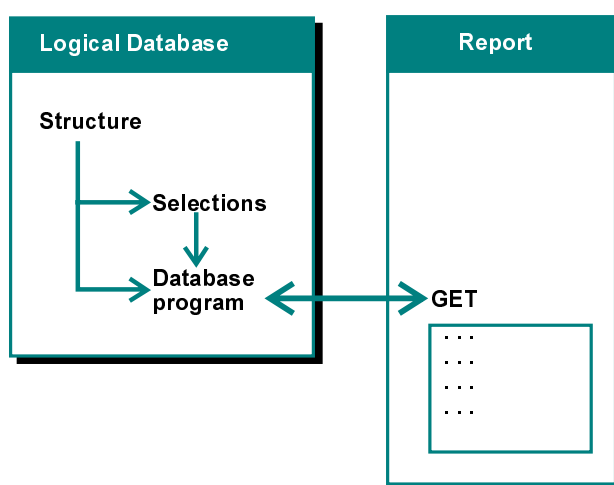
Basic Features of Logical Databases

The following definition explains the basic function of logical databases:

Definition

ABAP reports use logical databases to read and process data. The order in which data is made available to the executable program (report) depends on the hierarchical structure of the logical database concerned. Logical databases also supply the interface for a user dialog (i.e. the selection screen) and incorporate user input checks and error dialogs. You can modify and extend this interface in the executable program (report).

This definition is realized by the layout of logical databases:



Layout

A logical database consists of at least the following three components:

- **Structure**

The structure is the basic component of a logical database. It determines the structure of the other components and the behavior of the logical database at runtime.

[Structure of Logical Databases \[Page 1251\]](#)

- **Selections**

This component determines the user interface of each executable program (report). Its layout is usually determined by the structure. You can adjust and extend the selections to suit your requirements.

[Logical Database Selections \[Page 1252\]](#)

- **Database Program**

The database program is a collection of subroutines which select the data and pass it to the executable program (report). The layout of the database program is determined by both the structure and the selections. You can adjust and extend the database program to suit your requirements.

[Database Program of a Logical Database \[Page 1254\]](#)

Other components such as documentation, language-specific texts, and user-defined selection screens extend the functionality further.

Logical databases allow you to modularize applications used in reports:

Logical Databases and Reports

The subroutines in a logical database program and the processing blocks of the executable program (report), in which the logical database is specified as an attribute, constitute a **modularized** system for performing database accesses. Apart from the structure and selections of the logical database, the GET statements in the executable program (report) determine behavior of the database at runtime.

[Logical Databases and Reports \[Page 1258\]](#)

Structure of Logical Databases

Structure of Logical Databases

In general, the structure of logical databases reflects the foreign key dependencies of hierarchical tables in the SAP System (see also [Accessing Data Using Logical Databases \[Page 1202\]](#)).

Logical databases have a hierarchical structure which can be defined as follows:

- There is just one node at the highest level. This is known as the root node.
- Each node can have one or several branches.
- Each node is derived from one other node.

Nodes must be defined structures in the ABAP Dictionary. Normally, these are the structures of database tables which the logical database reads and passes to ABAP reports for further evaluation. However, it is also possible, and sometimes useful, to use ABAP Dictionary structures without an underlying database.

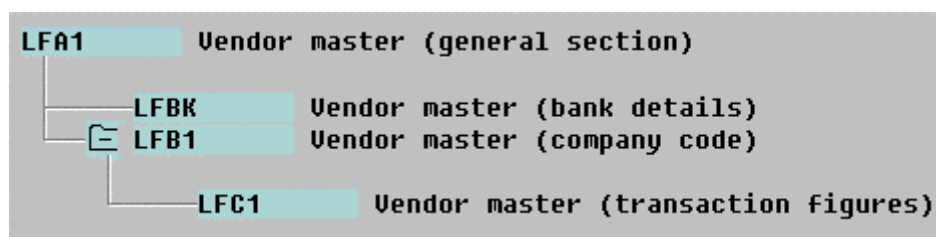
For technical reasons, there is an upper limit (MAX) to the possible number of nodes in a logical database structure. This limit is calculated as follows:

- $LEN = \text{maximum length of a name in the structure (e.g. 7)}$.
- $MAX = 1200 \text{ DIV } LEN \text{ (e.g. } 1200 \text{ DIV } 7 = 171)$.

An ABAP executable program (report) can contain a GET statement for each node in the structure of a logical database. At runtime, the processing blocks are executed in the order in which they are defined by the hierarchical structure.

If an executable program (report) does not contain a GET statement for every node of a logical database, the processing passes through all the nodes which lie in the path from the root to the nodes specified by GET statements.

Suppose LFA1 is the root node, LFBK and LFB1 are branches of LFA1, and LFC1 is a branch of LFB1.



If the executable program (report) contains GET statements for all nodes, the GET events are executed in the order LFA1, LFBK, LFB1, LFC1.

If the executable program (report) contains only one single GET statement for LFB1, the processing only passes through LFA1 and LFB1.

For further information about how the selections and the database program are affected by the structure, see [Creating and Maintaining Logical Databases \[Page 1269\]](#).

Logical Database Selections

In a logical database, you can define input fields on the selection screen with the SELECT-OPTIONS and PARAMETERS statements. This is achieved with the help of a special include program known as the selection include. In each ABAP executable program (report), you can extend these logical database selections by defining executable program (report)-specific selections. All program-specific selections are displayed after the database-specific selections.

When generating the selection screen of an executable program (report), the system only takes into account the database-specific selection criteria and parameters, whose corresponding tables (defined by the FOR option of the SELECT-OPTIONS and PARAMETERS statements in the selection include, see [Editing Selections \[Page 1278\]](#)) are declared by the TABLES statement in the program.

Suppose a logical database program contains the following lines:

```
SELECT-OPTIONS SLIFNR FOR LFA1-LIFNR.  
PARAMETERS   PBUKRS LIKE LFB1-BUKRS FOR TABLE LFB1.
```

The selection criterion SLIFNR is linked to table LFA1, the parameter PBUKRS to table LFB1.

If the TABLES statement in a executable program (report) declares LFA1 but not LFB1, SLIFNR is displayed on the selection screen, but PBUKRS does not appear.

You can format the selection screen (for example by defining boxes, pushbuttons, radio buttons and blank lines, or by writing several PARAMETERS in one line) with the SELECTION-SCREEN statement in the logical database include program (see [SELECTION-SCREEN - Formatting \[Page 832\]](#)).

You can display possible input values and field documentation for selection screen fields by using the additions VALUE-REQUEST and HELP-REQUEST with the SELECT-OPTIONS and PARAMETERS statements (see the appropriate keyword documentation).

Dynamic Selections

Dynamic selections allow the user to define further selections for database accesses in addition to the selection criteria already defined in the logical database selection include. For performance reasons, selection criteria should be used with the CHECK statement during a GET event in the executable program (report) only for selections which are not table-specific (see [Leaving GET Events Conditionally \[Page 1244\]](#)). Otherwise, the selection is not performed until after the database access.

On the other hand, dynamic selections are already effective during the database access in the logical database. To support dynamic selections for a database table <dbtab>, you must specify the following statement in the selection include of the logical database:

```
SELECTION-SCREEN DYNAMIC SELECTIONS FOR TABLE <dbtab>.
```

In this case, the *Dynamic selections* pushbutton is displayed on the selection screen if the table <dbtab> is used in the executable program (report) (see also keyword documentation for SELECTION-SCREEN). Pressing this button enables the user to enter dynamic selections for the fields defined by the logical database (see also [Standard Selection Screens and Logical Databases \[Page 797\]](#)). The logical database program then can use these in dynamic WHERE conditions when selecting data for the event keyword GET <dbtab> (see [Working with Dynamic Selections \[Page 1287\]](#)).

Logical Database Selections

In the ABAP Development Workbench, the user can define the field list for the dynamic selections in the form of a logical database selection view. These views are identified by their origin ('SAP' or 'CUS' for 'CUSTomer'), the logical database name, and a name which, for the functionality with selection screens described here, must always be 'STANDARD'. The system uses selection views with origin 'SAP' only if none with origin 'CUS' have been created. This way, users can define the best logical database selection views for their requirements.

The system transfers also the values of executable program (report) internal selection criteria to the logical database, if they are defined for the columns of a database table that is designated for dynamic selections (see [Program-specific Selection Criteria and Logical Databases \[Page 820\]](#)).

Defining Selection Screen Versions

The logical database selection screen (screen number 1000) has a standardized layout where selection criteria and parameters appear on separate lines in the order in which they are declared. The system generates this screen automatically for every executable program (report) which does not have its own selection screen specified in the attributes.

If you want to suppress certain input fields from the logical database selection screen for a report, you can define a selection screen version (with a screen number less than or equal to 999) in the selection include and enter this in the program attributes. By pressing F4 there, you can get an overview of the selection screen versions defined in the logical database concerned. You define selection screen versions with the SELECTION-SCREEN BEGIN|END OF VERSION and SELECTION-SCREEN EXCLUDE statements. The latter statement allows you to specify any objects you want to exclude from the selection screen version (see [Editing Selections \[Page 1278\]](#)). With the SELECTION-SCREEN statement, you can design the layout further.

If the program attributes contain the number of a selection screen version, the system uses this as the model when generating the selection screen. The screen flow logic is also generated automatically for own selection screen versions and therefore cannot be modified. Especially, deleting database-specific selections is not allowed.

Selection screen versions replace the user-specific selection screens used prior to Release 3.0.

For further information on selection screens and the PARAMETERS, SELECT-OPTIONS and SELECTION-SCREEN statements, see [Working with Selection Screens \[Page 795\]](#).

Database Program of a Logical Database

The name of the database program of a logical database <dba> conforms to the naming convention SAPDB<dba>. Essentially, it contains a collection of subroutines which the system calls at runtime of an ABAP executable program (report).

The interaction of these subroutines with the event keywords in the executable program (report) is described in the section [Logical Databases and Reports \[Page 1258\]](#).

The structure of the logical database determines the behavior of the PUT statement which is of particular importance in the subroutine PUT_<table> (see below).

The logical database program usually contains the following subroutines, all defined with the FORM statement, as described in the section [Defining Subroutines \[Page 445\]](#).

- FORM INIT
Called once before the selection screen is displayed.
- FORM PBO
Called each time before the selection screen is refreshed.
- FORM PAI
Called each time the user presses ENTER on the selection screen.
The system passes the parameters FNAME and MARK, which are defined and filled automatically, to the subroutine.
 - FNAME contains the name of a selection criterion or parameter on the selection screen.
 - MARK describes the selection made by the user:
MARK = SPACE means that the user has entered a simple single value or range selection.
MARK = '*' means that the user has also made entries on the *Multiple Selection* screen.
- FORM PUT_<table>
This subroutine is called in the order determined by the structure of the logical database. The data of the node <table> is read using SELECT statements and a PUT statement directs the program flow to the appropriate GET statement in the executable program (report). The PUT statement is the central statement in this subroutine:

Syntax

PUT <table>.

You can only use the PUT <table> statement in a subroutine of a logical database which contains the node <table> with a name beginning with PUT_<table>.

The PUT statement directs the program flow according to the structure of the logical database. The read depth is determined by the GET statements in the executable program (report) concerned.

Database Program of a Logical Database

First, the subroutine PUT_<root> is executed for the root node. The PUT statement then directs the program flow as follows:

1. If the database program contains the subroutine AUTHORITY_CHECK_<table>, this subroutine is executed first.
2. The PUT statement attempts to trigger a GET event in the executable program (report), that is, provided a relevant GET <table> statement exists, the appropriate block of code is executed.
3. The PUT statement directs the program flow
 - to the subroutine for the next node, if there is a GET statement for a node at a lower level of the relevant branch in the executable program (report).
 - to the subroutine of a node at the same level, if the preceding node branches to such a node and if a GET statement exists for such a node in the executable program (report).

The PUT statement starts with step 1 again. When it reaches the subroutine for the node at the lowest level of a branch for which there is a GET statement in the executable program (report), it does not branch further, but continues the processing of the current subroutine. When a subroutine PUT_<table> has been executed in its entirety, the program flow returns to the PUT statement from which it branched to the subroutine PUT_<table>.

4. After returning from a subordinate subroutine PUT_<table>, the PUT statement branches to the GET <table> LATE statement of the executable program (report), if one exists.

In the logical database structure, LFB1 is a branch of LFA1.

Suppose you define the following selection criteria in the selection include program:

```
SELECT-OPTIONS: SLIFNR FOR LFA1-LIFNR,
                SBUKRS FOR LFB1-BUKRS.
```

A section of the database program would then read:

```
FORM PUT_LFA1.
  SELECT * FROM LFA1
    WHERE LIFNR IN SLIFNR.
  PUT LFA1.
ENDSELECT.
ENDFORM.
```

```
FORM PUT_LFB1.
  SELECT * FROM LFB1
    WHERE LIFNR = LFA1-LIFNR.
    AND  BUKRS IN SBUKRS.
  PUT LFB1.
ENDSELECT.
ENDFORM.
```

An executable program (report) linked to the logical database would contain the lines:

Database Program of a Logical Database

```

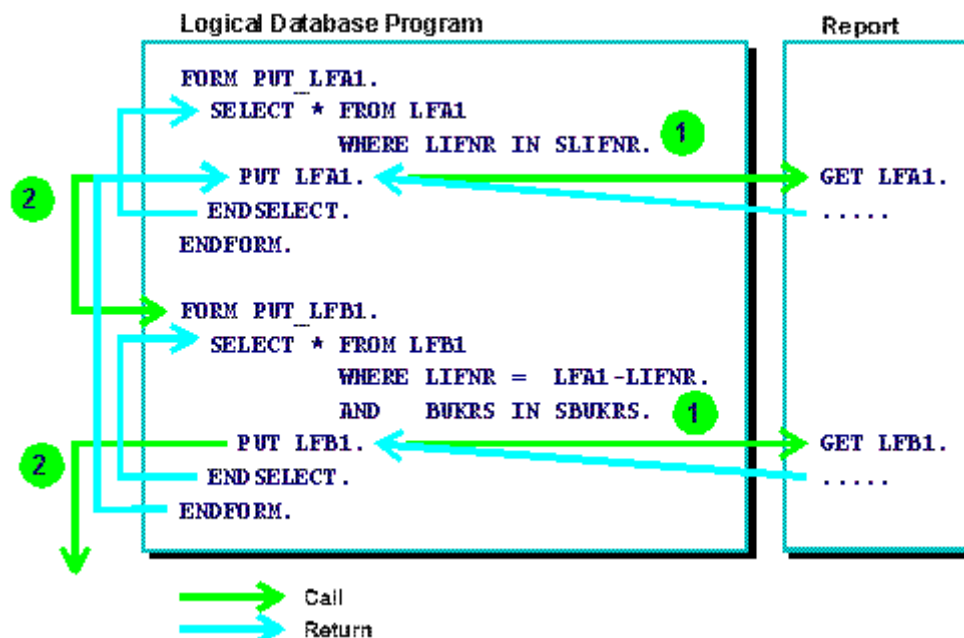
GET LFA1.
WRITE LFA1-LIFNR.

GET LFB1.
WRITE LFB1-BUKRS.

```

In this example, the system calls the routine PUT_LFA1 at the beginning of the selection process. The PUT LFA1 statement directs the program flow to the processing block of the GET LFA1 statement in the executable program (report). When this block has been executed, PUT LFA1 gives way to the subroutine PUT_LFB1 which directs the program flow to the GET LFB1 statement in the executable program (report). If LFB1 is the last node to be read, processing resumes with the SELECT loop in PUT_LFB1. Otherwise, the program flow moves to the subroutine PUT_<table> of the next node. At the end of the SELECT loop of the last node, processing resumes in the SELECT loop of the node at the next level up.

The following picture shows the program flow:



In this example, the PUT statements do not branch to the authorization check subroutines.

- FORM AUTHORITY_CHECK_<table>

Called automatically by the PUT <table> statement. In this subroutine, you can specify authorization checks for the appropriate node <table> from the structure of the logical database.

- FORM PUT_<dba>_MATCHCODE

Called in the case of a matchcode selection with the selected matchcode records. <dba> is the name of the logical database. From this subroutine, you can use the matchcode records to read the relevant entries from the root node <root>. The processing in the executable program (report) can then be called with PUT <root>.

Database Program of a Logical Database

- FORM BEFORE_EVENT, AFTER_EVENT

Called before or after an event, the name of which is passed in the parameter EVENT.

You may assign a value to the field EVENT with

EVENT = 'START-OF-SELECTION'.

and then use it in the USING list of the subroutines BEFORE_EVENT and AFTER_EVENT.

- FORM <par>_VAL, <selop>_VAL, <selop>-LOW_VAL, <selop>-HIGH_VAL

Called when the user presses F4 to get a list of possible entries for the input fields of the parameter <par> or the selection criterion <selop> (both database-specific) on the selection screen.

- FORM <par>_HLP, <selop>_HLP, <selop>-LOW_HLP, <selop>-HIGH_HLP

Called when the user presses F1 to get help on the input fields of the parameter <par> or the selection criterion <selop> (both database-specific) on the selection screen.

For further information on this topic, refer to the following:

- [Creating and Maintaining Logical Databases \[Page 1269\]](#)
- Keyword documentation for SELECT-OPTIONS with the additions VALUE-REQUEST and HELP-REQUEST
- Keyword documentation for PARAMETERS with the additions AS MATCHCODE STRUCTURE, VALUE-REQUEST, and HELP-REQUEST

Logical Databases and Reports

Logical Databases when Generating an Executable ABAP program (report)

Each ABAP executable program (report) is linked to the logical database specified in the program attributes. This logical database affects the generation of the executable program (report) as follows:

- It generates the selection screen, which contains the selections (selection criteria and parameters) of the logical database **and** the executable program (report).
- However, **only** the database-specific selections relevant for the data evaluation in the executable program (report) (determined by the TABLES statement) are displayed.

Runtime Behavior of an Executable Program (Report) Linked to a Logical Database

When you run an executable program (report) linked to a logical database, the system calls a series of processing blocks in a particular order (see also bc150e.doc047). Some of the processing is coded in the executable program (report) itself, some in the logical database program.

The database-specific subroutines are executed in the database program SAPDB<dba> (see [Database Program of a Logical Database \[Page 1254\]](#))

The processing blocks for the events are executed in the ABAP executable program (report) (for further information about all events and examples, see [Events and their Event Keywords \[Page 1211\]](#))

The following list contains the processing steps which the system executes for an ABAP executable program (report) linked to a logical database <dba>. In each case, the lines of ABAP program code specify the processing blocks (subroutines and events) which belong to these steps.

1. PBO, initializations before the selection screen is displayed (for example, default values for key data).
 - Subroutines:
FORM INIT
This subroutine is called once before the selection screen is first displayed.
FORM PBO.
This subroutine is called each time the selection screen is refreshed (after the user presses ENTER).
 - Events:
INITIALIZATION.
This event occurs once before the selection screen is first displayed (see [INITIALIZATION \[Page 1213\]](#)).
AT SELECTION-SCREEN OUTPUT.
This event occurs each time the selection screen is refreshed (see [PBO of the Selection Screen \[Page 1224\]](#)).
2. The system displays the selection screen and the user enters data in the input fields.

Logical Databases and Reports

3. Possible entries and help, in case the user presses F4 or F1 on the selection screen.
 - Subroutines:
FORM <par>_VAL.
FORM <selop>_VAL.
FORM <selop>-LOW_VAL.
FORM <selop>-HIGH_VAL.
If the user requests a list of possible entries (F4) for database-specific parameters <par> or selection criteria <selop>, these subroutines are called as required.

If the user requests help (F1) for these parameters, the subroutines are called with the ending _HLP instead of _VAL.
 - Events:
AT SELECTION-SCREEN ON VALUE-REQUEST FOR <par>.
AT SELECTION-SCREEN ON VALUE-REQUEST FOR <selop>-LOW.
AT SELECTION-SCREEN ON VALUE-REQUEST FOR <selop>-HIGH.
If the user requests a list of possible entries (F4) for program-specific parameters <par> or selection criteria <selop>, these events occur (see [Creating a List of Input Values \[Page 1220\]](#)).

If the user requests help (F1) for these parameters, the events with the addition ON HELP-REQUEST occurs instead of ON VALUE-REQUEST (see [Creating Help for Input Fields \[Page 1221\]](#)).
4. PAI, the system checks whether the user input is correct, complete, and plausible, as well as the user authorizations. If it detects an error, it conducts a dialog with the user and makes some of the fields ready for input again so that the error can be corrected.
 - Subroutine:
FORM PAI USING FNAME MARK.

The fields FNAME and MARK are determined and filled by the system.

FNAME contains the name of a selection criterion or parameter on the selection screen.

If MARK = SPACE, the user has entered a simple single value or range selection.

If MARK = '*', the user has also entered selections on the *Multiple Selection* screen.

Using the combination FNAME = '*' and MARK = 'ANY', you can check all entries at once after the user has chosen Enter.
 - Events:
AT SELECTION-SCREEN ON <fname>.

Event after processing a particular input field. The field <fname> must be specified in the executable program (report) (see [Processing a Particular Input Field \[Page 1218\]](#)).

AT SELECTION-SCREEN ON END OF <fname>.

Event after processing multiple selections. The field <fname> must be specified in the executable program (report) (see [Processing Multiple Selection \[Page 1219\]](#)).

AT SELECTION-SCREEN.

Event after the user displays the entire selection screen by choosing Enter. See [AT SELECTION-SCREEN \[Page 1216\]](#).

5. Data selection in the logical database and processing in the ABAP executable program (report)

- Subroutine:

FORM PUT_<table>.

The logical database reads the selected data of the node <table>.

- Events:

START-OF-SELECTION.

In this event, the ABAP executable program (report) carries out preparatory work (e.g. importing data from files). See [START-OF-SELECTION \[Page 1225\]](#).

GET <table> [LATE].

The executable program (report) processes the data read from <table> in the order determined by the structure of the logical database (see [GET <table> \[Page 1226\]](#) and [GET <table> LATE \[Page 1230\]](#)).

END-OF-SELECTION.

In this event, the ABAP executable program (report) performs concluding operations (e.g. calculating totals, exporting data to files). See [END-OF-SELECTION \[Page 1232\]](#).

Suppose TABLE1 is the root node and TABLE2 is its only subordinate node in a logical database. In this case, the nesting of the processing steps for data selection and processing would be as follows:

1. START-OF-SELECTION.

Preparatory steps in the executable program (report)

2. FORM PUT_TABLE1.

Loop to read TABLE1 in the database program

3. GET TABLE1.

Processing of data from TABLE1 in the executable program (report)

4. FORM PUT_TABLE2.

Loop to read TABLE2 in the database program

5. GET TABLE2.

Processing of data from TABLE2 in the executable program (report)

6. GET TABLE1 LATE.

End of loop on TABLE1, processing of data in the executable program (report)

7. END-OF-SELECTION.

Concluding steps in the executable program (report)

The subroutines

Logical Databases and Reports

- PUT_<dba>_MATCHCODE
- BEFORE_EVENT
- AFTER_EVENT

are called by the system at the appropriate points in the program flow.

You can place the subroutine AUTHORITY_CHECK_<table> in the executable program (report) according to the authorization checks you want to make (see [Authorization Checks with Logical Databases \[Page 1262\]](#)).

For further information about database programs, see [Editing the Database Program \[Page 1281\]](#).

Authorization Checks with Logical Databases

In general, you can include authorization checks in the following subroutines of the database program or processing blocks of the executable program (report):

- Subroutines in the database program:
 - PAI
 - AUTHORITY_CHECK_<table>
- Event keywords in the executable program (report):
 - AT SELECTION-SCREEN
 - AT SELECTION-SCREEN ON <fname>
 - AT SELECTION-SCREEN ON END OF <fname>
 - GET <table>

Whether you place the authorization checks in the database program or in the executable program (report) depends on the following:

- The structure of the logical database; you should, for example, only check a company code authorization if a line containing the company code field is read at runtime.
- Performance; do not, for example, perform repeated checks within SELECT loops.

In any case, the separation of database accesses and application logic allows you to code **all** authorization checks centrally in the logical database program. This makes it easier to maintain large programming systems.

Performance Aspects of Logical Databases

Since changes to a logical database are immediately effective in all ABAP reports concerned, you can improve the response times of many different objects in your program library by making optimizations centrally.

You can achieve the greatest improvements in performance by allowing the user to specify exactly which table entries the system is supposed to read from the database. To do this, you use the following techniques in the database program:

- Selection criteria and parameters (see [Working with Selection Screens \[Page 795\]](#)), possibly with default values and value lists.
- Dynamic selections (see [Logical Database Selections \[Page 1252\]](#)).
- Matchcode selection (see [Matchcode Selections \[Page 1297\]](#)).
- Views or store the entries read from the database in internal tables.

In addition, you should perform authorization checks at an early stage, that is as soon as possible during the processing of the selection screen rather than waiting until data selection.

Since they depend very much on the data to be read, there are no established procedure rules for optimizations. You should therefore be aware of the following points when attempting to optimize response times:

- The numerical relationship between the table contents at different levels of the structure is very important.

If one line of a database table at one level of the structure includes exactly one line of the database table at the next level (case A), other optimizations may make more sense for a 1:100 or 1:1000 ratio (case B).

- In case A, you can improve response times by using database views (for further information about views, see the documentation [ABAP Dictionary \[Ext.\]](#)).
- In case B, you can use internal tables. You first read the data from the database into an internal table in one operation (see [Reading Data into an Internal Table \[Page 555\]](#)). Then, you can process the internal table with LOOP/ENDLOOP within the logical database.

In case B, it may also be useful to process the selected lines using a cursor (see [Reading Lines of Database Tables Using a Cursor \[Page 588\]](#)).

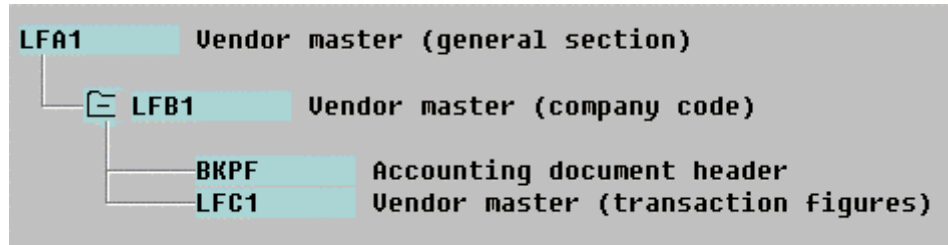
- Some ABAP reports refer to only part of the hierarchical structure with GET statements, while other reports access all nodes in the structure. In this case, you have the following options for improving the performance of individual reports:
 - In the logical database program, use the table GET_EVENTS. After generating a executable program (report) with the logical database, this table indicates whether each GET statement in the executable program (report) occurs or not, for each node of the structure (see [Editing the Database Program \[Page 1281\]](#)).
 - By using the
SELECTION-SCREEN FIELD SELECTION FOR TABLE <table>.

statement in the selection include, you can designate a database table <table> in the logical database for field selection. In the logical database program, you can then use the SELECT statement with a list instead of using SELECT * (see [Working with Field Selections \[Page 1291\]](#)). In the executable program (report), you can then use the appropriate GET statement with a list of fields to fill the SELECT list (see [Specifying Fields of the Database Table Explicitly \[Page 1229\]](#)).

Example of a Logical Database

Example of a Logical Database

Suppose the logical database HKS has the following **structure**:



Suppose the following selection criteria are defined in the selection include:

```

SELECT-OPTIONS: SLIFNR  FOR LFA1-LIFNR,
                  SBUKRS  FOR LFB1-BUKRS,
                  SGJAHR  FOR LFC1-GJAHR,
                  SBELNR  FOR BKPF-BELNR.
  
```

Below, you see the complete **database program**:

```

*-----*
* DATABASE PROGRAM OF THE LOGICAL DATABASE HKS
*-----*
PROGRAM SAPDBHKS DEFINING DATABASE HKS.
TABLES: LFA1,
        LFB1,
        LFC1,
        BKPF.

*-----*
* Initialize selection screen (process before PBO)
*-----*
FORM INIT.
....
ENDFORM.          "INIT

*-----*
* PBO of selection screen (process always after ENTER)
*-----*
FORM PBO.
....
ENDFORM.          "PBO

*-----*
* PAI of selection screen (process always after ENTER)
*-----*
FORM PAI USING FNAME MARK.
CASE FNAME.
  WHEN 'SLIFNR'.
    ....
  WHEN 'SBUKRS'.
    ....
  WHEN 'SGJAHR'.
    ....
  
```

```

      ....
      WHEN 'SBELNR'.
      ....
      ENDCASE.
      ENDFORM.                  "PAI

*-----*
* Call event GET LFA1
*-----*
FORM PUT_LFA1.
  SELECT * FROM LFA1
    WHERE LIFNR   IN SLIFNR.
  PUT LFA1.
  ENDSELECT.
  ENDFORM.                  "PUT_LFA1

*-----*
* Call event GET LFB1
*-----*
FORM PUT_LFB1.
  SELECT * FROM LFB1
    WHERE LIFNR   = LFA1-LIFNR
    AND BUKRS     IN SBULRS.
  PUT LFB1.
  ENDSELECT.
  ENDFORM.                  "PUT_LFB1

*-----*
* Call event GET LFC1
*-----*
FORM PUT_LFC1.
  SELECT * FROM LFC1
    WHERE LIFNR   = LFA1-LIFNR
    AND BUKRS     = LFB1-BUKRS
    AND GJAHR     IN SGJAHR.
  PUT LFC1.
  ENDSELECT.
  ENDFORM.                  "PUT_LFC1

*-----*
* Call event GET BKPF
*-----*
FORM PUT_BKPF.
  SELECT * FROM BKPF
    WHERE BUKRS   = LFB1-BUKRS
    AND BELNR     IN SBELNR
    AND GJAHR     IN SGJAHR.
  PUT BKPF.
  ENDSELECT.
  ENDFORM.                  "PUT_BKPF

```

The PROGRAM statement contains the addition DEFINING DATABASE HKS which defines the database program as belonging to the logical database HKS.

The nodes of the structure are declared with the TABLES statement which generates the appropriate table work areas. Since these table work areas are

Example of a Logical Database

shared by the database program and the reports concerned, they become the interface for data transfer between the logical database and the executable program (report).

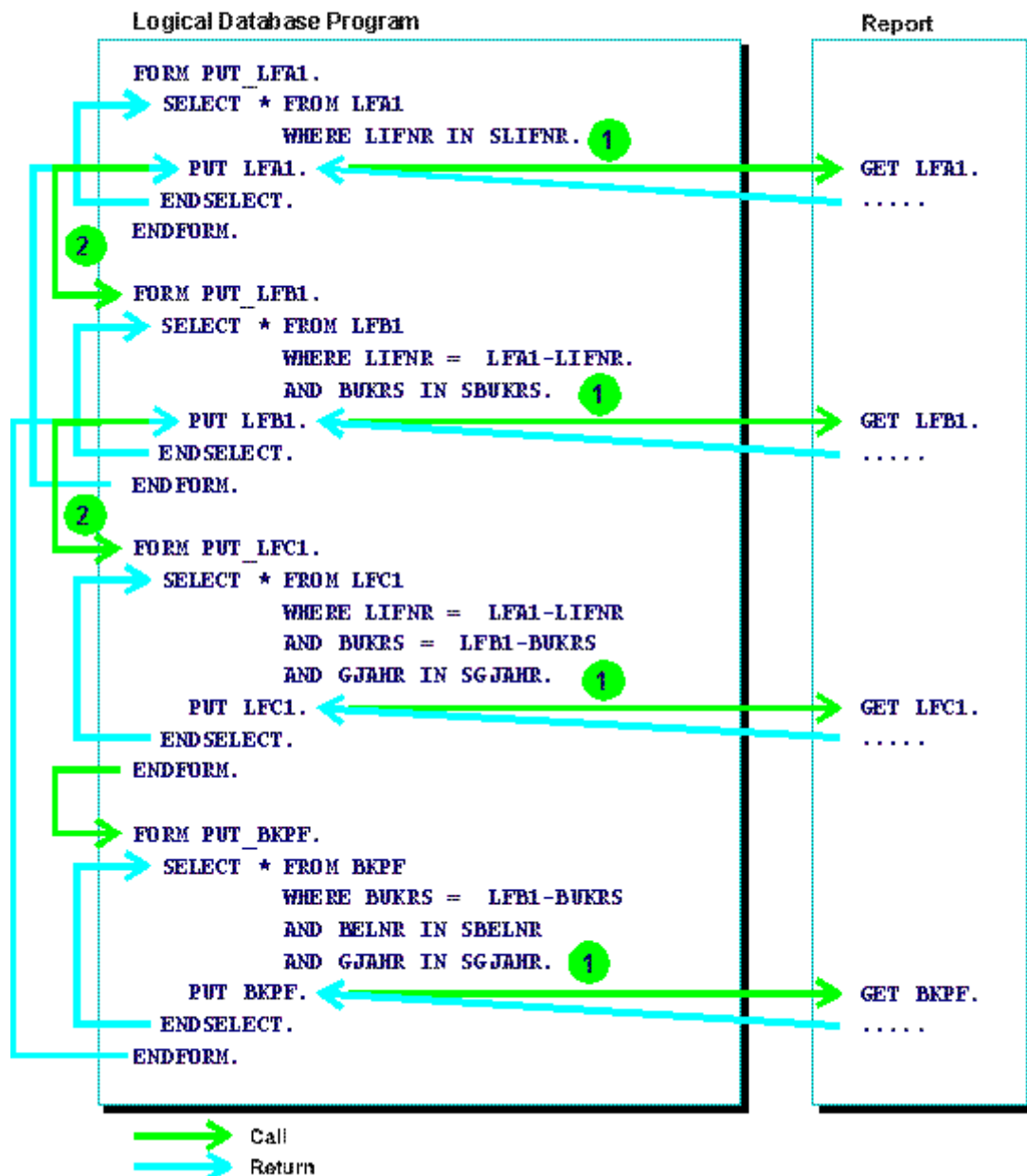
The subroutines INIT and PBO initialize the selection screen.

The subroutine PAI performs an authorization check on what the user has entered on the selection screen. Plausibility or value range checks are also possible. If a check produces a negative outcome, an appropriate error dialog is performed and the field concerned once again becomes ready for input.

The PUT_<table> subroutines read the database tables according to the selection criteria entered by the user and call the relevant processing blocks in the executable program (report). The order in which the subroutines are called is determined by the structure of the logical database.

The following picture shows the program flow determined by the database structure:

Example of a Logical Database



Creating and Maintaining Logical Databases

Creating and Maintaining Logical Databases

The transaction for creating or maintaining logical databases is SE36 or SLDB. To proceed, choose *Tools* → *ABAP Workbench* → *Development* → *Programming environ.* → *Logical databases*.

You then see the initial screen which is displayed as follows:

The screenshot shows the initial screen of the SE36 or SLDB transaction. At the top, there are three icons: a magnifying glass, a trash can, and a document. Below these, there is a text field labeled 'Logical database' and a button labeled 'Create' with a document icon. Underneath, there is a section titled 'Sub-objects' containing a list of radio buttons: 'Structure' (selected), 'Selections', 'Database program', 'Selection texts', 'Matchcode selection', and 'Documentation'. At the bottom, there are two buttons: 'Display' with a magnifying glass icon and 'Change' with a pencil icon.

In the field *Logical database*, enter the name of a logical database.

To display or change a logical database with *Display* or *Change*, select a logical database sub-object. To create a logical database, choose *Create*.

The following topics discuss how to create and maintain logical databases:

[Creating Logical Databases \[Page 1270\]](#)

[Processing the Structure \[Page 1273\]](#)

[Editing Selections \[Page 1278\]](#)

[Editing the Database Program \[Page 1281\]](#)

[Editing Selection Texts \[Page 1294\]](#)

[Editing Matchcode Selections \[Page 1297\]](#)

[Editing Documentation \[Page 1301\]](#)

[Further Editing Options \[Page 1302\]](#)

Creating a Logical Database Creating a Logical Database

When you create a logical database, the system does much of the work for you:

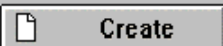
- You define the most important features of the logical database by defining its structure in a graphical editor.
- When you have defined the structure, the system automatically proposes the selection include.
- Finally, the system uses the structure and selections to generate a database program.

To make full use of the automatic generation of ABAP statements, you should process the sub-components in the following order:

To create a new logical database, choose *Create* on the initial screen.

1. In the dialog box which follows, enter a short text, confirm with *Create* and specify a development class (for further information about development classes, see [Maintaining Program Attributes \[Page 74\]](#)).

Creating a Logical Database Creating a Logical Database



Logical database **HKS** 

Create Logical Database

Database **HKS**

Short text **Demo for ABAP/4 User's Guide**


Master language **English**

Maintain Object Catalog Entry

Object **R3TR LDBA HKS**

Attributes



Development class 

Author **KELLERH**

System name **S11**

Master language ☐

Repaired ☐



 Local object 

If you want to change the short text or maintain it in other languages, you can do this later by choosing *Extras* → *Short text*.

2. Specify the root node of the structure, for example:

Create root node

Name of the root node of the structure **LFA1**

Creating a Logical Database

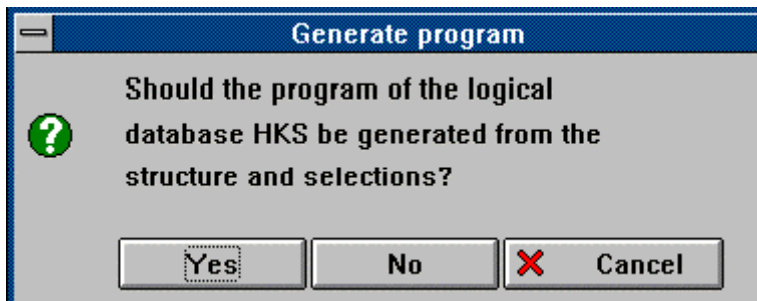
Confirm with *Create*.

3. Your logical database now has a structure with a single node.

LFA1 Vendor master (general section)

You can extend this structure as described in [Changing Structures \[Page 1276\]](#).

4. Save the structure. The system automatically proposes a selection include on the basis of this.
5. Choose *Goto* → *Selections* and maintain the include program as described under [Editing Selections \[Page 1278\]](#).
6. Save the selections and choose *Goto* → *Database program*. Confirm the following dialog box:



The system then generates a database program from your structure and selection conditions. All the necessary subroutines are already defined and most WHERE conditions of the SELECT statements are also generated. Maintain this database program as described under [Editing Database Programs \[Page 1281\]](#).

7. Finally, you can maintain the following optional sub-objects:
 - The selection texts that appear on selection screens (see [Editing With Selection Screens \[Page 1294\]](#))
 - Matchcode selection (see [Editing Matchcode Selections \[Page 1297\]](#)).
 - Matchcode selection (see [Editing Documentation \[Page 1301\]](#)).

Although the support you get from the system helps you to create an executable logical database quickly, you must take care of details such as improving performance yourself.

Processing the Structure

To display or change logical database structures, choose *Structure* on the initial screen.

You can do one of the following

[Displaying Structures \[Page 1274\]](#)

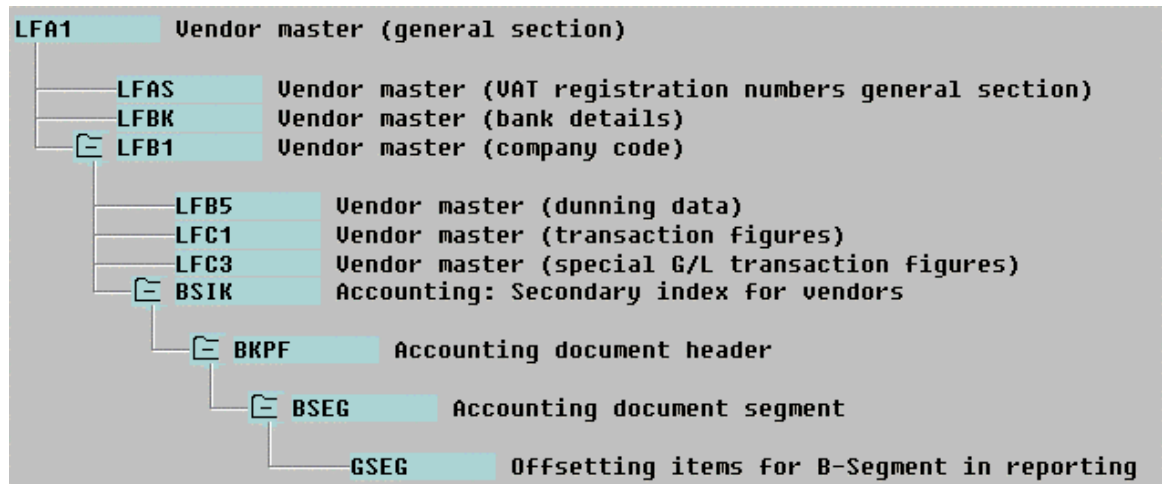
[Changing Structures \[Page 1276\]](#)

Displaying Structures

To display a structure, select *Structure* and choose *Display* on the initial screen.

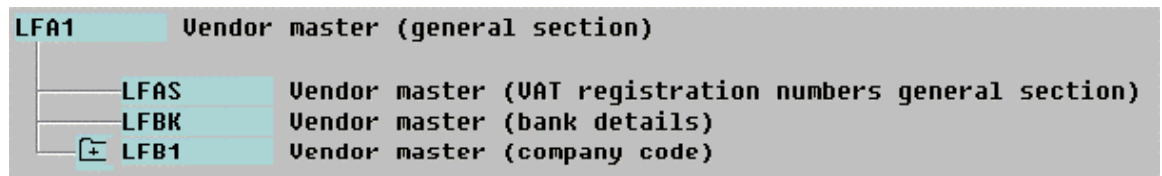
The system displays the name of the node at the highest level (the root node of the structure) in the top left corner. Each subordinate node appears underneath the previous one, indented to the right, while nodes at the same level are displayed in the same column.

The logical database KDF has the structure:



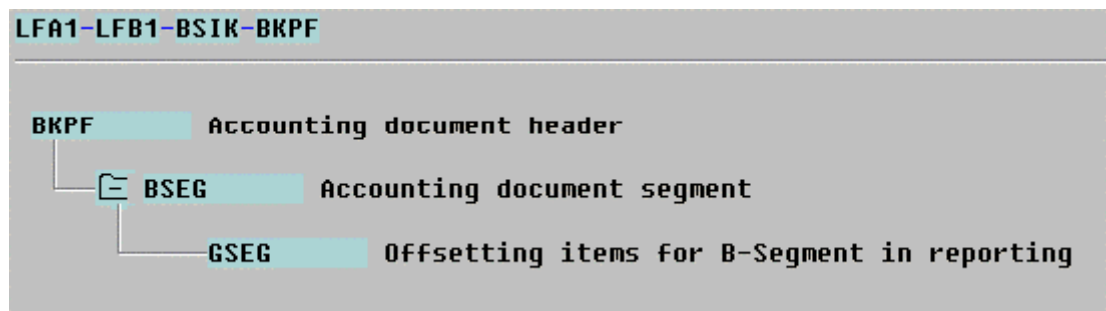
You can expand or collapse the subordinate hierarchy structure for one node by clicking on it. When you collapse a node, only the root of the sub-tree is displayed and prefixed with +. You can do this also by choosing *Edit* → *Sub-tree*.

Collapse all subordinate nodes of node LFB1 in KDF:



To display just the sub-tree of a particular node, select the node with the cursor and choose *Edit* → *Sub-tree* → *Display*. The first line of the result screen then additionally shows the path from the root to the selected sub-tree. You can also click on a node in this path to display the corresponding sub-tree.

Displaying Structures



To display the individual fields of a node, place the cursor on the node and double-click or choose *Display Table fields*.

The fields of the node LFAS of KDF are:

Display Table Fields					
Field name	Key	Type	Length	Decima	Short text
LFAS-MANDT	<input checked="" type="checkbox"/>	CLNT C	3		Client
LFAS-LIFNR	<input checked="" type="checkbox"/>	CHAR C	10		Vendor account number
LFAS-LAND1	<input checked="" type="checkbox"/>	CHAR C	3		Country key
LFAS-STCEG	<input type="checkbox"/>	CHAR C	20		VAT registration number

Changing Structures

To change an existing structure, select *Structure* and choose *Change* on the initial screen or choose *Database* → *Display* <-> *Change* on the display screen of the structure.

To change an existing node, position the cursor accordingly and choose *Edit* → *Node* → *Change*.

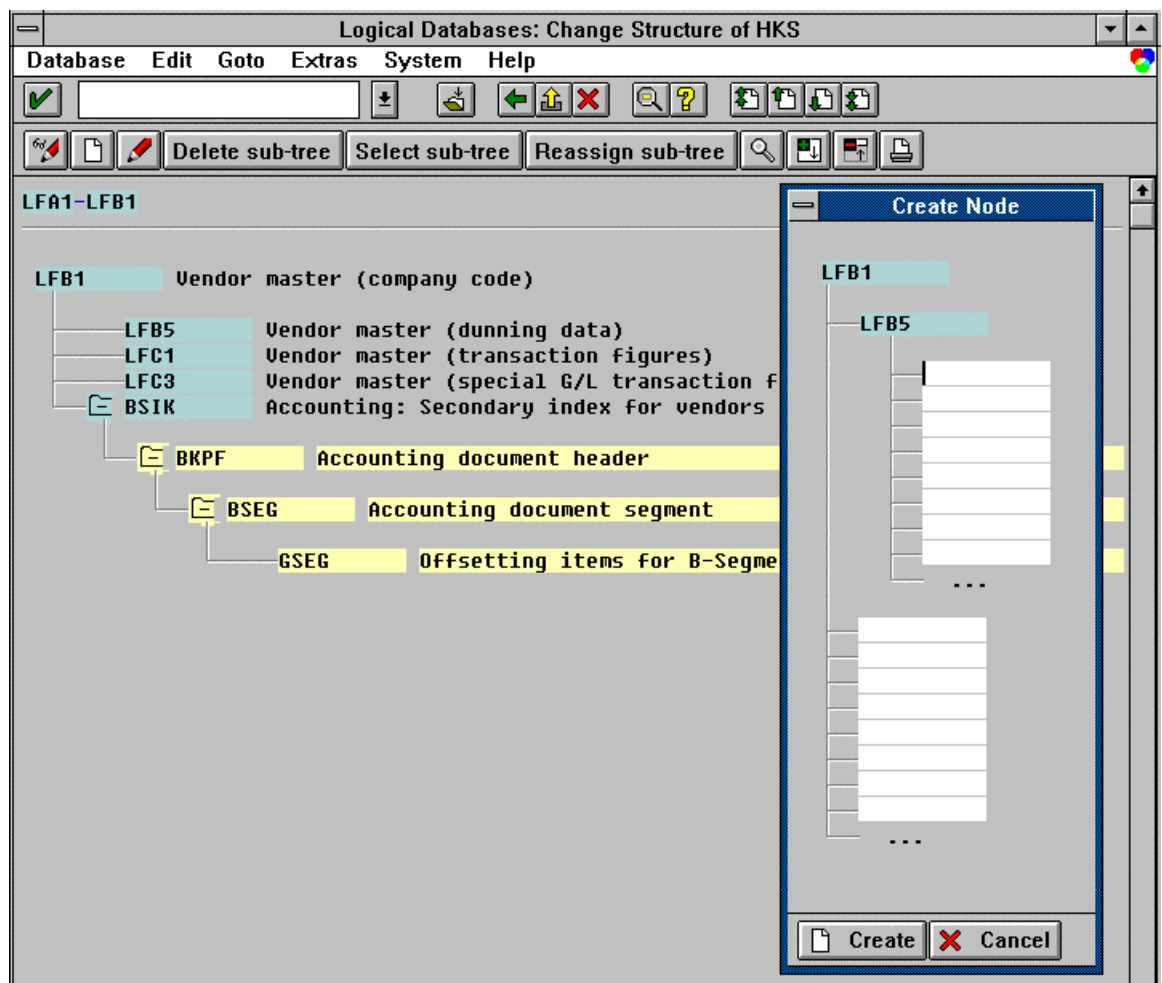
To create a new node at a subordinate level to the cursor position, or on the same level, choose *Edit* → *Node* → *Create*.

To select/deselect a sub-tree, choose *Edit* → *Sub-tree* → *Select/deselect*.

To move a selected sub-tree in a structure to a position indicated by the cursor, choose *Edit* → *Sub-tree* → *Reassign*.

To delete a sub-tree, place the cursor on the node or select it and choose *Edit* → *Sub-tree* → *Delete*.

On the following screen, the user first placed the cursor on the node BKPF and chose *Edit* → *Sub-tree* → *Select*. The next action was to place the cursor on LFB5 and choose *Edit* → *Node* → *Create*.



Changing Structures

As you see, the sub-tree for BKPF is selected. You could now reassign it. Then, you see a dialog box called *Create node* where you can create nodes below LFB5 and at the same level.

Editing Selections

To edit the selection screen of a logical database, select *Selections* and then choose *Change* on the initial screen. This takes you into the editor for the include program DB<dba>SEL. <dba> is the name of the logical database.

You **cannot** use an INCLUDE statement to incorporate this include program in the database program because the system does this automatically for the database program and all the other relevant programs.

If no selections were made before, the system automatically generates the associated SELECT-OPTIONS statements for all database tables in the structure and proposes selection criteria for all key fields (according to the ABAP Dictionary). You must then assign names for these selection criteria. To do this, you enter a name up to 8 characters for each '?' in the include program and delete the comment characters '*' before the statements.

PARAMETERS statements for a matchcode selection are also proposed.

In addition to the proposed selection criteria, you can extend the selection screen with the following elements according to your requirements:

- With the PARAMETERS statement and its additions, you can reference additional parameters which may be used, for example, to control the program flow (see [PARAMETERS - Defining Input Fields for Variables \[Page 803\]](#)).

In the include program DB<dba>SEL, you **must** use the addition FOR TABLE of the PARAMETERS statement. When the selection screen is generated, this ensures that the system only takes into account the parameters connected with tables which are declared in the TABLES statement of the executable program (report).

- You can design the layout of the selection screen with the SELECTION-SCREEN statement and the additions specified under [SELECTION-SCREEN - Formatting \[Page 832\]](#).
- If you write the statement
SELECTION-SCREEN DYNAMIC SELECTIONS FOR TABLE <table>.
you provide for a database table for dynamic selections (see [Standard Selection Screens and Logical Databases \[Page 797\]](#)).
- If you write the statement
SELECTION-SCREEN FIELD SELECTION FOR TABLE <table>.
you provide for a database table for field selection (see [Specifying Fields of the Database Table Explicitly \[Page 1229\]](#)).
- The statements
SELECTION-SCREEN BEGIN|END OF VERSION
and
SELECTION-SCREEN EXCLUDE

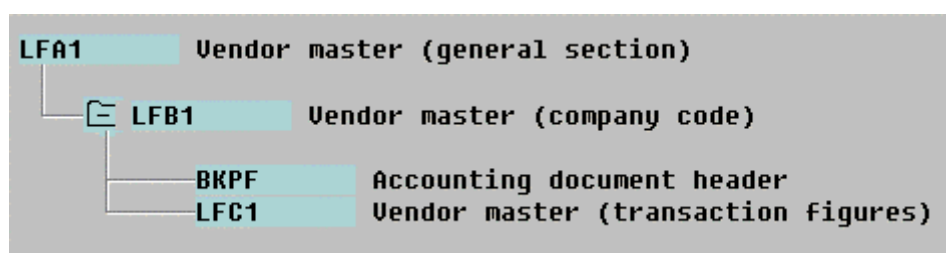
Editing Selections

allow you to create different versions of the selection screen (see the keyword documentation for SELECTION-SCREEN).

If an include program with selections already exists for a logical database, you can overwrite this with the system-defined program by choosing *Extras* → *Generate* → *Selections*. You must then confirm this in a dialog box.

To check the include program DB<dba>SEL for syntax errors, choose *Check* on the initial screen. The syntax of the include program is also checked if you choose *Check* while you are editing the database program.

Suppose the logical database HKS has the following **structure**:



The proposed include program would then be as below:

```

*-----*
* INCLUDE DBHKSSEL
* Automatically included in the database program.
*-----*
*
* If the source code is automatically generated,
* please perform the following steps:
* 1. Replace ? by suitable names (at most 8 characters).
* 2. Activate SELECT-OPTIONS and PARAMETERS (delete
* asterisks).
* 3. Save source code.
* 4. Edit database program
*
* Hint: Syntax check is not possible in this include
* because it is checked during the syntax check of the database
* program.
*-----*
* SELECT-OPTIONS: ? FOR LFA1-LIFNR.
* Parameter for matchcode selection (ABAP Dictionary
* structure MCPARAMS):
* PARAMETERS p_mc AS MATCHCODE STRUCTURE FOR TABLE LFA1.
*
* SELECT-OPTIONS:
* ? FOR LFB1-LIFNR,
* ? FOR LFB1-BUKRS.
*
* SELECT-OPTIONS:
* ? FOR BKPF-BUKRS,
* ? FOR BKPF-BELNR,
* ? FOR BKPF-GJAHR.

```

```
* SELECT-OPTIONS:
*      ?      FOR LFC1-LIFNR,
*      ?      FOR LFC1-BUKRS,
*      ?      FOR LFC1-GJAHR.
```

You could, for example, modify this automatically generated include program in the following way:

* Selection criteria:

```
SELECT-OPTIONS: SLIFNR  FOR LFA1-LIFNR.
SELECT-OPTIONS: SBUKRS  FOR LFB1-BUKRS.
SELECT-OPTIONS: SGJAHR  FOR LFC1-GJAHR.
SELECT-OPTIONS: SBELNR  FOR BKPF-BELNR.
```

* Self-defined parameters:

```
PARAMETERS PDATE LIKE SY-DATUM FOR TABLE BKPF.
```

* Dynamic selections for LFA1 and LFB1:

```
SELECTION-SCREEN DYNAMIC SELECTIONS FOR TABLE: LFA1, LFB1.
```

* Field selection for LFC1:

```
SELECTION-SCREEN FIELD SELECTION FOR TABLE: LFC1.
```

Here, selections are chosen from the available selection criteria and are given names. An additional parameter PDATE is declared and linked to the table BKPF. Dynamic selections are defined for the tables LFA1 and LFB1. Field selection is defined for the table LFC1.

Editing the Database Program

Editing the Database Program

To edit the logical database access program, select *Database program* and choose *Change* on the initial screen. This takes you into the editor for the program SAPDB<dba>. <dba> is the name of the logical database.

If the program does not exist, the system automatically proposes a program generated according to the information in the structure and in the selection include. If you want to overwrite an existing program with the proposed program, choose *Extras* → *Generate* → *Program*. You must then confirm this action in a dialog box.

You cannot change the predefined TABLES statement and the predefined names of the automatically generated subroutines. However, you can define other subroutines or change the ABAP statements for the database accesses.

The WHERE clauses of the automatically generated SELECT statements contain **all the key fields** in the table concerned. For the compare fields, you have the following options:

- If the selection include defines a selection criterion for a field, the field and the relevant selection table are compared with IN, as described under [Using Selection Tables in the WHERE Clause \[Page 858\]](#).

If the selection include contains

```
SELECT-OPTIONS SLIFNR FOR LFA1-LIFNR.
```

the following subroutine automatically appears in the database program:

```
FORM PUT_LFA1.
  SELECT * FROM LFA1
    WHERE LIFNR IN SLIFNR.
  PUT LFA1.
ENDSELECT.
ENDFORM.
```

- If the selection include contains **no** selection criterion for a field, but a key field of a superior table is a foreign key for this field, the fields that are linked by the foreign key dependency are compared.

In the logical database structure, LFB1 is a subordinate node of LFA1 and the key field LIFNR in LFA1 is a foreign key to LFB1.

If the selection include contains

```
SELECT-OPTIONS SBUKRS FOR LFB1-BUKRS.
```

the following subroutine automatically appears in the database program:

```
FORM PUT_LFB1.
  SELECT * FROM LFB1
    WHERE LIFNR = LFA1-LIFNR.
    AND BUKRS IN SBUKRS.
  PUT LFB1.
ENDSELECT.
ENDFORM.
```

Here, a selection criterion SBUKRS is defined for the field BUKRS in LFB1. The WHERE condition BUKRS IN SBUKRS appears as described above. No selection criterion is defined for the field LIFNR, but LIFNR is also a key field in LFA1. Therefore, the WHERE clause is extended by the condition LIFNR = LFA1-LIFNR using AND.

- If you designated database tables for dynamic selections or for field selections in the selection include, you must modify the automatically created SELECT statements accordingly.

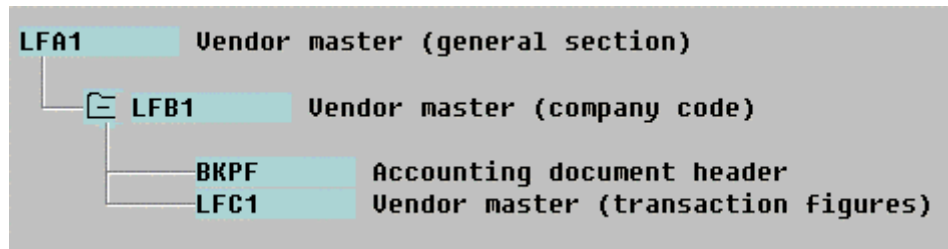
The following topics describe how you can do this:

[Working with Dynamic Selections \[Page 1287\]](#)

[Working with Field Selections \[Page 1291\]](#)

Below is a complete example of an automatically generated ABAP program:

Suppose the logical database HKS has the following **structure**:



Let the include program DBHKSSEL contain the following coded selections:

SELECT-OPTIONS: SLIFNR FOR LFA1-LIFNR.

SELECT-OPTIONS: SBUKRS FOR LFB1-BUKRS.

SELECT-OPTIONS: SGJAHR FOR LFC1-GJAHR.

SELECT-OPTIONS: SBELNR FOR BKPF-BELNR.

The most important lines of the automatically generated database program are listed below. The program also contains some user and performance hints as comment lines, but these are not included here.

```

*-----*
* DATABASE PROGRAM OF LOGICAL DATABASE HKS
*-----*

PROGRAM SAPDBHKS DEFINING DATABASE HKS.

TABLES: LFA1,
        LFB1,
        BKPF,
        LFC1.

*-----*
* BEFORE_EVENT will be called before event EVENT
* Possible values for EVENT: 'START-OF-SELECTION'
*-----*
* FORM BEFORE_EVENT USING EVENT.
* CASE EVENT.
```

Editing the Database Program

```

*   WHEN 'START-OF-SELECTION'
*
*   ENDCASE.
*   ENDFORM.               "BEFORE_EVENT
*
*   -----*
*   AFTER_EVENT will be called after event EVENT
*   Possible values for EVENT: 'END-OF-SELECTION'
*   -----*
*   FORM AFTER_EVENT USING EVENT.
*   CASE EVENT.
*     WHEN 'END-OF-SELECTION'
*
*   ENDCASE.
*   ENDFORM.               "AFTER_EVENT
*
*   -----*
*   Initialize selection screen (processed before PBO)
*   -----*
FORM INIT.

ENDFORM.                   "INIT.
*
*   -----*
*   PBO of selection screen (always processed after ENTER)
*   -----*
FORM PBO.

ENDFORM.                   "PBO.
*
*   -----*
*   PAI of selection screen (always processed after ENTER)
*   -----*
FORM PAI USING FNAME MARK.
*   CASE FNAME.
*     WHEN 'SLIFNR ' .
*     WHEN 'SBUKRS ' .
*     WHEN 'SBELNR ' .
*     WHEN 'SGJAHR ' .
*     WHEN '*'.
*   ENDCASE.
*   ENDFORM.               "PAI
*
*   -----*
*   Call event GET LFA1
*   -----*
FORM PUT_LFA1.
*   SELECT * FROM LFA1
*     INTO TABLE ?
*     WHERE LIFNR   IN SLIFNR.
*   PUT LFA1.
*   ENDSELECT.
*   ENDFORM.               "PUT_LFA1
*
*   -----*
*   Call event GET LFB1

```

```

*-----*
FORM PUT_LFB1.
* SELECT * FROM LFB1
*   INTO TABLE ?
*   WHERE LIFNR   = LFA1-LIFNR
*   AND BUKRS    IN SBUKRS.
  PUT LFB1.
* ENDSELECT.
ENDFORM.                "PUT_LFB1

*-----*
* Call event GET BKPF
*-----*
FORM PUT_BKPF.
* SELECT * FROM BKPF
*   INTO TABLE ?
*   WHERE BUKRS   = LFB1-BUKRS
*   AND BELNR    IN SBELNR
*   AND GJAHR    = ?.
  PUT BKPF.
* ENDSELECT.
ENDFORM.                "PUT_BKPF

*-----*
* Call event GET LFC1
*-----*
FORM PUT_LFC1.
* SELECT * FROM LFC1
*   INTO TABLE ?
*   WHERE LIFNR   = LFB1-LIFNR
*   AND BUKRS    = LFB1-BUKRS
*   AND GJAHR    IN SGJAHR.
  PUT LFC1.
* ENDSELECT.
ENDFORM.                "PUT_LFC1

*-----*
* Authority check for table LFA1
*-----*
* FORM AUTHORITYCHECK_LFA1.
*   AUTHORITY-CHECK...
* ENDFORM.                "AUTHORITYCHECK_LFA1

*-----*
* Authority check for table LFB1
*-----*
* FORM AUTHORITYCHECK_LFB1.
*   AUTHORITY-CHECK...
* ENDFORM.                "AUTHORITYCHECK_LFB1

*-----*
* Authority check for table BKPF
*-----*
* FORM AUTHORITYCHECK_BKPF.
*   AUTHORITY-CHECK...
* ENDFORM.                "AUTHORITYCHECK_BKPF

```

Editing the Database Program

```

*-----*
* Authority check for table LFC1
*-----*
* FORM AUTHORITYCHECK_LFC1.
*   AUTHORITY-CHECK...
* ENDFORM.                "AUTHORITYCHECK_LFC1

*-----*
* PUT_HKS_MATCHCODE.
* Processed when matchcode selection is used,
* i.e. user input into PARAMETERS p_mc AS MATCHCODE STRUCTURE.
*-----*
* FORM PUT_HKS_MATCHCODE.
* ENDFORM.                " PUT_HKS_MATCHCODE

```

The subroutines BEFORE_EVENT, AFTER_EVENT and PUT_<dba>_MATCHCODE are described below. For the description of the remaining subroutines, see [Database Program of a Logical Database \[Page 1254\]](#).

You must delete the comment asterisks '*' before the ABAP statements you want to use in addition to the obligatory statements and replace the question marks (?) by valid ABAP syntax components. Then, you save and check the program. The selection include is automatically checked at the same time.

Some tables and other subroutines are always available at runtime and you can use these if you wish:

- The internal table GET_EVENT tells you which nodes of the logical database are used in the executable program (report). It is generated as follows when the executable program (report) is generated:

```

DATA: BEGIN OF GET_EVENTS OCCURS 10,
      NODE(10),
      KIND,
      END OF GET_EVENTS.

```

The table contains the names of all nodes of the logical database (one per line) in the field NODE. The field KIND specifies whether and how the node is used in the executable program (report) (that is in GET or GET-LATE):

- KIND = 'X': Table is addressed in GET and GET LATE.
- KIND = 'G': Table is addressed only in GET.
- KIND = 'L': Table is addressed only in GET LATE.
- KIND = 'P': Table is addressed neither in GET nor in GET LATE. However, a subordinate table is addressed in GET or GET LATE.
- KIND = ' ': Table is addressed neither in GET nor in GET LATE. No subordinate table is addressed either.

- The subroutine BEFORE_EVENT is generated in the logical database program as a comment (see above example). You can modify it and also activate it by deleting the asterisks (*).

BEFORE_EVENT is called before the event specified in the parameter EVENT is processed.

- The subroutine AFTER_EVENT is generated in the logical database program as a comment (see above example). You can modify it and also activate it by deleting the asterisks (*).
AFTER_EVENT is called after the event specified in the parameter EVENT is processed.
- The subroutine PUT_<dba>_MATCHCODE is generated in the logical database program as a comment. You can modify it and also activate it by deleting the asterisks (*). <dba> is the name of the logical database (see above example).

For further information, see [Editing Matchcodeselections \[Page 1297\]](#).

Working with Dynamic Selections

Working with Dynamic Selections

If you write the statement

```
SELECTION-SCREEN DYNAMIC SELECTIONS FOR TABLE <table>.
```

into the selection Include, you designate the database table <table> for dynamic selections.

To process the dynamic selections in the SELECT statements of the subroutine PUT-<table>, you must use the data object DYN_SEL. The system automatically creates DYN_SEL in the logical database program as follows:

```
TYPE-POOLS RSDS.
```

```
DATA DYN_SEL TYPE RSDS_TYPE.
```

You do **not** need to program these statements yourself. The data object DYN_SEL is available in the database program but not in a connected executable program (report).

The type of this data object (RSDS_TYPE) is defined in the type group RSDS as follows (for information on type groups, see [Type Groups \[Page 137\]](#)):

```
TYPE-POOL RSDS.
```

```
* WHERE-clauses -----
```

```
TYPES: RSDS_WHERE_TAB LIKE RSDSWHERE OCCURS 5.
```

```
TYPES: BEGIN OF RSDS_WHERE,
      TABLENAME LIKE RSDSTABS-PRIM_TAB,
      WHERE_TAB TYPE RSDS_WHERE_TAB,
      END OF RSDS_WHERE.
```

```
TYPES: RSDS_TWHERE TYPE RSDS_WHERE OCCURS 5.
```

```
* Expressions Polish notation -----
```

```
TYPES: RSDS_EXPR_TAB LIKE RSDSEXPR OCCURS 10.
```

```
TYPES: BEGIN OF RSDS_EXPR,
      TABLENAME LIKE RSDSTABS-PRIM_TAB,
      EXPR_TAB TYPE RSDS_EXPR_TAB,
      END OF RSDS_EXPR.
```

```
TYPES: RSDS_TEXPR TYPE RSDS_EXPR OCCURS 10.
```

```
* Selections as RANGES-tables -----
```

```
TYPES: RSDS_SELOPT_T LIKE RSDSSELOPT OCCURS 10.
```

```
TYPES: BEGIN OF RSDS_FRANGE,
      FIELDNAME LIKE RSDSTABS-PRIM_FNAME,
      SELOPT_T TYPE RSDS_SELOPT_T,
      END OF RSDS_FRANGE.
```

```
TYPES: RSDS_FRANGE_T TYPE RSDS_FRANGE OCCURS 10.
```

```
TYPES: BEGIN OF RSDS_RANGE,
      TABLENAME LIKE RSDSTABS-PRIM_TAB,
      FRANGE_T TYPE RSDS_FRANGE_T,
      END OF RSDS_RANGE.
```

```
TYPES: RSDS_TRANGE TYPE RSDS_RANGE OCCURS 10.
```

* Definition of RSDS_TYPE

```
TYPES: BEGIN OF RSDS_TYPE,
        CLAUSES TYPE RSDS_TWHERE,
        TEXPR  TYPE RSDS_TEXPR,
        TRANGE TYPE RSDS_TRANGE,
      END OF RSDS_TYPE.
```

RSDS_TYPE is a complex internal table without a header line. It contains the following components:

CLAUSES

CLAUSES contains the dynamic selections entered by the user (or possibly program-internal selection criteria, see [Program-specific Selection Criteria and Logical Databases \[Page 820\]](#)) as internal tables, which you can use directly in dynamic WHERE clauses (see [Specifying Conditions for Line Selection at Runtime \[Page 563\]](#)).

CLAUSES is an internal table that contains another internal table WHERE_TAB as a component. Each line of the CLAUSES-TABLENAME column contains the name of a database table that is designated for dynamic selections. For each of these database tables, the WHERE_TAB tables contain the selection criteria of the dynamic selections. The WHERE_TAB tables have a format that allows you to use them directly in dynamic WHERE clauses.

To use WHERE_TAB in the logical database, you must program the dynamic WHERE clause for each table designated for dynamic selection in the corresponding subroutine PUT_<table>. To do so, you must read the internal WHERE_TAB table that corresponds to <table> from the data object DYN_SEL. The following example shows how you can use a local data object of the subroutine for this purpose:

Suppose the database table SCARR is the root node of the logical database ZHK and that SPFLI is its single branch.

The selection Include DBZHKSEL contains the following lines:

```
SELECT-OPTIONS S_CARRID FOR SCARR-CARRID.
```

```
SELECT-OPTIONS S_CONNID FOR SPFLI-CONNID.
```

```
SELECTION-SCREEN DYNAMIC SELECTIONS FOR TABLE SCARR.
```

The subroutine PUT_SCARR of the database program SAPDBZHK uses the dynamic selection as follows:

```
FORM PUT_SCARR.
```

```
  STATICS: DYNAMIC_SELECTIONS TYPE RSDS_TWHERE,
           FLAG_READ.
```

```
  IF FLAG_READ = SPACE.
```

```
    DYNAMIC_SELECTIONS-TABLENAME = 'SCARR'.
```

```
    READ TABLE DYN_SEL-CLAUSES
```

```
      WITH KEY DYNAMIC_SELECTIONS-TABLENAME
```

```
      INTO DYNAMIC_SELECTIONS.
```

```
    FLAG_READ = 'X'.
```

```
  ENDIF.
```


Working with Dynamic Selections

```

SELECT * FROM SCARR
      WHERE CARRID IN S_CARRID
      AND (DYNAMIC_SELECTIONS-WHERE_TAB).

PUT SCARR.

ENDSELECT.

ENDFORM.

```

The line of the internal table DYN_SEL-CLAUSES that contains the value 'SCARR' in column DYN_SEL-CLAUSES-TABLENAME is read into the local table DYNAMIC_SELECTIONS. The STATICS statements and the FLAG_READ field assure that table DYN_SEL is only read once during each program execution. The nested table DYNAMIC_SELECTIONS-WHERE_TAB is used in the dynamic WHERE clause.

For each executable program (report) that uses the logical database ZHK and contains a TABLES statement for SCARR or SPFLI, the user can enter dynamic selections for the fields of the database table SCARR on the selection screen. Because of the dynamic WHERE clause, the logical database reads only the data specified as dynamic selections (see [also Standard Selection Screens and Logical Databases \[Page 797\]](#)).

TEXPR

TEXPR contains the limits of the dynamic selections in an internal format (Polish notation). You can use this format with function modules FREE_SELECTIONS_INIT and FREE_SELECTIONS_DIALOG in order to work with dynamic selections within a program (for more information, see the documentation of these function modules). It is the same format as the internal storage format of variants.

TRANGE

TRANGE contains the dynamic selections entered by the user (or possibly executable program (report))-internal selection criteria, see [Program-specific Selection Criteria and Logical Databases \[Page 820\]](#) as RANGES tables, which you can use with the IN operator in WHERE clauses or in logical expressions (see [RANGES \[Page 818\]](#)).

TRANGE is an internal table that contains another internal table FRANGE_T as a component. Each line in the column TRANGE-TABLENAME contains the name of a database table that is designated for dynamic selections. For each of these database tables, the FRANGE_T tables contain the selection criteria of the dynamic selections in the format of RANGES tables. FRANGE_T contains a FIELDNAME column that contains the column names of the database table for which the RANGES tables are defined. The other component of FRANGE_T, SELOPT_T, contains the actual RANGES tables.

With TRANGE, you can directly access the selections of individual database columns. Furthermore, it is easier to modify selection criteria stored in the RANGES format than those stored in the WHERE clause format. The following example shows how you can use local data objects of the corresponding subroutine PUT_<table> to work with TRANGE:

Suppose the database table SCARR is the root node of the logical database ZHK and that SPFLI is its single branch.

The selection Include DBZHKSEL contains the following lines:

```
SELECT-OPTIONS S_CARRID FOR SCARR-CARRID.
```

```
SELECT-OPTIONS S_CONNID FOR SPFLI-CONNID.
```

```
SELECTION-SCREEN DYNAMIC SELECTIONS FOR TABLE SCARR.
```

The subroutine PUT_SCARR of the database program SAPDBZHK uses the dynamic selection as follows:

```
FORM PUT_SCARR.
```

```
  STATICS: DYNAMIC_RANGES TYPE RSDS_RANGE,
            DYNAMIC_RANGE1 TYPE RSDS_FRANGE,
            DYNAMIC_RANGE2 TYPE RSDS_FRANGE,
            FLAG_READ.
```

```
  IF FLAG_READ = SPACE.
```

```
    DYNAMIC_RANGES-TABLENAME = 'SCARR'.
    READ TABLE DYN_SEL-TRANGE
      WITH KEY DYNAMIC_RANGES-TABLENAME
      INTO DYNAMIC_RANGES.
```

```
    DYNAMIC_RANGE1-FIELDNAME = 'CARRNAME'.
    READ TABLE DYNAMIC_RANGES-FRANGE_T
      WITH KEY DYNAMIC_RANGE1-FIELDNAME
      INTO DYNAMIC_RANGE1.
```

```
    DYNAMIC_RANGE2-FIELDNAME = 'CURRCODE'.
    READ TABLE DYNAMIC_RANGES-FRANGE_T
      WITH KEY DYNAMIC_RANGE2-FIELDNAME
      INTO DYNAMIC_RANGE2.
```

```
    FLAG_READ = 'X'.
```

```
  ENDIF.
```

```
  SELECT * FROM SCARR
    WHERE CARRID IN S_CARRID
    AND   CARRNAME IN DYNAMIC_RANGE1-SELOPT_T
    AND   CURRCODE IN DYNAMIC_RANGE2-SELOPT_T.
```

```
  PUT SCARR.
```

```
ENDSELECT.
```

```
ENDFORM.
```

The line of the internal table DYN_SEL-TRANGE that contains the value 'SCARR' in column DYN_SEL-CLAUSES-TABLENAME is read into the local table DYNAMIC_RANGES. The nested tables DYNAMIC_RANGES-FRANGE_T are read into the local tables DYNAMIC-RANGE1 and DYNAMIC-RANGE2 according to the contents of DYNAMIC_RANGES-FIELDNAME. The STATICS statements and the FLAG_READ field assure that the tables are read only once during each executable program (report) execution. After the READ statements, the nested tables SELOPT_T of the local tables contain the RANGES tables for the columns CARRNAME and CURRCODE of the database table SCARR.

The tables SELOPT_T are used in the SELECT statement directly as selection tables. Besides CARRNAME, CURRCODE and the primary key, there are no further columns in the database table SCARR. Therefore, this logical database has the same function as the logical database in the above example using the CLAUSES component.

Working with Field Selections

Working with Field Selections

If you write the statement

```
SELECTION-SCREEN FIELD SELECTION FOR TABLE <table>.
```

into the selection Include, you designate the database table <table> for field selection.

This means that you can work with a SELECT list in the SELECT statements of the subroutine PUT-<table> instead of using SELECT *. This can enhance the performance of the logical database significantly (see [Performance Notes \[Page 597\]](#)).

You can define the SELECT list for each database table designated for field selection using the FIELDS option of the event keyword GET <table> in the reports that use the logical database (see [Specifying Fields of the Database Table Explicitly \[Page 1229\]](#)). The system automatically creates a data object SELECT_FIELDS, which the logical database can use to handle the SELECT lists:

```
TYPE-POOLS RSFS.
```

```
DATA SELECT_FIELDS TYPE RSFS_FIELDS.
```

You do **not** need to program these statements yourself. The data object SELECT_FIELDS is available in the database program and also in each connected executable program (report).

The type of this data object (RSDS_FIELDS) is defined in the type group RSFS as follows (for information on type groups, see [Type Groups \[Page 137\]](#)):

```
TYPE-POOL RSFS.
```

* Fields to be selected per table

```
TYPES: BEGIN OF RSFS_TAB_FIELDS,
      TABLENAME LIKE RSDSTABS-PRIM_TAB,
      FIELDS LIKE RSFS_STRUC OCCURS 10,
      END OF RSFS_TAB_FIELDS.
```

* Fields to be selected for all tables

```
TYPES: RSFS_FIELDS TYPE RSFS_TAB_FIELDS OCCURS 10.
```

RSDS_FIELDS is a complex internal table without a header line. It contains the components TABLENAME and FIELDS. Each line of the TABLENAME column contains the name of a database table that is designated for field selection. For each of these database tables, the FIELDS tables contain the SELECT lists defined in the executable program (report). The FIELDS tables have a format that allows you to use them directly in dynamic SELECT lists in SELECT statements.

In order to use SELECT lists in the logical database, you must place the dynamic SELECT list FIELDS in the SELECT statement for each table designated for dynamic selection in the corresponding subroutine PUT-<table> (see [Selecting and Processing Data from Specific Columns \[Page 546\]](#)). To do so, you must read the internal table FIELDS that corresponds to <table> from the data object SELECT_FIELDS. The following example shows how you can use a local data object of the subroutine for this purpose:

Suppose the database table SCARR is the root node of the logical database ZHK and that SPFLI is its single branch.

The selection Include DBZHKSEL contains the following lines:

SELECT-OPTIONS S_CARRID FOR SCARR-CARRID.

SELECT-OPTIONS S_CONNID FOR SPFLI-CONNID.

SELECTION-SCREEN FIELD SELECTION FOR TABLE SPFLI.

The subroutine PUT_SCARR of the database program SAPDBZHK uses the field selections as follows:

FORM PUT_SPFLI.

 STATICS: FIELDLISTS TYPE RSFS_TAB_FIELDS,
 FLAG_READ.

 IF FLAG_READ = SPACE.

 FIELDLISTS-TABlename = 'SPFLI'.

 READ TABLE SELECT_FIELDS WITH KEY FIELDLISTS-TABlename
 INTO FIELDLISTS.

 FLAG_READ = 'X'.

 ENDIF.

 SELECT (FIELDLISTS-FIELDS)
 INTO CORRESPONDING FIELDS OF SPFLI FROM SPFLI
 WHERE CARRID = SCARR-CARRID
 AND CONNID IN S_CONNID.

 PUT SPFLI.

ENDSELECT.

ENDFORM.

The line of the internal table SELECT_FIELDS that contains the value 'SCARR' in column SELECT_FIELD-TABlename is read into the local table FIELDLISTS. The STATICS statements and the FLAG_READ field assure that the table DYN_SEL is read only once during each executable program (report) execution. The nested table FIELDLISTS-FIELDS is used in the dynamic SELECT list.

An executable program (report) that is connected to the logical database HKZ can contain the following lines, for example:

TABLES SPFLI.

GET SPFLI FIELDS CITYFROM CITYTO.

.....

The FIELDS option of the event keyword GET defines which fields besides the primary key the logical database should read from the database table (see also [Specifying Fields of the Database Table Explicitly \[Page 1229\]](#)).

The system fills the table SELECT_FIELDS with the corresponding values. You can control this by adding the following lines to the executable program (report) program, for example:

DATA: ITAB LIKE SELECT_FIELDS,
 ITAB_L LIKE LINE OF ITAB,
 JTAB LIKE ITAB_L-FIELDS,
 JTAB_L LIKE LINE OF JTAB.

START-OF-SELECTION.

Working with Field Selections

```
ITAB = SELECT_FIELDS.  
LOOP AT ITAB INTO ITAB_L.  
  IF ITAB_L-TABLENAME = 'SPFLI'.  
    JTAB = ITAB_L-FIELDS.  
    LOOP AT JTAB INTO JTAB_L.  
      WRITE / JTAB_L.  
    ENDLOOP.  
  ENDIF.  
ENDLOOP.
```

These lines write the SELECT list to the screen as follows:

```
CITYTO  
CITYFROM  
MANDT  
CARRID  
CONNID
```

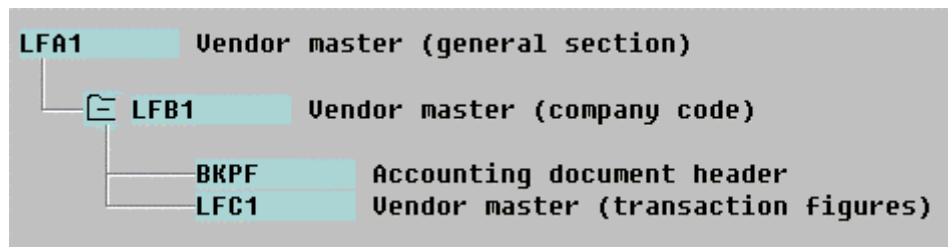
Note that the system automatically added the fields of the primary key (MANDT, CARRID, CONNID) to the self defined SELECT list.

Editing Selection Texts

The selection texts, that is, the texts displayed with the input fields on the selection screen, are normally the names of the selection criteria. You can maintain these texts outside the logical database program (or outside the include program for the selections). This makes your logical database program language-independent. See also: [Working with Text Elements \[Page 146\]](#)

To edit the selection texts in each logon language, select *Selection texts* and choose *Change* on the initial screen. If the logon language is different from the original language (i.e. the logon language under which the logical database was generated), you see a dialog box which prompts you to indicate whether you want to change the text elements in the original language or whether you want to change the original language. By changing the original language, you can maintain the selection texts in any languages.

Suppose the logical database HKS has the following **structure**:



Let the include program DBHKSSEL contain the following coded selections:

SELECT-OPTIONS: SLIFNR FOR LFA1-LIFNR.

SELECT-OPTIONS: SBUKRS FOR LFB1-BUKRS.

SELECT-OPTIONS: SGJAHR FOR LFC1-GJAHR.

SELECT-OPTIONS: SBELNR FOR BKPF-BELNR.

When you select *Selection texts*, the following screen appears:

Name	Text
SBELNR	?... (SBELNR)
SBUKRS	?... (SBUKRS)
SGJAHR	?... (SGJAHR)
SLIFNR	?... (SLIFNR)

You could complete the text fields as follows:

Editing Selection Texts

Name	Text
SBELNR	Document reference number
SBUKRS	Company code
SGJAHR	Fiscal year
SLIFNR	Vendor account number

Assume then the following executable program (report) linked to the logical database HKS:

REPORT SAPMZTST.

TABLES: LFA1, LFB1, LFC1, BKPF.

GET LFA1.

.....

GET LFB1.

.....

GET LFC1.

.....

GET BKPF.

.....



The resulting selection screen would look as follows:

Vendor account number	<input type="text"/>	To	<input type="text"/>	
Company code	<input type="text"/>	To	<input type="text"/>	
Document reference number	<input type="text"/>	To	<input type="text"/>	
Fiscal year	<input type="text"/>	To	<input type="text"/>	

If you then log on to the R/3 System in the logon language "D" for German and call the executable program (report), the names of the selection criteria are displayed on the selection screen. Therefore, go into maintenance of selection texts via the initial screen for logical databases, change the original language to German in the dialog box and enter the following texts:

Name	Text
SBELNR	Belegnummer
SBUKRS	Buchungskreis
SGJAHR	Geschäftsjahr
SLIFNR	Kontonummer, Lieferant

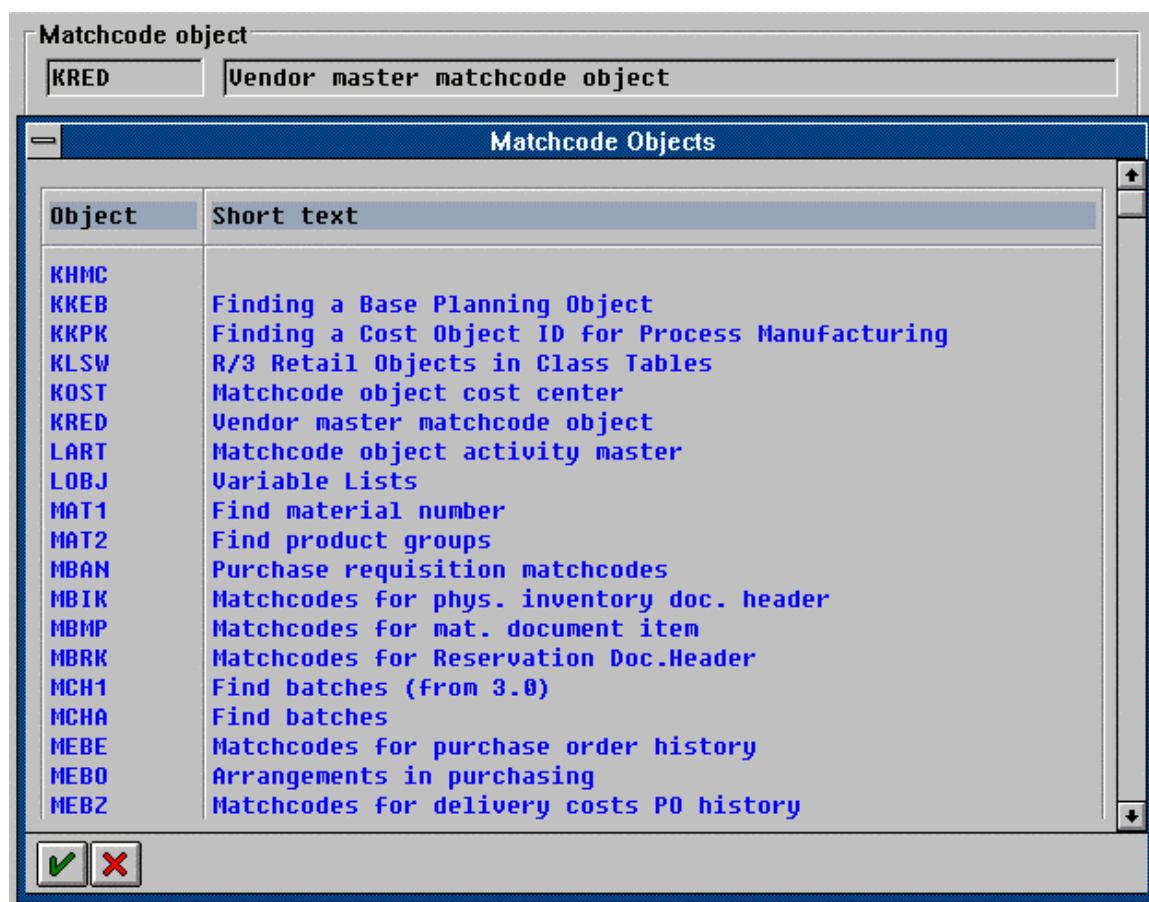
When you log on again in German, the selection screen of the executable program (report) then looks like the one below, but continues to show the English selection texts for English users:

Kontonummer, Lieferant	<input type="text"/>	bis	<input type="text"/>	
Buchungskreis	<input type="text"/>	bis	<input type="text"/>	
Belegnummer	<input type="text"/>	bis	<input type="text"/>	
Geschäftsjahr	<input type="text"/>	bis	<input type="text"/>	

Editing Matchcode Selections

Editing Matchcode Selections

To display, change or create a matchcode selection for the logical database, select *Matchcode selection* on the initial screen. This takes you to the appropriate screen for changing matchcode objects. You can then choose a matchcode object from a list of values (using F4) or delete an existing object:



To find out which matchcode object you need as a matchcode selection for a logical database, you have to know which tables of the logical database structure occur in the matchcode object view. For further information on editing matchcodes, see the online documentation for the [ABAP Dictionary \[Ext.\]](#).

When you have chosen a matchcode object for a logical database, you can offer the user the matchcode selections via a parameter which includes the addition `AS MATCHCODE STRUCTURE` in its declaration in the selection include (see the keyword documentation for `PARAMETERS`). On the selection screen, this results in the display of the group box *Matchcode selection*.

The logical database KDF contains the matchcode selection KRED. The selection include contains the following line:

```
PARAMETERS KD_INDEX AS MATCHCODE STRUCTURE FOR TABLE LFA1.
```

If you call an executable program (report) linked to KDF, the following box is displayed on the selection screen:

The input fields in the group box *Matchcode selection* on the selection screen are *Matchcode ID* (input example "D") and *Search string* (input example "...EDW..").

The system evaluates the user input and chooses the appropriate matchcode records with the key fields. Then, it makes these records available to the database program in the internal table <dba>_MC and calls the subroutine PUT_<dba>_MC. <dba> is the name of the logical database. Using the key from <dba>_MC, this subroutine must choose the data and trigger the event GET <root> via the PUT <root> statement. <root> is the name of the root node. Due to the properties of the PUT statement, you must use an appropriate subroutine PUT_<root>.....for this task.

The structure of the internal table <dba>_MC and other automatically generated tables is displayed as a comment in the automatically generated source code of the database program. The usage of these tables is also documented in the source code.

Suppose the logical database HZS has the root node KNA1.

Let the include program DBHZSSEL contain the lines:

```
SELECT-OPTIONS: SKUNNR FOR KNA1-KUNNR.
```

```
PARAMETERS P_MC AS MATCHCODE STRUCTURE FOR TABLE KNA1.
```

DEBI is specified as the matchcode object.

The source code of the database program now includes more comment lines which indicate that the following tables and fields were created in addition to those listed under [Editing the Database Program \[Page 1281\]](#):

- Internal table HZS_MC:

After START-OF-SELECTION, this table contains the key fields of the matchcode records which match the search string on the selection screen. It has the following structure:

```
DATA: BEGIN OF HZS_MC OCCURS 1000,
      KNA1_MANDT      LIKE KNA1-MANDT,
      KNA1_KUNNR      LIKE KNA1-KUNNR,
END   OF HZS_MC.
```

- Internal table MC_FIELDS:

This table contains fields to which values are assigned during matchcode selection (important only if values are not assigned to all fields during matchcode selection). Its structure is:

Editing Matchcode Selections

```

DATA: BEGIN OF MC_FIELDS OCCURS 10
      INCLUDE STRUCTURE RSMCFIELDS.
DATA: END   OF MC_FIELDS.

```

The field MC_FIELDS-SUPPLIED is not equal to SPACE if a value was assigned to the field in MC_FIELDS-FIELDNAME via the matchcode interface.

- Internal tables MC_TABLES:

Values are assigned to these tables during matchcode selection (important only if the matchcode object contains fields from different tables). Its structure is:

```

DATA: BEGIN OF MC_TABLES OCCURS 10
      INCLUDE STRUCTURE RSMCTABS.
DATA: END   OF MC_TABLES.

```

The field MC_TABLES-SUPPLIED is not equal to SPACE if a value was assigned to the table in MC_FIELDS-TABLENAME via the matchcode interface.

- Field MC_EVENTS:

This is a field with length 200. Each byte in MC_EVENTS stands for a table in the logical database structure (for example, the first character stands for the root node). The contents of the individual positions have the following meaning for the table:

- 'X': Table is addressed in the executable program (report) with the GET statement and values are assigned to key fields via the matchcode.
- 'R': Table is addressed in the executable program (report) with the GET statement, but no values are assigned to key fields via the matchcode.
- 'M': Table is not addressed in the executable program (report) with the GET statement, but values are assigned to key fields via the matchcode.
- ' ': Table is not addressed in the executable program (report) with the GET statement and no values are assigned to key fields via the matchcode.

The inactive subroutine PUT_HZS_MC (see [Editing the Database Program \[Page 1281\]](#)) can, for example, be modified and activated as follows in order to use the matchcode records from the internal table HZS_MC:

```

FORM PUT_HZS_MC.
  IF MC_EVENTS(1) NE SPACE.
    READ TABLE GET_EVENTS WITH KEY 'KNA1'.
    IF SY-SUBRC = 0 AND GET_EVENTS-KIND NE SPACE.
      SELECT * FROM KNA1 FOR ALL ENTRIES IN HZS_MC.
        WHERE KUNNR = HZS_MC-KNA1_KUNNR
      PERFORM PUT_KNA1_MC.
    ENDSELECT.
  ENDIF.
ENDIF.
ENDFORM.

```

```
FORM PUT_KNA1_MC.  
  PUT KNA1.  
ENDFORM.
```

The table GET_EVENTS (see [Editing the Database Program \[Page 1281\]](#)) is used to check whether the linked executable program (report) contains a GET statement for KNA1 or a subordinate node. Depending on the result, a SELECT loop is executed on KNA1 which contains the lines which satisfy the conditions in the table HZS_MC (for more information about this WHERE condition, see [Specifying conditions for Line Selection at Runtime \[Page 563\]](#)). This SELECT loop calls a subroutine PUT_KNA1_MC which executes the PUT KNA1 statement, since the PUT statement can only be used in this way.

You can also use matchcode selections for performance optimization:

The internal tables GET_EVENTS, MC_FIELDS, and MC_TABLES, as well as the field string MC_EVENTS allow you to code different database accesses in the database program, depending on the tables and fields used and filled. For example, it is possible to use views and collect the read records together in internal tables. You can then process these internal tables with LOOP/ENDLOOP and trigger the appropriate GET events.

Editing Documentation

To display or edit the documentation of a logical database, select *Documentation* and choose *Display* or *Change* on the initial screen.

Suppose HKS is a new logical database. When you have created the structure, the selection include, and the database program, you can create the documentation. If you choose *Change* on the initial screen, you see the *SAPscript* Editor. Here, you could proceed as follows:

U1	&DESCRIPTION&
AS	The logical database HKS is for demonstration in ABAP/4 User's Guide only.
U1	&EXAMPLE&
AS

If you then choose *Display*, you see the following documentation for HKS on the initial screen:

RE SAPDBHKS

Description

The logical database HKS is for demonstration in ABAP/4 User's Guide only.

Example

.....

Further Editing Options

The initial screen also offers you the following editing options:

[Editing the Data Model \[Page 1303\]](#)

[Checking Logical Databases \[Page 1305\]](#)

[Copying Logical Databases \[Page 1306\]](#)

[Deleting Logical Databases \[Page 1307\]](#)

Editing the Data Model

Editing the Data Model

To select views and entities belonging to logical database tables, choose *Goto → Data model → Views and entities* on the initial screen. You then see a screen which displays, for each table in the structure, all views which point at least partly to this table.

You choose views by selecting the appropriate checkbox. When doing this, you also automatically choose the entity which is displayed on the same line.

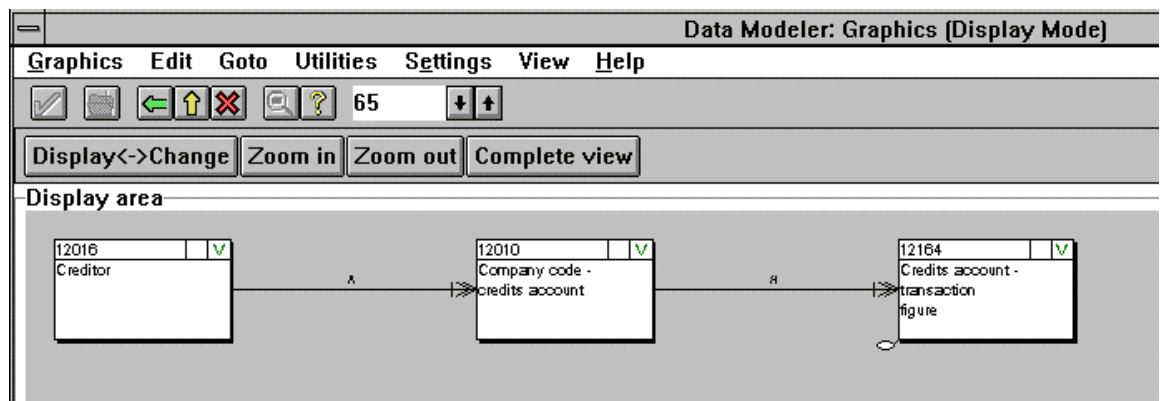
To display a graphic with all dependencies between the selected entities, choose *Graphics*.

Suppose a logical database contains the three nodes LFA1, LFB1 and LFC1.

If you choose *Goto → Data model → Views and entities*, you see the following screen:

Table		
View	Entity	
Vendor master (general section) LFA1		
<input checked="" type="checkbox"/> ENT2016	Creditor	12016
<input type="checkbox"/> ENT2024	Payee	12024
<input type="checkbox"/> ENT2167	Dunning recipient	12167
<input type="checkbox"/> ENT2331	Creditor branch	12331
<input type="checkbox"/> ENT5047	Vendor	15047
Vendor master (company code) LFB1		
<input checked="" type="checkbox"/> ENT2010	Company code - credits account	12010
<input type="checkbox"/> ENT2333	Creditor branch - credits account determination	12333
Vendor master (transaction figures) LFC1		
<input checked="" type="checkbox"/> ENT2164	Credits account - transaction figure	12164
Vendor master record purchasing organization data LFM1		
<input type="checkbox"/> U_15036	Purchasing organization vendor information	15036

If the checkboxes are marked as shown here and you then choose *Graphics*, the SAP network editor appears as follows:



Checking Logical Databases

Checking Logical Databases

To check whether a logical database is correct and complete, choose *Check* on the initial screen.

You then see a screen which displays the following checks:

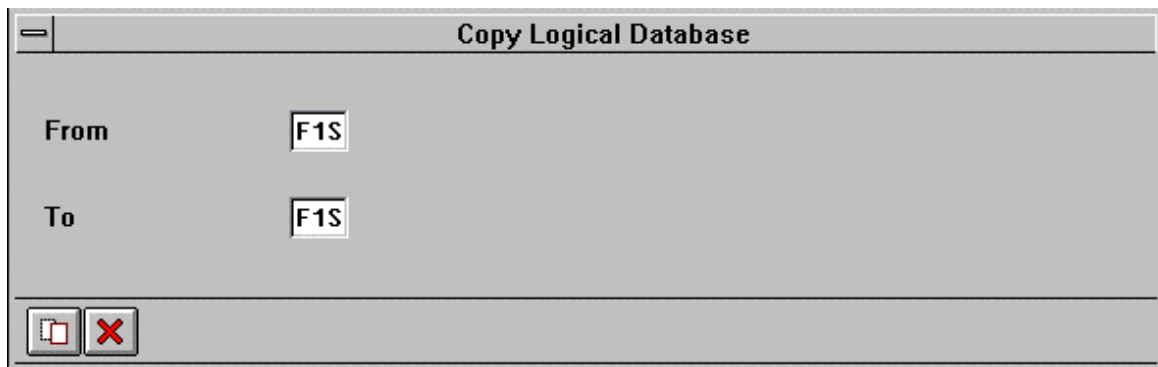
Check	Result
Does logical database exist?	Yes
Does short text exist?	No
Does structure exist?	Yes
Do selections exist?	Yes
Do selection texts exist?	
Does database program exist?	No
Is syntax of database program correct?	
Does documentation exist?	No

You can use this check to determine which sub-objects are available or correct.

Copying Logical Databases

To copy a logical database, choose *Copy* on the initial screen.

Enter the name of the target database in the subsequent dialog box by overwriting the standard value which is always the name of the source database.



The image shows a dialog box titled "Copy Logical Database". It has a standard Windows-style title bar with a minimize button. The main area contains two labels, "From" and "To", each followed by a text input field. Both input fields contain the text "F1S". At the bottom left of the dialog, there are two buttons: a button with a document icon and a button with a red "X" icon.

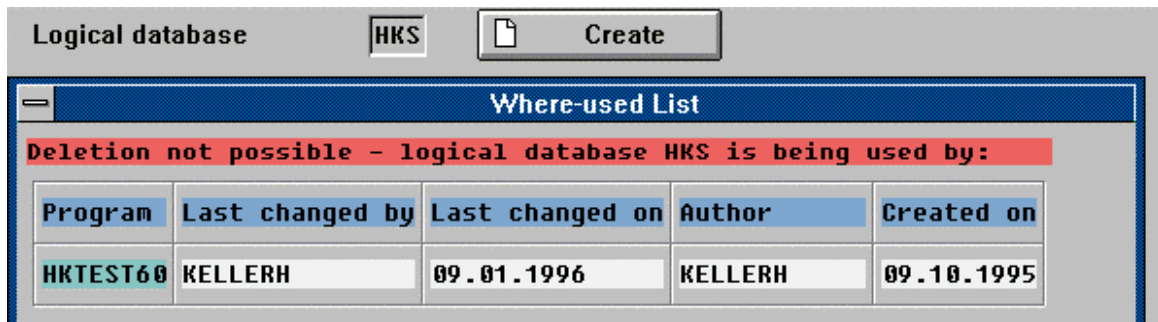
Copy Logical Database	
From	F1S
To	F1S

Deleting Logical Databases

Deleting Logical Databases

To delete a logical database, choose *Delete* on the initial screen.

If the logical database is linked to reports, i.e. specified in the program attributes, you get a message containing the program names concerned. In this case, you cannot delete the database because the affected programs would be rendered incorrect:



Logical database: **HKS**

Where-used List

Deletion not possible - logical database HKS is being used by:

Program	Last changed by	Last changed on	Author	Created on
HKTEST60	KELLERH	09.01.1996	KELLERH	09.10.1995

If the logical database is not used by any executable program (report), you can delete it.

Dialog Mode

[Dialog Mode \[Page 1309\]](#)

Dialog Mode

Dialog Mode

This section gives you an introduction into the ABAP dialog mode. The following topics are described:

[Dialog Mode Programs - Transactions \[Page 1310\]](#)

[A Sample Program \[Page 1313\]](#)

[Screen \[Page 1315\]](#)

[ABAP Module Pool \[Page 1317\]](#)

[Interaction between Dialog Screen and ABAP Module Pool \[Page 1320\]](#)

Dialog Mode Programs - Transactions

Dialog mode programs conduct a dialog with the user. In a typical dialog mode program, the system displays a screen on which the user can enter or request information. As a reaction on the the user input or request, the program executes the appropriate actions: it branches to the next screen, displays an output, or changes the database.

A travel agent wants to book a flight. The agent enters the corresponding data on the screen. The system either confirms the desired request, that is, the agent can book the flight and the customer travels on the desired day on the reserved seat to the chosen destination, or the system displays the information that the flight is already booked up.

To fulfil such requirements, a dialog program must offer:

- a user-friendly user interface
- format and consistency checks for the data entered by the user
- easy correction of input errors
- access to data by storing it in the database.

ABAP offers a variety of tools and language elements to meet the requirements stated above in the dialog programs.

Terminology

Dialog mode programs cannot be executed directly, but only using a transaction code (see [Starting a Program Using a Transaction Code \[Ext.\]](#)). Dialog mode programs are therefore often called **transactions**. However, these are not necessarily transactions in the business sense of the word. For more information about the latter, see [Programming Database Updates \[Ext.\]](#).

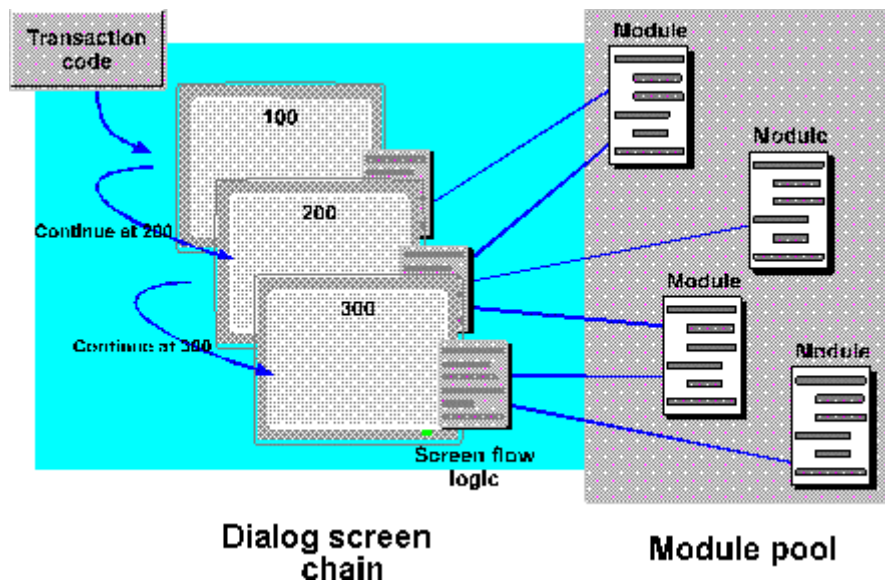
Since dialog mode programs cannot be executed in the background, we often refer to them simply as **dialog programs**.

An ABAP program which can only be run using a transaction code and screens is called a **module pool**. Module pools have program type M.

Components of a Dialog Program

A dialog mode program consists of the following basic components:

Dialog Mode Programs - Transactions



- Transaction code

You can only start a program in dialog mode using a transaction code. Transaction codes are linked to an ABAP program and an initial screen using Transaction SE93 in the ABAP Workbench.

- Screens

Each dialog in an SAP system is controlled by one or more dialog screens. Called dynpros in the original German (DYNAmic PROgram), they consist of a screen and its underlying flow logic, and control exactly one dialog step. The flow logic determines which processing takes place before displaying the screen (PBO-Process Before Output) and after receiving the entries the user made on the screen (PAI-Process After Input).

The screen layout fixed in the Screen Painter determines the positions of input/output fields, text fields, and graphical elements such as radio buttons and checkboxes. In addition, the Menu Painter allows to store menus, icons, pushbuttons, and function keys in one or more GUI statuses. Dynpros and GUI statuses refer to the ABAP program that control the sequence of the dynpros and GUI statuses at runtime.

- ABAP module pool

Each dialog screen refers to exactly one ABAP dialog mode program. Such a dialog program is also called a module pool, since it consists of interactive modules. The flow logic of a dynpro contains calls of modules from the corresponding module pool. Interactive modules called at the PBO event are used to prepare the screen template in accordance to the context, for example by setting field contents or by suppressing fields from the display that are not needed. Interactive modules called at the PAI event are used to check the user input and to trigger appropriate dialog steps, such as the update task.

All dialog screens to be called from within one program refer to a common module pool. The screens in a module pool are numbered. For each screen, the system stores the number of the screen which is normally displayed next.. This screen

sequence or chain can be linear as well as cyclic. From within a screen chain, you can even call another screen chain and, after processing it, return to the original chain.

Transferring Field Data

How do I display fields known in an ABAP module on the screen? How do I transfer user entries on the screen to the module? In contrast to report programming, you **cannot** write field data to the screen using the WRITE statement. The system instead transfers data by comparing screen field names with ABAP variable names. If both names are the same, it transfers screen field values to ABAP program fields and vice versa. This happens immediately before and immediately after displaying the screen.

Field Attributes

For all screen fields of a dialog screen, field attributes are defined in the Screen Painter. If a field name in the screen corresponds to the name of an ABAP Dictionary field, the system automatically establishes a reference between these two fields. Thus, a large number of field attributes in the screen is automatically copied from the ABAP Dictionary. The field attributes together with data element and domain of the assigned Dictionary field form the basis for the standard functions the screen executes in a dialog (automatic format check for screen fields, automatic value range check, online help, and so on).

Error Dialogs

Another task of the screen processor is to conduct error dialogs. Checking the input data is carried out either automatically using check tables of the ABAP Dictionary or by the ABAP program itself. The screen processor includes the error message into the received screen and returns the screen to the user. The message may be context-sensitive, that is, the system replaces placeholders in the message text with current field contents. In addition, only fields whose contents is related to the error and for which a correction may solve the error can accept input. For more information on error handling, see [Handling Errors and Messages \[Page 1172\]](#)

Data Consistency

To keep data consistent within complex applications, ABAP offers techniques for optimizing database updates that operate independent of the underlying database and correspond to the special requests of dialog programming. For more information on database updates, see [Programming Database Updates \[Ext.\]](#)

To illustrate the concept and usage of transactions, a sample transaction follows.

A Sample Program

A Sample Program

Transaction TZ10 (development class SDWA) is delivered with the system. This transaction consists of a single dialog screen. The user can enter the ID of an airline company and a flight number to request flight information:

Display Flight Data

Flight data Edit Goto System Help

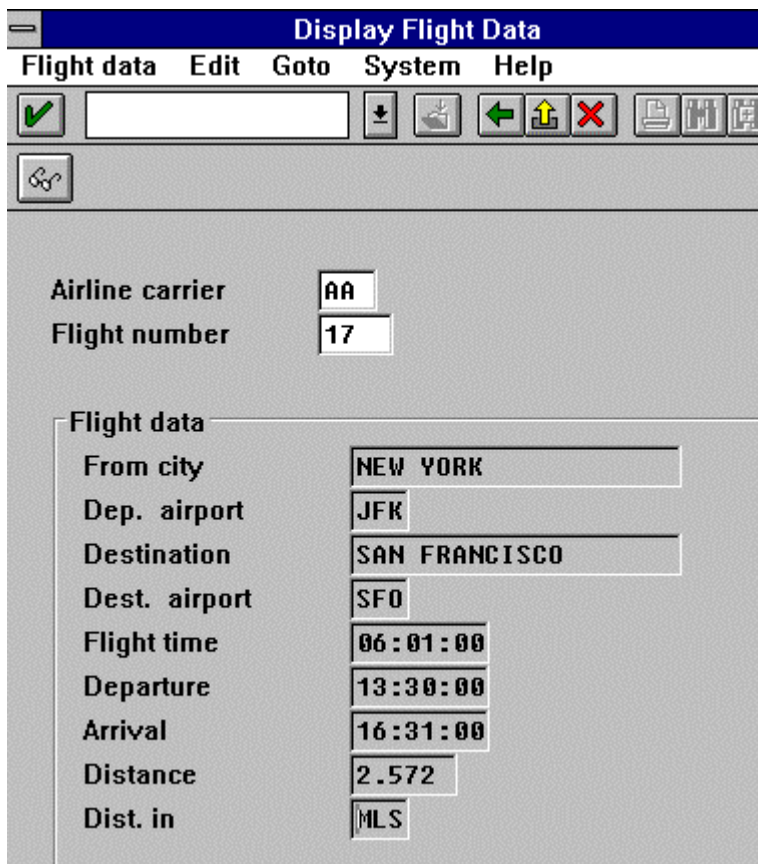
Airline carrier












Flight number

Flight data

From city	<input type="text"/>
Dep. airport	<input type="text"/>
Destination	<input type="text"/>
Dest. airport	<input type="text"/>
Flight time	<input type="text" value="00:00:00"/>
Departure	<input type="text" value="00:00:00"/>
Arrival	<input type="text" value="00:00:00"/>
Distance	<input type="text" value="0"/>
Dist. in	<input type="text"/>

If the user chooses *Display*, the system retrieves the requested data from the database and displays it:



Display Flight Data	
Flight data Edit Goto System Help	
         	
 <input type="text"/>	
Airline carrier <input type="text" value="AA"/>	
Flight number <input type="text" value="17"/>	
Flight data	
From city	<input type="text" value="NEW YORK"/>
Dep. airport	<input type="text" value="JFK"/>
Destination	<input type="text" value="SAN FRANCISCO"/>
Dest. airport	<input type="text" value="SFO"/>
Flight time	<input type="text" value="06:01:00"/>
Departure	<input type="text" value="13:30:00"/>
Arrival	<input type="text" value="16:31:00"/>
Distance	<input type="text" value="2.572"/>
Dist. in	<input type="text" value="MLS"/>

The structure of transaction TZ10 is described in the following topics:

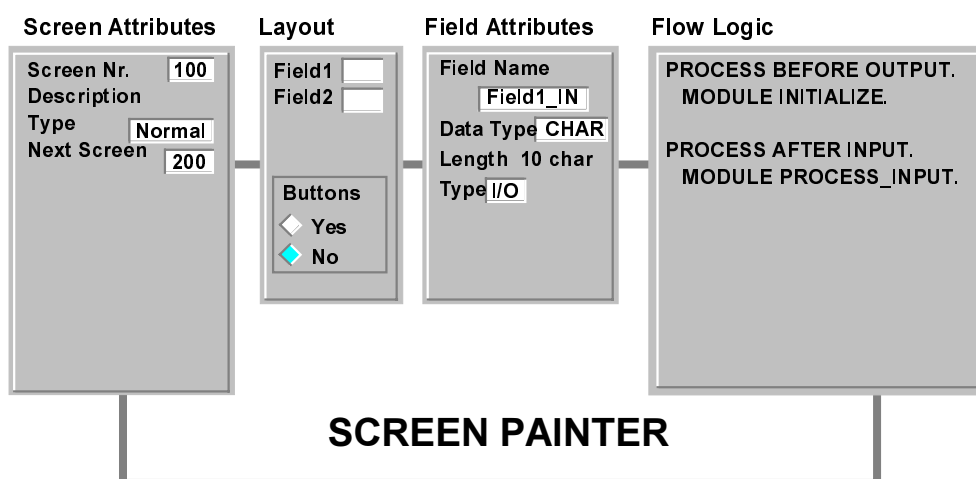
Screen

Screen

Each dialog screen contains fields used to display or request information. Fields can be text strings, input or output fields, radio buttons, checkboxes, or pushbuttons. The screen of Transaction TZ10 contains only texts and input/output fields.

An SAP dialog screen consists of several components:

- **Flow logic:** Calls of the ABAP modules for a screen.
- **Screen layout:** Positions of the texts, fields, pushbuttons, and so on for a screen.
- **Screen attributes:** Number of the screen, number of the subsequent screen, and others.
- **Field attributes:** Definition of the attributes of the individual fields on a screen.



You create and edit all components of a dialog screen in the Screen Painter. To call the Screen Painter, create a screen in the Object Browser or double-click on an existing one. The Object Browser then calls the Screen Painter. There, you can enter the flow logic of the new screen. By pressing the corresponding pushbutton you can maintain the *Screen attributes*, branch to the Fullscreen Editor or you choose the pushbutton *Field list* and change the attributes of fields. For more information on the Screen Painter, see [BC ABAP Workbench Tools \[Ext.\]](#).

Screen Attributes

From the user's point of view, a transaction is a sequence of screens, displayed one after another. How do I determine this sequence? The transactions's attributes determine the first screen to be displayed. The attributes of the individual dynpros determine which screen to display after the current screen. You can also set the number of the subsequent screen dynamically from within the ABAP program.

For our example, the screen attributes need not be changed, since no subsequent screen is called.

Layout

Choose *Fullscreen* to go to the screen editor. Here you can determine the layout of the screen. For Transaction TZ10, the desired fields can be copied from table SPFLI of the ABAP Dictionary. For more information on the fullscreen editor, see [BC ABAP Workbench Tools \[Ext.\]](#).

Field Attributes

To display and modify the attributes of the individual fields (input/output fields, input required, possible entries button, invisible, and so on), use the *Field list*.

The fields *Company* (SPFLI-CARRID) and *Flight number* (SPFLI-CONNID) are defined as input/output fields. All other fields are used only for outputting the flight data.

Flow Logic

The flow control code of a dialog screen consists of a few statements that syntactically resemble ABAP statements. However, you cannot use flow control keywords in ABAP and vice versa. You enter the flow control code in the Screen Painter as one component of the screen.

The flow control for the screen in Transaction TZ10 looks like this:

```
PROCESS BEFORE OUTPUT.  
  MODULE SET_STATUS_0100.  
*  
PROCESS AFTER INPUT  
  MODULE USER_COMMAND_0100.
```

The PROCESS statement names the event type for the screen and the MODULE statement tells the system which ABAP routine to call for this event. In this example, there is only one MODULE for each event PBO and PAI. However, an event can contain several statements with several keywords. (The flow control language contains only few statement types. The most important are MODULE, FIELD, CHAIN, LOOP, CALL SUBSCREEN.)

To display information on the statement syntax in the flow logic, choose *Utilities → Help on...* in the flow logic editor. In the subsequent dialog window, mark *Flow logic keyword*, enter the name of the desired keyword, and press ENTER.

ABAP Module Pool

ABAP Module Pool

In the Object Browser, the module pool code belongs to one of the following categories:

- **Global fields:** data declarations that can be used by all modules in the module pool
- **PBO modules:** modules that are called before displaying the screen
- **PAI modules:** modules that are called in response to the user input
- **Subroutines:** subroutines that can be called from any position within the module pool

By default, the system divides a module pool into one or several include programs. An include program can contain several modules of the same type (only PBO modules or only PAI modules). The main program then consists of a sequence of `INCLUDE` statements that link the modules to the module pool:

```
*&-----*
*& Module pool  SAPMTZ10          *
*&                                     *
*&-----*
*&                                     *
*& Display data of Table SPFLI    *
*&                                     *
*&-----*
```

```
* Global data
INCLUDE MTZ10TOP.
```

```
* PAI modules
INCLUDE MTZ10I01.
```

```
* PBO modules
INCLUDE MTZ10O01.
```

In the ABAP editor, you can display the code hidden behind the `INCLUDE` statements by choosing *Edit* → *More functions* → *EXPAND include*. With all `INCLUDE` statements expanded, the module pool looks like this:

```
*&-----*
*& Module pool  SAPMTZ10          *
*& FUNCTION: Display data from Table SPFLI *
*&                                     *
*&-----*
* INCLUDE MTZ10TOP (This is the TOP include: *
*   the TOP module contains global data declarations) *
*-----*
PROGRAM SAPMTZ10.
TABLES: SPFLI.

DATA OK_CODE(4).

*-----*
* INCLUDE MTZ10I01 (This is a PAI include.) *
```

```

*-----*
*&-----*
*& Module USER_COMMAND_0100 INPUT
*&-----*
* Retrieve data from SPFLI or leave transaction *
*-----*
MODULE USER_COMMAND_0100 INPUT.
CASE OK_CODE.
  WHEN 'SHOW'.
    CLEAR OK_CODE.
    SELECT SINGLE * FROM SPFLI WHERE CARRID = SPFLI-CARRID
      AND CONNID = SPFLI-CONNID.
  WHEN SPACE.
  WHEN OTHERS.
    CLEAR OK_CODE.
    SET SCREEN 0. LEAVE SCREEN.
ENDCASE.
ENDMODULE.

*-----*
* INCLUDE MTZ10001 (This is a PBO include.) *
*-----*
*&-----*
*& Module STATUS_0100
*&-----*
* Specify GUI status and title for screen 100 *
*-----*
MODULE STATUS_0100.
SET PF-STATUS 'TZ0100'.
SET TITLEBAR '100'.
ENDMODULE.

```

You use the ABAP Dictionary to store frequently used data declarations centrally. Objects defined in the Dictionary are known throughout the system. Active Dictionary definitions can be accessed by any application. Data defined in the Dictionary can be included in a screen or used by an ABAP program. You declare global data in the TOP module of the transaction, using the TABLES, STRUCTURE, LIKE statements and others. Transaction TZ10 accesses the Dictionary definition of Table SPFLI to provide the desired flight data display. If the TOP include contains the TABLES: SPFLI declaration, all modules in the module pool can access the table fields of Table SPFLI.

The PAI module USER_COMMAND_0100 checks which pushbutton the user activated (CASE OK_CODE). The *Display* pushbutton in Transaction TZ10 has the function code 'SHOW'. (For more information on handling function codes, see [Processing User Input on a Screen \[Page 701\]](#)). The program then tries to select those records in the SPFLI database that correspond to the data the user entered. The WHERE condition determines matching records by comparing the fields SPFLI-CARRID and SPFLI-CONNID with the database key fields CARRID and CONNID. As soon as a matching record is found, the database transfers all accompanying SPFLI fields to the program table.

When the screen is displayed again, the complete information appears in the output fields of the screen. The system automatically displays these fields, since the ABAP field names SPFLI-CARRID and SPFLI-CONNID are the same as the screen field names.

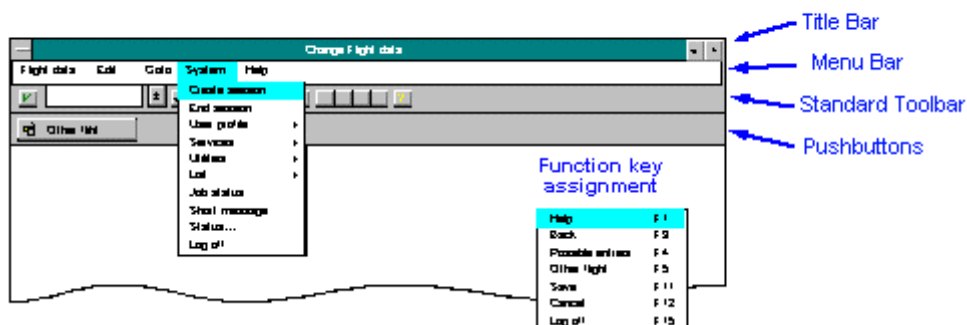
In the PBO module STATUS_0100 of Transaction TZ10, the screen 100 receives a GUI status (using SET PF-STATUS) and a GUI title (using SET TITLEBAR):

ABAP Module Pool

```
SET PF-STATUS 'TZ0100'.
SET TITLEBAR '100'.
```

A *GUI status* is a subset of the interface elements used for a certain screen. The status comprises those elements that are currently needed by the transaction. The GUI status for a transaction may be composed of the following elements:

GUI-Status Elements



The GUI title is the screen title displayed in the title bar of the window. In contrast to the GUI status that can be used for several screens, a GUI title belongs to one screen.

To create and edit GUI status and GUI title, you use the Menu Painter. To start the Menu Painter, create a GUI status or GUI title in an object list in the Object Browser (or double-click on an existing status or title).

For more information on the Menu Painter, see the documentation [BC ABAP Workbench Tools \[Ext.\]](#).

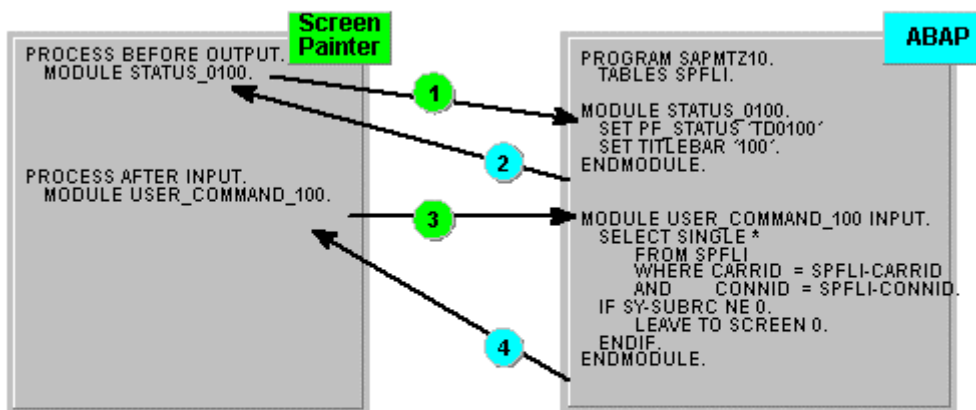
Interaction between Dialog Screen and ABAP Module Pool

In its most simple form, a transaction is a collection of screens and ABAP routines, controlled and executed by a dialog processor. The dialog processor processes screen after screen, thereby triggering the appropriate ABAP processing for each screen.

For each screen, the system executes the flow logic that contains the corresponding ABAP processing. The control passes from screen flow logic to ABAP code and back.

The sequence of events for Transaction TZ10, for example, looks like this:

Interaction between Screens and ABAP Modules



- 1 3 Control switches from screen processing to ABAP processing**
- 2 4 Control switches from ABAP processing to screen processing**

1. In the PBO event, the statement `MODULE STATUS_0100` passes control to the corresponding ABAP module.
In the ABAP module pool, the screen to be displayed receives a menu interface.
2. After processing the module `STATUS_0100`, control returns to the flow logic.
For the PBO event, no further processing is required. The system display the screen and receives entries from the user. The entries are:
 - the values for the fields *Company* and *Flight number*.
 - the four-character function code that tells which pushbutton the user activated.
3. The user input triggers the PAI event. The first PAI statement passes control to the ABAP module `USER_COMMAND_0100`.

Module `USER_COMMAND_0100` processes the requests of the user. In our example, only one request is possible: displaying the flight data for the specified flight. The ABAP statement `SELECT` retrieves the data from the database and displays it.

Interaction between Dialog Screen and ABAP Module Pool

4. After processing MODULE USER_COMMAND_0100, control returns to PAI. This terminates the dialog.

Calling Programs Externally

[Calling External Program Components \[Page 1323\]](#)

Calling External Program Components

Calling External Program Components

In ABAP you have several options for modularizing your transaction. These options include all the ways you can call code components that are external to your program. These external components can be function modules, other transactions, dialog modules or reports.

For information, see:

[Embedding Program Calls \[Page 1324\]](#)

[Calling Function Modules \[Page 1325\]](#)

[Calling Other Transactions \[Page 1330\]](#)

[Calling Dialog Modules \[Page 1331\]](#)

[Using Transactions as Dialog Modules \[Page 1332\]](#)

[Submitting Executable Programs \(Reports\) \[Page 1334\]](#)

[Passing Data Between Programs \[Page 1338\]](#)

Calling external components following user interaction in a list is described in [Calling Programs \[Page 1112\]](#).

Embedding Program Calls

External program components are maintained by the system and are available to all programs. You can call these components in any combination in your transaction.

External programs and the roll area

A roll area contains the program's runtime context. In addition to the runtime stack and other structures, all local variables and any data known to the program are stored here. How does the system handle roll areas for external program components?

- Transactions run in their own roll areas
- Reports run in their own roll areas
- Dialog modules run in their own roll areas
- Function modules run in the roll areas of their callers

When you call external programs that run their own roll areas, you can embed up to 9 levels of call. A called function modules does not add another level.

External programs and LUW processing

At runtime, transactions make database updates that must be run in an all-or-nothing fashion. Either they are all performed, or they are all thrown away. An "LUW" (logical unit of work) is the span of time during which any updates requested belong to an all-or-nothing unit.

A "SAP LUW" refers to the span of time over which an ABAP transaction makes a unitary set of updates. (A SAP-LUW is different from a database LUW, see [Transactions in the SAP System \[Ext.\]](#)).

When you call an external program, it is important to know how updates performed in the called or calling program relate to each other. Does the external program run in the same SAP LUW as the caller, or in a separate one?

- Transactions run with a **separate SAP LUW**
- Executable program (reports) run with a **separate SAP LUW**
- Dialog modules run in the **same SAP LUW** as the caller
- Function modules run in the **same SAP LUW** as the caller

The only exceptions to the above rules are function modules called with IN UPDATE TASK (V2 function only) or IN BACKGROUND TASK (ALE applications). These always run in their own (separate) update transactions.

The fact that an external program shares (or doesn't share) the SAP LUW with its caller has special consequences if the program calls update-task functions or uses COMMIT WORK. For more information, see [Programming Database Updates \[Ext.\]](#).

Calling Function Modules

Calling Function Modules

Function modules are general-purpose library routines that are available system-wide. They can have any number of uses, for example, manipulating strings, making special calculations, making calls to programs on remote systems, or issuing a standard sequence of screens.

Every function module belongs to a *function group*. A function group is a collection of logically related modules that share global data with each other. All the modules in the group are included in the same main program. When an ABAP program contains a CALL FUNCTION statement, the system loads the entire function group in with the program code at runtime.

Accessing the Function Library

The system administers function modules in the Function Library. There you can look up existing functions, their calling interfaces, and documentation, as well as create new functions. To access the Function Library, press *Function Library* in the Workbench.

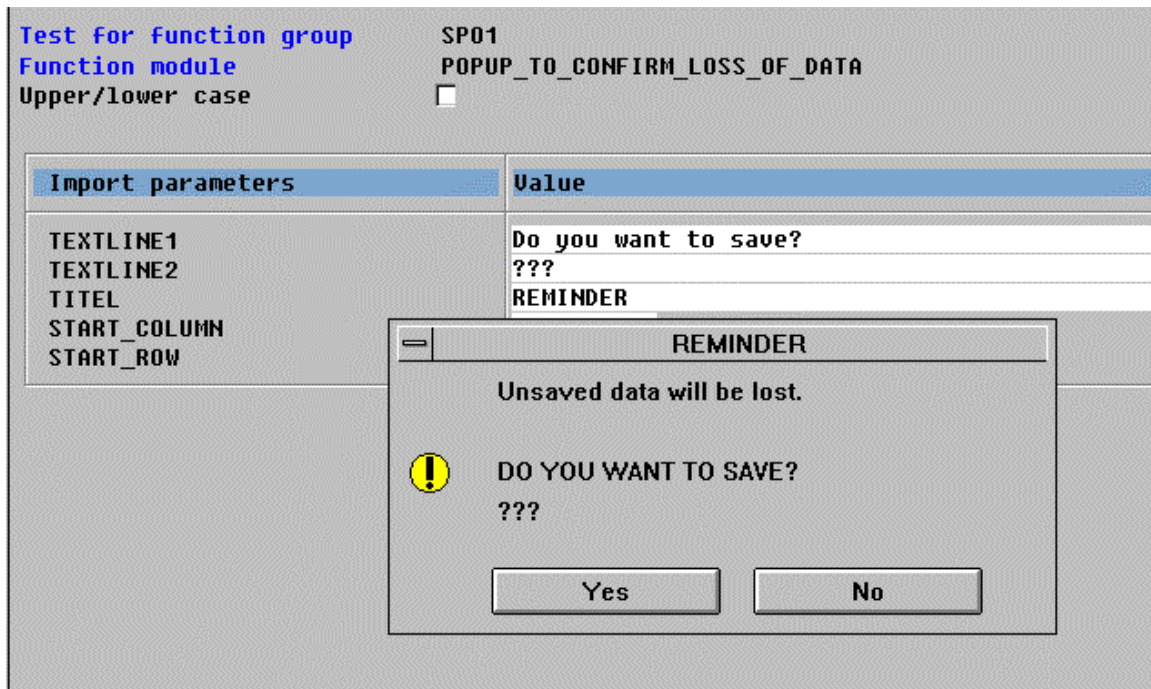
From the ABAP editor, you can double-click on the function module name in your code, or use the *Edit → Insert statement* function. *Insert statement* looks up a function module's interface and inserts a template function call into your program. The inserted call contains pre-formatted parameters.

Making the Call

You call function modules with the CALL FUNCTION statement. For example, suppose you want to prompt your users to save if they exit from a transaction without saving. A function module exists to do this prompting for you:

```
CALL FUNCTION 'POPUP_TO_CONFIRM_LOSS_OF_DATA'
  EXPORTING
    TEXTLINE1   = 'Do you want to save?'
    TEXTLINE2   = '????'
    TITEL       = 'REMINDER'
  IMPORTING
    ANSWER      = REPLY.
```

POPUP_TO_CONFIRM_LOSS_OF_DATA puts out a popup using your TEXTLINE parameters:



On return from the call, the variable `REPLY` contains the user's answer: either yes ('J') or no ('N').

This section explains how to call and write function modules. For information on using the function library tool, see [BC ABAP Workbench Tools \[Ext.\]](#).

Using the Function Module Interface

Your program can only pass data to the function module using the parameters declared in the function module interface. In a `CALL FUNCTION` statement, parameter assignments always have the form: `<formal parameter> = <actual parameter>` where the formal parameters be the names specified in the interface. The `<actual parameters>` may be variables or constants.

In the call to `POPUP_TO_CONFIRM_LOSS_OF_DATA`, the caller sends the text strings 'Do you want to save?', '???' and 'REMINDER' to the function, using formal parameters `TEXTLINE1`, `TEXTLINE2`, and `TITEL`. The user's answer is placed in the variable `REPLY`.

In general, function module can have four types of parameters:

- **EXPORTING:** for passing data to the called function. The corresponding `<formal parameters>` are specified as import parameters in the function module's interface. You can omit **EXPORTING** parameters if they are marked as *Optional* in the function module interface screen.
- **IMPORTING:** for receiving data returned from the function module. The `<formal parameters>` are specified as export parameters in the function module's interface. You can omit any **IMPORTING** parameters if you don't need them: they are all optional.
- **TABLES:** for passing internal tables only, by reference (that is, by address). You cannot omit **TABLES** parameters unless marked as *Optional* in the function module interface screen.

Calling Function Modules

- **CHANGING**: for passing parameters to and from the function (changes to the caller's version are possible). You can omit CHANGING parameters if they are marked as *Optional* in the function module interface screen.

You can also use the *Edit → Insert statement* function (in the ABAP editor) to tell which parameters optional. When the function inserts a CALL FUNCTION statement, the optional parameters are commented out.

For more information on handling exceptions in a function modules, see:

[Handling Exceptions \[Page 1328\]](#)

Handling Exceptions

Function modules let the programmer decide whether the calling program will handle exceptions, or leave exception-handling to the system. To tell the system that the calling program should handle the exceptions, specify EXCEPTIONS in the CALL FUNCTION statement:

```
CALL FUNCTION 'CONVERT_TO_FOREIGN_CURRENCY'
  EXPORTING
    DATE          = TRANS_DATE
    FOREIGN_CURRENCY = FCURRKEY
    LOCAL_AMOUNT   = AMOUNT
    LOCAL_CURRENCY  = LCURRKEY
  IMPORTING
    EXCHANGE_RATE  = RATE_USED
    FOREIGN_AMOUNT  = CONVD_AMT
    FOREIGN_FACTOR  = FCURR_FACTOR
  EXCEPTIONS
    NO_RATE_FOUND   = 1
    OVERFLOW        = 2.
```

Exception types are defined in the function module's interface. For each exception type mentioned in the CALL FUNCTION, the system assumes the caller will handle that error type itself. In the above statement, the programmer signals that the program will handle two error-types (NO_RATE_FOUND and OVERFLOW), and the system should handle any others.

To handle all exception types, use:

```
CALL FUNCTION 'CONVERT_TO_FOREIGN_CURRENCY'
  EXPORTING...
  IMPORTING...
  EXCEPTIONS
    NO_RATE_FOUND   = 1
    OVERFLOW        = 2
    NO_FACTORS_FOUND = 3
    OTHERS          = 4.
```

The OTHERS keyword covers all exceptions not listed in the CALL FUNCTION statement. Use OTHERS to include all those exception types you plan to handle all in the same way. For example, if you coded:

```
CALL FUNCTION 'CONVERT_TO_FOREIGN_CURRENCY'
  EXPORTING...
  IMPORTING...
  EXCEPTIONS
    OTHERS          = 4.
```

your program would still be notified of all the same exceptions, but this time all with a single exception code.

When the caller handles the exception

If an exception occurs that the calling program should handle, the function module sets SY-SUBRC to the corresponding number, and returns controls directly to the program. Only parameters called by reference will contain return values. If the exception was raised with MESSAGE..... RAISING, the system passes message information back in the following system fields:

Handling Exceptions

- SY-MSGID (Message-Id)
- SY-MSGTY (Message type)
- SY-MSGNO (Message number)
- SY-MSGV1 to SY-MSGV4 (contents of fields <f1> to <f4>, included in a message).

These fields are helpful for using the MESSAGE statement with special parameters:

```
MESSAGE SY-MSGID TYPE SY-MSGTY NUMBER SY-MSGNO  
        WITH SY-MSGV1 SY-MSGV2 SY-MSGV3 SY-MSGV4.
```

If any of the SY-MSGV1 to SY-MSGV4 fields are empty, the system just disregards them. For more information about issuing messages, see: [Issuing Messages \[Page 1186\]](#).

When the system handles the exception

If you let the system handle the exceptions, one of two things can happen for the user:

- the program terminates
- the system displays a message and continues processing according to the message type.

If you are writing the function module as well as the calling program, see [Programming Function Modules \[Page 494\]](#) for more information about exception handling.

Calling Other Transactions

From within your transaction, you can branch to or call other transactions. *Branching* to a new transaction terminates the original transaction completely: the user cannot return. *Calling* a new transaction lets you return to the original transaction when the called transaction ends. On return, execution resumes with the instruction immediately following the call.

Leaving to a Transaction

To branch to another transaction and end the current one, use the LEAVE TO TRANSACTION statement:

```
LEAVE TO TRANSACTION '<TRAN>'.
```

The system displays the initial screen of the transaction to which you are branching. Once the new transaction starts, the user can not return to the previous transaction context by pressing the *Exit* icon. Any data the user did not save in the previous transaction is lost.

Calling a Transaction

If you want the user to be able to return to the initial transaction context after processing an interim transaction, use the ABAP statement:

```
CALL TRANSACTION '<TRAN>'.
```

In contrast to LEAVE TO TRANSACTION, the CALL TRANSACTION statement causes the system to start a new SAP LUW (see [Transactions in the SAP System \[Ext.\]](#)). This second SAP LUW runs parallel to the SAP LUW for the calling transaction. By using the keyword LEAVE within the called transaction, you can let the user return to the program context for the caller.

When you call a transaction, you can tell the system to suppress the transaction's initial screen and proceed directly to the next screen in the sequence:

```
CALL TRANSACTION '<TRAN>' AND SKIP FIRST SCREEN.
```

The initial screen is processed but not displayed. Suppressing the first screen only makes sense if all required fields in the initial screen already have values in them. If they don't, your program must pass in data values when calling the transaction. For information on parameter-passing techniques, see [Passing Data Between Programs \[Page 1338\]](#).

Calling Transactions that Share the Caller's SAP LUW

Sometimes you need to call an independent transaction, but you want it to run in the same SAP LUW as the caller. One technique for doing this is to convert the existing transaction to a dialog module. To do this, simply create a new dialog module whose main program and initial screen are the same as those for the existing transaction. Then call the new dialog module with CALL DIALOG.

Transactions that are used both as transactions and as dialog modules must be programmed to obey certain rules. For more information, see [Using Transactions as Dialog Modules \[Page 1332\]](#).

Calling Dialog Modules

Calling Dialog Modules

A dialog module is a callable sequence of screens that does not belong to a particular transaction. Dialog modules have their own module pools, and can be called by any transaction. A dialog module runs within the same SAP LUW (see [Transactions in the SAP System \[Ext.\]](#)) as the calling transaction. For this reason, the system ignores any update routines that a dialog module sends to the update task. Dialog modules are therefore of particular interest when you are programming an application that uses asynchronous updates. Because dialog modules run in their own roll area (function modules share the caller's roll area), dialog modules run more slowly than function modules.

The system administers dialog modules with a tool similar to the Function Library. You can use this tool to look up existing dialogs, their calling interfaces and documentation, as well as to create new dialogs.

To reach the dialog module tool in the Workbench, use the menu option *Development → Programming environment → Dialog modules*. In the Object Browser, use *Environment → Program development → Dialog modules*.

Dialog Module Execution at Runtime

Call dialog modules using the CALL DIALOG statement. For example:

```
CALL DIALOG 'SWO_OBJTYPE_GENERATE'
  EXPORTING
    DIALOG_MODE          FROM MODE
    DIALOG_OBJTYPE       FROM OBJTYPE
  IMPORTING
    DIALOG_RETURN TO RESULT.
```

Using Transactions as Dialog Modules

Transactions used as dialog modules must be programmed to run as either. The following sections describe the two areas in which transaction code must be adapted for double use.

Programming a Double-Use SAP LUW

The transaction must be able to run both in its own SAP LUW (see [Transactions in the SAP System \[Ext.\]](#)), and in the SAP LUW of the caller.

When a dialog module runs, (in the SAP LUW of its caller), special conditions hold:

- Locks created with ENQUEUE are inherited from the caller
When the transaction runs as a dialog module, it may assume that locks for given objects already exist. When the transaction runs as a transaction, it must enqueue its own locks. You can use the system variable SY-CALLD to determine at runtime whether the program is running in call mode or not.
For more information about locking, see [Locking in the SAP System \[Ext.\]](#).
- Update-task keys are inherited from the caller
For information on update-task processing, see [Bundled Updating in the Update Task \[Ext.\]](#).
- COMMIT WORK statements are ignored
You can include COMMIT WORK in a double-use transaction, and the statement will be ignored when the program runs as a dialog module. All updates requested in the dialog module will be processed at the next COMMIT WORK in the caller. For more information, see [COMMIT WORK Processing \[Ext.\]](#).
- Function modules called IN UPDATE TASK are not triggered in the dialog module.
You must ensure that any function modules called with ON UPDATE TASK can be delayed until the next COMMIT WORK in the calling program.
- PERFORM ON COMMIT routines do not execute in the dialog module.
You must ensure that any FORM routines called with PERFORM ON COMMIT can be delayed until the next COMMIT WORK in the calling program. In particular, data local to the dialog module's roll area disappears when control returns to the calling program. No FORM routines called in this way should rely on this local data. For more information, see [Dialog Modules that Call Update-Task Functions \[Page 696\]](#).

You usually need the above features when you program a Save function (F12) or at other spots where you let the user confirm his actions. In general, double-use transactions must be coded in such a way that all updates requested for commit processing can take place either in the called program or in the caller without loss of correctness.

Programming Double-Use Exits

Exits from a double-use transaction must be adapted to return from both call-mode and leave-mode. Execution is in call-mode if the program was invoked by CALL TRANSACTION, CALL DIALOG, or SUBMIT REPORT (and others). In this case, the system variable SY_CALLD is set to 'X'. Execution is in leave-mode if no embedded call is currently active.

The following LEAVE statements are used in one or the other:

Using Transactions as Dialog Modules

- The LEAVE statement causes a return from call-mode.
- The LEAVE TO SCREEN NNN causes a return from leave-mode.

Since double-use transactions could be in either call- or leave-mode, you should use both statements at once, in the following order:

```
AT USER-COMMAND.  
....  
  LEAVE.  
  LEAVE TO SCREEN NNN.
```

If currently running as a transaction, the first LEAVE is ignored, and the LEAVE TO SCREEN NNN is executed. If running as a dialog module, the first LEAVE takes effect, and the second LEAVE is never reached.

Submitting Executable Programs (Reports)

You can start an executable program (report) from a transaction using the SUBMIT statement.

You can code list processing directly in your module pool and output the list using LEAVE TO LIST-PROCESSING instead of calling an external program using SUBMIT. This is particularly appropriate if the transaction has already collected most of the data which you want to appear in the list. For more information, see [Switching Between Dialog Screens and List Display \[Page 1162\]](#).

Use the SUBMIT statement to start a separate executable program (report) from the transaction. This program runs in its own roll area and shares no common data areas with the calling program. For this reason, SUBMIT is preferable when the transaction and the report use little common data. To use SUBMIT, the syntax is:

```
SUBMIT RSBBB013.
```

To execute this statement, the system leaves the current program and starts up the program called.

Returning to the Calling Program: To allow the user to return to the calling transaction, use the keywords AND RETURN.

```
SUBMIT RSFLFIND AND RETURN.
```

In this case, the system opens an internal session for the called program. When the user returns from the list display, the system returns to the transaction screen from which the called program was started. (Execution continues with the instruction directly after the SUBMIT.)

Displaying the Selection Screen:

By default, when you use a simple SUBMIT statement, the standard selection screen of the called program does not appear. To make the program display its standard selection screen, use the VIA SELECTION-SCREEN keywords:

```
SUBMIT RSFLFIND VIA SELECTION-SCREEN.
```

The system displays the selection screen, and the user can specify his own selection criteria. After return from the list display, the selection screen appears again (to let the user request another list). For more information on using SUBMIT, see:

[Passing Data to the Called Program \[Page 1335\]](#)

[Saving or Printing the List \[Page 1337\]](#)

Passing Data to the Called Program

Passing Data to the Called Program

There are three options for passing selection and parameter data to the called executable program. They are:

- using SUBMIT...WITH
- using a variant
- using a RANGE table

These options are described here.

Using the WITH Keyword

You can specify all values needed for parameters or select options using the WITH keyword:

SUBMIT RSBBB013 WITH CARRID = SPFLI-CARRID.

In the above statement, *carrid* is a select option (airline) for program RSBBB013. The list displays only those records relevant to the requested airline.

You can find the names of the corresponding selection-screen fields by:

- using the *Edit* → *Insert statement* function (in the ABAP editor) to add a SUBMIT statement to your code
- requesting field help with the F1 key (in the selection screen)
- using *Utilities* → *Help on* to request logical-database information

Specifying Ranges of Values

You can specify ranges of values with WITH. For example:

SUBMIT RSFLFIND WITH DATE BETWEEN '19950301' AND '19951003'.

Here, *date* is a select option for RSFLFIND that causes the program to display all records with dates between the specified boundaries. There are several legal formats for the WITH specification:

```
WITH <P> <OPERATION> <Q> SIGN <S>
WITH <P> BETWEEN <F1> AND <F2> SIGN <S>
WITH <P> NOT BETWEEN <F1> AND <F2> SIGN <S>

WITH <P> IN <RANGES-TABLE>
WITH SELECTION-TABLE <SELTAB>
WITH FREE SELECTIONS <EXPRESSION-TABLE>
```

The <operation> can be any of EQ, NE, CP, NP, GE, LT, LE, GT. The sign <S> is optional. If used, <S> must be 'I' (including) or 'E' (excluding). All operands <Q>, <F1>, and <F2> are transferred to the program in internal (not display) format.

When <P> is a select option, the system creates entries in the relevant select-options table, setting the LOW, HIGH, OPTION, and SIGN fields as specified. If <P> is a parameter, then all values for <operation> are interpreted to be EQ.

For details on using WITH, see the keyword documentation for SUBMIT.

For information on using RANGES tables, see [RANGES \[Page 818\]](#).

For more information on select-options tables, see [Selection Tables \[Page 816\]](#).

Using Variants

You can use variants to specify parameter and select option values. To do this, use the USING SELECTION-SET keywords:

```
SUBMIT RSFLFIND USING SELECTION-SET VARIANT1.
```

For more information on variants, see [Pre-Setting Selections Using Variants \[Page 864\]](#).

Using RANGES to Specify Select Options

You can specify select option values by filling a RANGES table. For example:

```
TABLES SPFLI.  
RANGES S_CARRID FOR SPFLI-CARRID.  
  
S_CARRID-SIGN  = 'I'.  
S_CARRID-OPTION = 'EQ'.  
S_CARRID-LOW   = 'LH'.  
  
APPEND S_CARRID.  
  
SUBMIT RSFLFIND WITH CARRID IN S_CARRID.
```

In this example, S_CARRID is an internal table with the same structure as a selection table. By referring to the column CARRID (database table SPFLI), the fields S_CARRID-LOW and S_CARRID-HIGH get the same data type as CARRID. The header line of the internal table S_CARRID is filled and appended to the table. The selection condition defined in this table functions like the logical expression SPFLI-CARRID EQ 'LH'.

For information on using a RANGES table, see [RANGES \[Page 818\]](#).

Saving or Printing the List

Saving or Printing the List

You need not display the list to the user. You can also have it sent to the printer, or save it to a storage device somewhere.

Sending the List to the Printer

You can send the list to the printer instead of displaying it on the screen. To do this, use the keywords TO SAP-SPOOL:

```
SUBMIT RSFLFIND... TO SAP-SPOOL DESTINATION 'LT50'.
```

When you use this feature, you have several options for specifying print-request parameters. See the SUBMIT online documentation (*Tools → Help on...* in the ABAP editor) for more information.

Saving the List in Memory

You can use SUBMIT to generate a list and save it in ABAP memory, instead of displaying it on the screen. To do this, use the keywords EXPORTING LIST TO MEMORY:

```
SUBMIT RSFLFIND... AND RETURN  
EXPORTING LIST TO MEMORY.
```

This feature places the generated list in ABAP memory, where the calling program can access it on return from the SUBMIT call. The function group SLST provides function modules for accessing the saved list, including for example:

```
LIST_FROM_MEMORY  
WRITE_LIST  
DISPLAY_LIST
```

Note that the AND RETURN keywords are required with this feature, while the TO SAP-SPOOL keywords are not allowed. See the SUBMIT online documentation (*Tools → Help on...* in the ABAP editor) for more information.

Passing Data Between Programs

Transactions, dialog modules and executable programs (reports) all run in their own roll areas. When you call any of these from your transaction, you must also pass them the data they need to run. The options for passing data to an external program are:

- Using SPA/GPA parameters (SAP memory)

This is the most common method for passing data from one external program to another. For information, see: [Passing Data with SPA/GPA Parameters \[Page 1339\]](#).

- Using EXPORT/IMPORT data (ABAP memory)

Any program can store clusters of data fields in ABAP memory using the EXPORT statement. This data is then available globally (using IMPORT), in the program itself, and in any called transactions, reports or other modules. To use EXPORT:

```
EXPORT <OBJECT1> <OBJECT2>... <OBJECTN> TO MEMORY ID <ID-NAME>.
```

The program you are calling then retrieves the data with :

```
IMPORT <OBJECT1> <OBJECT2>... <OBJECTN> FROM MEMORY ID <ID-NAME>.
```

The ID parameter identifies the data cluster uniquely. If you export the same objects again to the same ID, you overwrite the first version of the cluster in memory. If you export a subset of the objects the second time, you still overwrite *all* the objects (not just the subset) in the first version of the group.

Use EXPORT/IMPORT to implement parameter passing only when the calling and called programs are always used together. EXPORT/IMPORT is not recommended for calling programs that will be available to outside applications, since these applications will have no way of knowing the interface required for calling.

For information about EXPORTing and IMPORTing data clusters, see [Data Clusters in ABAP Memory \[Page 363\]](#).

Passing Data with SPA/GPA Parameters

Passing Data with SPA/GPA Parameters

You can pass data to a called program using SPA/GPA parameters. SPA/GPA parameters are field values saved globally in memory. Each parameter is identified by a three-character code: you can define these parameters in the object browser by selecting *Other objects* on the first screen. The SPA/GPA storage is user-specific and valid throughout all the user's sessions.

There are two ways to use SPA/GPA parameters:

- by setting field attributes in the Screen Painter

The *SET param.*, *GET param.*, and *Param. ID.* attributes tell the system whether to store or retrieve values from the *Param.ID* parameter field. The system uses these values to initialize screen field values automatically.

Mark the *SET param.* attribute for the given fields in the calling screen and the *GET param.* attribute for the corresponding fields in the called screen. The system will automatically transfer the field contents from the calling transaction to the one it triggers.

- by using the SET PARAMETER or GET PARAMETER statements

These statements let you store and retrieve SPA/GPA values from an ABAP program. If the selection screens for the two transactions do not share the same required fields, use these statements to store screen fields explicitly by name.

Before calling the new transaction from a PAI module, store the caller transaction's fields under one name:

```
SET PARAMETER ID 'RID' FIELD <FIELD NAME1>.
```

The system stores the value in <field name1> in the SPA parameter 'RID'. The three-character identifier 'RID' must be defined in the SAP table TPARA. If the SPA parameter 'RID' already contains a value, the SET PARAMETER statement overwrites it (with the contents of <FIELD NAME1>).

In the PBO module for the called transaction, retrieve the fields under the other name:

```
GET PARAMTER ID 'RID' FIELD <FIELD NAME2>.
```

The system reads the contents of 'RID' and transfers them to <FIELD NAME2>.

For example, suppose you want to pass screen fields and other data from a calling transaction to a called transaction. The calling transaction can store certain values in the SPA parameters:

```
SET PARAMETER ID 'RID' FIELD REPORTID.  
CALL TRANSACTION 'SA38'.
```

The called transaction can then fetch the information at PBO in order to display it on the screen. Here, the initial screen for transaction SA38 appears with the program name already filled in. This technique is especially useful when you use CALL TRANSACTION AND SKIP FIRST SCREEN. The first screen cannot be suppressed unless required field values are supplied from memory.

Syntax Conventions

The conventions for syntax statements in this documentation are as follows:

Key	Definition
STATEMENT	Keywords and options of statements are uppercase.
<variable>	Variables, or words that stand for values that you fill in, are in angle brackets. Do not include the angle brackets in the value you use (exception: field symbols).
[]	Square brackets indicate that you can use none, one, or more of the enclosed options. Do not include the brackets in your option.
	A bar between two options indicates that you can use either one or the other of the options.
()	Parentheses are to be typed as part of the command.
,	The comma means you may choose as many of the options shown as you like, separating your choices with commas to be typed as part of the command.
<f ₁ > <f ₂ >	Variables with indices mean that you can list as many variables as you like. They must be separated with the same symbol as the first two.
.....	Dots mean that you can put anything here that is allowed in the context.

In syntax statements, keywords are in upper case, variables are in angle brackets. You can disregard case when you type keywords in your program. WRITE is the same as Write is the same as write.

Output on the output screen is either shown as a screen shot or in the following format:

Screen output.