SDF-P V2.4A

Programming in the Command Language

Edition May 2007

Comments… Suggestions… Corrections…

The User Documentation Department would like to know your opinion on this manual. Your feedback helps us to optimize our documentation to suit your individual needs.

Feel free to send us your comments by e-mail to:
manuals@fujitsu-siemens.com

Certified documentation according to DIN EN ISO 9001:2000

To ensure a consistently high quality standard and user-friendliness, this documentation was created to meet the regulations of a quality management system which complies with the requirements of the standard DIN EN ISO 9001:2000.

cognitas. Gesellschaft für Technik-Dokumentation mbH
www.cognitas.de

Copyright and Trademarks

Contents

1 Preface . . . 17

1.1 Brief product description . . . 17

1.2 Summary of contents . . . 19

1.3 Changes since the last edition of the manual . . . 21
1.3.1 README file . . . 24

2 Brief introduction to SDF-P . . . 25

2.1 Introductory remarks for users familiar with non-S procedures . . . 25

2.2 What is an S procedure? . . . 28

2.3 Rules for writing S procedures . . . 28
2.3.1 Uppercase/lowercase notation . . . 28
2.3.2 Line format . . . 29
2.3.3 Comments . . . 29

2.4 Creating S procedures . . . 30
2.4.1 Format of an S procedure . . . 30
2.4.2 Procedure head . . . 31
2.4.3 Procedure body . . . 32
2.4.4 S variables . . . 32
2.4.5 Expressions . . . 34
2.4.6 Function calls . . . 36
2.4.7 Conditions and loops . . . 37
2.4.8 & replacement . . . 39
2.4.9 Arrays and lists . . . 40
2.4.10 Structures . . . 41
2.4.11 Error handling . . . 43
2.4.12 Programming statement sequences . . . 45
2.4.13 Storing command output in variables . . . 46
2.4.14 S variable streams . . . 47
2.4.15 Using SDF-P interactively . . . 47

2.4.16 Runtime security . . . 48
2.4.17 Testing procedures . . . 48

3 The procedure concept in SDF-P . . . 49

3.1 Structured procedure format . . . 50

3.2 Conventions for S procedures . . . 51
3.2.1 Procedure lines . . . 51
3.2.2 Expression replacement . . . 55
3.2.3 Data and statements . . . 59

3.3 Error handling . . . 69
3.3.1 Error handling blocks . . . 70
3.3.2 Command return codes . . . 71
3.3.3 Error condition when reading in data lines . . . 73
3.3.4 Error messages . . . 73
3.3.5 Error transfer . . . 73

3.4 Procedure compiler . . . 74

3.5 SDF-P commands in interactive mode . . . 76
3.5.1 Rules for entering SDF-P commands interactively . . . 76
3.5.2 Example . . . 79
3.5.2.1 Compiling and linking a program . . . 79

3.6 Calling command sequences from a program . . . 80

4 Creating S procedures . . . 81

4.1 Creating the procedure head . . . 81
4.1.1 Setting procedure attributes . . . 81
4.1.1.1 Defining the procedure call command . . . 82
4.1.1.2 Defining the SYSFILE environment . . . 82
4.1.1.3 Defining the length of procedure lines . . . 83
4.1.1.4 Setting the logging . . . 84
4.1.1.5 Defining the interruptibility of a procedure . . . 84
4.1.1.6 Activation of error handling . . . 84
4.1.1.7 Setting the type of error handling . . . 85
4.1.1.8 Defining the escape character in data records . . . 86
4.1.1.9 Setting implicit declaration of variables . . . 87
4.1.1.10 Setting job variable replacement . . . 87
4.1.1.11 Suppressing specific SDF-P messages . . . 87
4.1.2 Adopting the default values for procedure attributes . . . 88

4.1.3 Declaring the procedure parameters 89
4.1.3.1 Defining the parameter name 89
4.1.3.2 Specifying the data type 90
4.1.3.3 Assigning an initial value 90
4.1.3.4 Specifying the type of parameter transfer 91
4.1.3.5 Initializing procedure parameters with permanent variables 91

4.2 Creating the procedure body 92
4.2.1 Defining single command blocks 92
4.2.2 Defining conditional branches 93
4.2.3 Defining loops 96

4.3 Defining branches 100
4.3.1 The beginning of a block as a jump destination 101
4.3.2 End of block as branch destination 102
4.3.2.1 Branch to end of random command block 102
4.3.2.2 Branch to end of loop 103
4.3.3 Random branch destinations 104

5 Calling and controlling procedures 105

5.1 Calling S procedures in the foreground 105
5.1.1 Choosing the call command 105
5.1.2 Specifying procedure containers 106
5.1.3 Passing procedure parameters 106
5.1.3.1 Passing procedure parameters as positional parameters 107
5.1.3.2 Passing procedure parameters as keyword parameters 108
5.1.3.3 Mixing positional and keyword parameters 109
5.1.3.4 Type of parameter transfer 109
5.1.4 Requesting logging 111
5.1.4.1 Permissibility of logging 112
5.1.4.2 Logging the normal execution of a procedure 112
5.1.4.3 Logging procedure test runs 113
5.1.4.4 Restrictions for procedures with a read password 113
5.1.4.5 Contents of the logging records 113
5.1.5 Unloading programs 115
5.1.6 Setting the execution mode 115
5.1.7 Error transfer 116
5.1.8 Procedure termination 117

5.2 Calling S procedures in the background 118
5.2.1 The ENTER-PROCEDURE call command 118
5.2.2 Specifying procedure containers 119
5.2.3 Passing procedure parameters 119

5.2.4 Requesting logging . . . 119
5.2.5 Job attributes . . . 120
5.2.5.1 Setting the job attributes (including monitoring job variables) . . . 120
5.2.5.2 Setting the startup behavior . . . 120
5.2.5.3 Specifying resource usage . . . 121
5.2.6 Error transfer . . . 121
5.2.7 Terminating procedures . . . 121

5.3 Nesting S procedures . . . 122

5.4 Internal subprocedures . . . 124

5.5 Procedure interruption . . . 125

5.6 Uninterruptibility . . . 126
5.6.1 Implicit procedure protection . . . 127
5.6.2 Protecting programs explicitly . . . 127

5.7 Internal processing of S procedures . . . 131
5.7.1 Analyzing procedures . . . 131
5.7.1.1 Procedure interpreter . . . 131
5.7.1.2 Preanalysis . . . 131
5.7.2 Procedure processing and execution . . . 132

5.8 Compiled procedures . . . 133

6 Using variables in S procedures . . . 135

6.1 Variable concept . . . 135
6.1.1 Basics of the variable concept . . . 135
6.1.1.1 Simple procedure creation . . . 136
6.1.1.2 Runtime security in procedures . . . 137

6.2 Variable declaration . . . 138
6.2.1 Variable types . . . 138
6.2.1.1 Simple variables . . . 139
6.2.1.2 Complex variables . . . 139
6.2.2 Variable names . . . 150
6.2.2.1 Variable name syntax . . . 150
6.2.2.2 Reserved words . . . 154
6.2.2.3 Reserved variable names . . . 154
6.2.3 Data type . . . 156
6.2.4 Initial value . . . 156
6.2.5 Scope of variables . . . 157
6.2.5.1 Scope TASK . . . 158
6.2.5.2 Scopes PROCEDURE and CURRENT . . . 159

6.2.6 Variable containers . . . 162
6.2.6.1 S variables as containers . . . 162
6.2.6.2 Job variables as containers . . . 165
6.2.7 Multiple declaration . . . 166

6.3 Variable processing . . . 167
6.3.1 Assignment of values to variables . . . 167
6.3.1.1 Simple variables . . . 168
6.3.1.2 Complex variables . . . 168
6.3.2 Deleting and removing variables and variable declarations . . . 173
6.3.2.1 Deleting the contents of variables and removing elements . . . 173
6.3.2.2 Deleting variable declarations . . . 175
6.3.3 Input to variables . . . 175
6.3.3.1 Input destination . . . 175
6.3.3.2 Input source . . . 177
6.3.4 Output from variables . . . 178
6.3.4.1 Output source . . . 178
6.3.4.2 Output destination . . . 178
6.3.4.3 Structure layout output . . . 179
6.3.5 Converting SDF command strings to S variables and vice versa . . . 180
6.3.6 S variables and procedure parameters . . . 184
6.3.7 Job variables and S variables . . . 185

7 S variable streams . . . 187

7.1 The concept of S variable streams . . . 187
7.1.1 Functional scope . . . 188
7.1.2 S variable streams SYSINF, SYSMSG and SYSVAR . . . 189
7.1.3 Assigning S variable streams . . . 190
7.1.4 Using S variable streams to transmit S variables . . . 192
7.1.5 Showing S variable stream assignments . . . 193
7.1.6 Deleting S variable streams . . . 194

7.2 Structured output in S variables . . . 195
7.2.1 Method . . . 196
7.2.2 Example . . . 199

7.3 FHS as output server . . . 201
7.3.1 Using FHS as the output server . . . 201
7.3.2 Use and control of FHS applications by S procedures . . . 207
7.3.2.1 Using FHS to output S variables . . . 207
7.3.2.2 Outputting and creating S variables in FHS-TIAM programs . . . 207
7.3.2.3 Controlling FHS applications from nested S procedures . . . 209
7.3.2.4 Application example . . . 211

8 Functions . . . 223

8.1 Function call . . . 223
8.1.1 Input parameters in function calls . . . 225
8.1.2 Functions without input parameters . . . 225
8.1.3 Functions with input parameters . . . 226
8.1.4 Transferring default values . . . 228
8.1.5 Guidelines on specifying values for input parameters . . . 230
8.1.6 Abbreviations of names and keywords . . . 231

8.2 Function result . . . 232

8.3 Function groups with predefined functions . . . 233
8.3.1 String functions . . . 233
8.3.2 Environment information . . . 236
8.3.3 Conversion functions . . . 240
8.3.4 Command return codes / error messages . . . 241

8.4 System administration functions . . . 242
8.4.1 Naming conventions . . . 242
8.4.2 Creating programs . . . 242
8.4.3 Updating BIFTAB and objects . . . 243
8.4.4 Parameter transfer . . . 243
8.4.5 Examples . . . 244

9 Expressions . . . 249

9.1 Base terms . . . 250
9.1.1 Numbers . . . 250
9.1.2 Boolean constants . . . 251
9.1.3 String literals . . . 252
9.1.4 Variable names . . . 254
9.1.5 Function call . . . 255

9.2 Operators . . . 256
9.2.1 Arithmetic operators . . . 256
9.2.1.1 Addition . . . 256
9.2.1.2 Subtraction . . . 257
9.2.1.3 Multiplication . . . 257
9.2.1.4 Division . . . 258
9.2.1.5 Modulo operation . . . 259
9.2.2 Relational operators . . . 260
9.2.3 Logical operators . . . 263
9.2.4 Concatenation operator . . . 264

9.3 Expression types . . . 265

9.4 Evaluation of expressions . . . 266
9.4.1 Operator priority . . . 266
9.4.2 Parentheses . . . 269

10 Optimizing S procedures . . . 271

10.1 SDF syntax analysis . . . 272

10.2 Using variables . . . 273
10.2.1 Grouping commands that use a variable . . . 273
10.2.2 Supplying list variables with values . . . 273
10.2.3 Creating unneeded variables . . . 273

10.3 Using predefined functions . . . 274

10.4 Procedure calls . . . 274
10.4.1 Calling with CALL-PROCEDURE or INCLUDE-PROCEDURE . . . 274
10.4.2 Procedures as library elements . . . 275
10.4.3 Passing parameters or information . . . 275

10.5 Searching for a string in a list . . . 276

10.6 Comments . . . 276

10.7 Program calls in procedures . . . 277
Using the EDT and LMS utility routines in a procedure . . . 277

10.8 Example of an optimized procedure . . . 278

11 Testing S procedures . . . 281

11.1 Tracing procedure execution step-by-step . . . 282

11.2 Modifying logging . . . 283

11.3 Preventing endless loops . . . 283

11.4 Procedure interruption . . . 284

11.5 Simulating the runtime environment . . . 285

12 Converting non-S procedures . . . 287

12.1 Foreground non-S procedures . . . 287
12.2 Enter job . . . 291
12.3 Compatibility of commands for procedure control . . . 293
12.4 Conversion examples . . . 296

13 Program interfaces . . . 305

13.1 Program interfaces for systems support . . . 305
13.1.1 Assembler macros for creating user-written functions . . . 305
BIFDEF . . . 305
BIFDESC . . . 307
BIFMDL1 . . . 309
BIFMDL2 . . . 311
13.1.2 Exit routines . . . 312

13.2 Program interfaces for the user . . . 313
CLIEXPR . . . 314
CLIGET . . . 320
CLISET . . . 323
CMD . . . 326
GETVAR . . . 327
PUTVAR . . . 330
SHOWSSA . . . 333
TRANSVV . . . 336
VARINF . . . 342

14 Predefined functions . . . 347

ACCOUNT( ) Request account number . . . 348
ARRAY-INDEX( ) Request array index . . . 349
BOOLEAN( ) Convert to Boolean value . . . 351
CHARACTER-TO-INTEGER( ) Convert character to integer . . . 352
CHECK-DATA-TYPE( ) Check operand value . . . 354
COUNTER( ) Count function calls . . . 363
CURRENT-TYPE( ) Request variable type . . . 364
DATE( ) Output date . . . 366
DATE-VALUE( ) Output particular date . . . 368
DAY( ) Output day of the week . . . 370
ELAPSED-DAYS( ) Output number of days difference . . . 371

EXPLICIT-CALL( ) Output explicit command call 373
EXTEND-SDF-LIST( ) Append list element 375
EXTRACT-FIELD( ) Extract field 377
FILL( ) Fill string 379
FIRST-VARIABLE-NAME( ) Request variable element name 381
FROM-C-LITERAL( ) Convert C literal 383
FROM-X-LITERAL( ) Convert X literal 385
HASH-STRING( ) Encrypt expression as string 386
HASH-VALUE( ) Encrypt expression as integer value 387
HOME-CAT-ID( ) Request catalog ID of home pubset 389
HOST( ) Request host name 390
INDEX( ) Search for string 391
INSTALLATION-PATH( ) Output path name 396
INTEGER( ) Convert expression to integer 399
INTEGER-TO-CHARACTER( ) Convert integer to character 401
INTEGER-TO-X-LITERAL( ) Convert integer to X literal 402
IS-C-LITERAL( ) Check C literal 404
IS-CATALOGED-FILE( ) Check catalog entry 406
IS-CATALOGED-JV( ) Request job variable 409
IS-DECLARED( ) Check variable declaration 411
IS-EMPTY-FILE( ) Check file size 413
IS-INITIALIZED( ) Check variable initialization 415
IS-INTEGER( ) Check expression 417
IS-LIBRARY( ) Check library name 419
IS-LIBRARY-ELEMENT( ) Check library element 421
IS-SDF-LIST( ) Analyze string against criteria for SDF lists 423
IS-SDF-P( ) Check whether SDF-P is loaded 424
IS-SDF-STRUCTURE( ) Analyze string against criteria for SDF structures 426
IS-VARIABLE-NAME( ) Check variable name 428
IS-X-LITERAL( ) Check X literal 430
JOB-CLASS( ) Request job class 432
JOB-MONJV( ) Request MONJV 433
JOB-NAME( ) Request job name 434
JV( ) Request job variable 436
LAYOUT-SCOPE( ) Request layout scope 438
LENGTH( ) Request string length 440
LIMIT( ) Request maximum list size 442
LOGGING-MODE( ) Check logging 444
LOWER-CASE( ) Convert uppercase letters to lowercase 446
MAINCODE( ) Request error code 448
MONTH( ) Output name of month 450
MSG( ) Output message text 451
NEXT-VARIABLE-NAME( ) Request variable level 453
PROC-LEVEL( ) Request nesting level 455

PROCESSOR( ) Request processor name 457
PROG-MONJV( ) Request MONJV program 458
PROG-NAME( ) Request program name 459
RENAME( ) Generate new name using wildcards 460
REPLACE( ) Overwrite or replace substring 463
RUN-PRIORITY( ) Request runtime priority 466
SDF-P-VERSION( ) Request SDF-P version 467
SDF-STRUCTURE-VALUE( ) Output value of structure 468
SEARCH-LIST-INDEX( ) Search for string in list 471
SESSION-NUMBER( ) Request system sequence number 477
SIZE( ) Request size of complex variables 478
STATION( ) Request TIAM station name 480
STATION-TYPE( ) Request TIAM device type 481
STD-CAT-ID( ) Request catalog ID 482
STMT-SPINOFF( ) Request statement spin-off 483
STRING( ) Convert expression to string 485
SUBCODE1( ) Request subcode1 486
SUBCODE2( ) Request subcode2 488
SUBLIST( ) Select element from SDF list 490
SUBLIST-NUMBER( ) Request number of elements in SDF list 492
SUBSTRING( ) Output substring 493
SYSCMD( ) Request SYSCMD assignment 495
SYSDTA( ) Request SYSDTA assignment 497
SYS-ID( ) Request system identification 499
SYSLST( ) Request SYSLST assignment 500
SYSOUT( ) Request SYSOUT assignment 502
SYSTEM-CALL( ) Output command source 504
SYSTEM-INFORMATION( ) Request system information 506
TASK-MODE( ) Request task mode 510
TIME( ) Request time 511
TO-C-LITERAL( ) Convert string to C literal 513
TO-X-LITERAL( ) Convert string to X literal 514
TRANSLATE( ) Assign result values to input values 515
TRANSLATE-BOOLEAN( ) Check Boolean expression 518
TRIM( ) Remove matching characters at the beginning or end of a string 520
TSN( ) Request TSN 522
UPPER-CASE( ) Convert lowercase letters into uppercase 523
USER-IDENTIFICATION( ) Request user identification 525
USER-SWITCH( ) Evaluate user switch 526
VARIABLE-ATTRIBUTE( ) Request variable attributes 528
VARIABLE-TO-STRING( ) Convert variable 532
VERIFY( ) Verify strings 534
WILDCARD( ) Search for pattern 536
X-LITERAL-TO-INTEGER( ) Convert string to integer 538

15 SDF-P commands . . . 541

15.1 SDF syntax representation . . . 542

15.2 Command return codes . . . 559

15.3 Privileges . . . 562

15.4 Commands . . . 563

ASSIGN-STREAM
Assign S variable stream . . . 563
BEGIN-BLOCK
Initiate command block . . . 568
BEGIN-PARAMETER-DECLARATION
Declare procedure parameters . . . 572
BEGIN-STRUCTURE
Declare static structure . . . 573
CALL-PROCEDURE
Start command sequence . . . 576
CLOSE-VARIABLE-CONTAINER
Close variable container . . . 581
COMPILE-PROCEDURE
Compile procedure . . . 582
CYCLE
Terminate loop pass . . . 587
DECLARE-CONSTANT
Declare variable with constant value . . . 589
DECLARE-ELEMENT
Declare structure element . . . 594
DECLARE-PARAMETER
Declare procedure parameters . . . 600
DECLARE-VARIABLE
Declare variable . . . 607
DELETE-STREAM
Delete S variable stream . . . 616
DELETE-VARIABLE
Delete variable . . . 618
ELSE
Initiate ELSE branch in IF block . . . 620
ELSE-IF
Initiate alternative branch in IF block . . . 621
END-BLOCK
Terminate command block . . . 622
END-FOR
Terminate FOR block . . . 623

END-IF
Terminate IF block 625
END-PARAMETER-DECLARATION
Terminate procedure parameter declaration 627
END-STRUCTURE
Identify end of structure declaration 628
END-WHILE
Terminate WHILE block 629
ENTER-PROCEDURE
Start procedure in background as batch job 631
EXECUTE-CMD
Execute command and structured output 648
EXIT-BLOCK
Terminate processing of command block 653
EXIT-PROCEDURE
Terminate procedure 656
FOR
Initiate FOR block 659
FREE-VARIABLE
Delete contents of variable 664
GOTO
Branch to tag 670
IF
Initiate IF block 672
IF-BLOCK-ERROR
Initiate block error handling 674
IF-CMD-ERROR
Initiate command error handling 676
IMPORT-VARIABLE
Import variable 678
INCLUDE-BLOCK
Executing a BEGIN block as a subprocedure 681
INCLUDE-CMD
Call command sequence from program 683
INCLUDE-PROCEDURE
Start command sequence as include procedure 686
MODIFY-PROCEDURE-OPTIONS
Modify procedure attributes during procedure execution 692
MODIFY-PROCEDURE-TEST-OPTIONS
Modify logging and limit number of back branches 697
OPEN-VARIABLE-CONTAINER
Open variable container 700
RAISE-ERROR
Generate return code 704

READ-VARIABLE
Assign values to variables . . . 705
REPEAT
Initiate REPEAT block . . . 716
REPEAT-CMD
Repeat a command . . . 717
REPEAT-STMT
Repeat a statement . . . 721
SAVE-RETURNCODE
Save current command return code . . . 724
SAVE-VARIABLE-CONTAINER
Save variable container . . . 725
SELECT-VARIABLE-ELEMENTS
Select elements from list variable . . . 727
SEND-DATA
Transfer data record to program . . . 732
SEND-STMT
Transfer statement record to program . . . 733
SET-PROCEDURE-OPTIONS
Set procedure attributes . . . 734
SET-VARIABLE
Assign value to variable . . . 740
SHOW-STREAM-ASSIGNMENT
Show S variable stream . . . 750
SHOW-STRUCTURE-LAYOUT
Output element name of structure layout . . . 754
SHOW-VARIABLE
Output contents of variables . . . 758
SHOW-VARIABLE-ATTRIBUTES
Output variable attributes . . . 769
SHOW-VARIABLE-CONTAINER-ATTR
Display open variable containers . . . 776
SORT-VARIABLE
Sort list variable . . . 779
TRACE-PROCEDURE
Resume interrupted procedure in stages . . . 782
TRANSMIT-BY-STREAM
Transmit variables . . . 783
UNTIL
Terminate REPEAT block . . . 790
WHILE
Initiate WHILE block . . . 791

16 Installation and configuration 793

16.1 Installing SDF-P 793
16.1.1 Installation files for SDF-P 794
16.1.2 Installation files for SDF-P-BASYS 795

16.2 Software configuration 796

17 Messages 797

Glossary 833

Related publications 839

Index 843

1 Preface

1.1 Brief product description

The SDF-P software product is a procedure language that extends the BS2000 command language to a programming language in which structured programming can be performed just as it is using high-level programming languages. With SDF-P, even a beginner can generate short procedures quickly and easily. In addition, the creation and maintenance of large and complex procedures are also greatly simplified. Finally, it is possible, by assigning structured variable streams (referred to briefly as S-variable streams) to store structured outputs in variables, which can then be further processed in numerous ways: e.g. they can be diverted to graphical user interfaces.

Procedures which are created in accordance with the rules of SDF-P are called structured procedures or S procedures. (Procedures which are not created in accordance with the SDF-P rules are called non-S procedures.)

On the command level, SDF-P supports functions of high-level programming languages.

As delivered, SDF-P includes a number of predefined (or built-in) functions with which, amongst other things, variables can be processed and converted or environmental information such as the job status, the processor name or the current date can be determined. These functions can be used in procedures in the same way as the functions of high-level programming languages are used in programs.

In BS2000, SDF-P offers a variable concept of the type that is familiar from high-level programming languages. Thus, it supports not only simple variables, but also complex variables. Variables in SDF-P are also characterized by their data type and their life span or visibility. SDF-P permits the processing of variables both at the command interface and the program interface.

As in high-level programming languages, procedure execution is controlled by loops and branches implemented by means of SDF-P commands. The names selected for these commands are those that are already familiar from various programming languages: WHILE, FOR, REPEAT, IF, ELSE.

SDF-P is a block-oriented programming language, i.e. an essential characteristic of procedures under SDF-P is their structuring in command blocks. These command blocks are not only formed from loops and branches; the programmer can also define any associated procedure parts as command blocks.

The advantages of working with command blocks are, firstly, that the organization of the procedure structure is easily understood and, secondly, error handling is also block-oriented, meaning it can be applied to defined procedure parts.

Product structure

The entire functionality of SDF-P is implemented in two subsystems: SDF-P and SDFPBASY (selectable unit SDF-P-BASYS). While the SDF-P subsystem is chargeable, the SDF-P-BASYS subsystem is included in the BS2000 basic configuration. Procedures offering the SDF-P functionality can be executed subject to the following conditions:

– The SDF-P subsystem must be installed and loaded.
– The SDF-P subsystem is not installed but the procedures are available in a (compiled) intermediate format created on a system where SDF-P was installed.
  A system where SDF-P is not installed permits syntax checks to be performed on S procedures, with the exception of control flow commands and the COMPILE-PROCEDURE command. However, attempts to execute procedures which include chargeable functions will be rejected unless the procedures exist in compiled format.
  The commands and functions of the SDF-P-BASYS subsystem are also described in the manual “Commands, Volume 6” [4].

A description of dependencies between SDF-P and SDF-P-BASYS versions is given in section “Software configuration” on page 796.

Target group

This manual is intended both for BS2000 users who generate procedures as a way of making their everyday work easier and for programmers or system administrators who, for example, perform their system administration tasks with the aid of complex procedures.

Since creating SDF-P procedures is similar to writing procedures in high-level programming languages, the originator of the procedure is generally called a programmer.

1.2 Summary of contents

This manual explains the procedure concept in SDF-P as well as describing all SDF-P interfaces and the creation of S procedures. Although the manual does not deal with the rules applicable to non-S procedures, the behavior of non-S procedures is described where incompatibilities between S and non-S procedures need to be taken into account.

The following is an overview of the contents of the individual chapters:

Chapter 1, “Preface”,
contains a brief description of the BS2000 product SDF-P and some notes on using this manual.

Chapter 2, “Brief introduction to SDF-P”,
introduces users familiar with creating non-S procedures to the creation of S procedures.

Chapter 3, “The procedure concept in SDF-P”,
describes the guidelines which were applied in devising the SDF-P procedure language.

Chapter 4, “Creating S procedures”,
explains how procedures are constructed in SDF-P, and how to write them correctly.

Chapter 5, “Calling and controlling procedures”,
presents the details which the user needs to know in order to be able to call up procedures in the foreground or in the background, or to nest them, etc.

Chapter 6, “Using variables in S procedures”,
deals with the concept of variables, and the rules for declaring and processing variables.

Chapter 7, “S variable streams”,
describes structured output from commands and programs to S variables.

Chapter 8, “Functions”,
describes the use of the system administration functions and predefined functions.

Chapter 9, “Expressions”,
deals with the use of operands, operators etc.

Chapter 10 “Optimizing S procedures”
describes how the performance of S procedures can be optimized during their creation.

Chapter 11, “Testing S procedures”,
describes the aids available to the programmer when looking for errors in S procedures.

Chapter 12, “Converting non-S procedures”,
contains notes on how, among other things, foreground and background non-S procedures can be converted.

Chapter 13, “Program interfaces”,
is a reference chapter dealing with program interfaces.

Chapter 14, “Predefined functions”,
is a reference chapter dealing with predefined functions.

Chapter 15, “SDF-P commands”,
is a reference chapter dealing with the SDF-P commands.

Chapter 16, “Installation and configuration”,
lists the syntax, message and product files required for installing SDF-P and describes dependencies between the versions of the subsystems involved.

Chapter 17, “Messages”,
is a summary of all messages in the SDP message class.

At the back of the manual there is a glossary, a list of related publications and an index.

Conventions used in this manual

The conventions applying to the chapters of the reference part are described in those chapters. The following conventions apply to the introductory part.

Note
The word *“Note”* preceding an indented paragraph indicates that this paragraph contains important information.

[1]
Numbers in square brackets in the text refer to the item of the same number in the list of related publications at the back of the manual.

Bold type
Wherever syntax representations are explained, the lines being discussed are printed in bold type.
The rules described in the relevant chapters of the reference part apply to all syntax representations.

`SYNTAX /Example`
Representations of syntax and sample inputs and outputs are printed in a different type font. In addition, representations of syntax are enclosed in boxes.

`[ ]`
Characters enclosed in square brackets in syntax representations may be omitted.

1.3 Changes since the last edition of the manual

This manual describes the functionality of SDF-P V2.4A. The following changes have been made since the last edition of this manual:

General changes and additions

All chapters of the manual have been revised. All examples have been checked for executability and updated to conform to BS2000/OSD-BC V7.0.

Functional changes and enhancements from SDF-P V2.3A

Functions

Name	Operand	Functionality, Comment
HOST( )		Functional change: can now be called independently of the JV subsystem
SEARCH-LIST-INDEX( )	DIRECTION=	New operand: permits the forward and reverse search
SESSION-NUMBER( )		Functional change: can now be called independently of the JV subsystem

Commands

Name	Operand	Functionality, Comment
FREE-VARIABLE	FROM-INDEX=*LAST	New operand value: specifies the last element
	NUMBER-OF-ELEMENTS= *REST	New operand value: specifies the number of elements from the start value to the last element in the list
IMPORT-VARIABLE	VARIABLE-NAME=<structured-name 1..20 with-wild(40)> / list-poss(2000): <structured-name 1..20>	Operand extended: permits wildcards in the variable name and the specification of multiple variable names in list form

Name	**Operand**	**Functionality, Comment**

READ-VARIABLE	INPUT=<file-name>(...)
	BEGIN-RECORD=
	END-RECORD=
	BEGIN-COLUMN=
	END-COLUMN=
	PATTERN=
	
	PATTERN-TYPE=
		Structure extended by new operands:
		Reads from the start record to the end record
		Reads within the record from the start column to the end column
		Selects records which contain a search string
		Specifies whether the search string is a string or a regular expression

	INPUT=*SYSDTA(...) REMOVE-KEY=	Structure assigned new operand: permits specification for handling the ISAM key if SYSDTA is read from an ISAM file
SHOW-STRUCTURE-LAYOUT	NAME=<structured-name 1..20 with-wild(40)>	Operand extended: permits wildcards in the name of the structure layout
SHOW-VARIABLE	OUTPUT=*LIBRARY(...) WRITE-MODE=	Structure assigned new operand: specifies whether library members are to be extended or overwritten
SORT-VARIABLE		New command: sorted elements of a list variable

Functional changes and enhancements in SDF-P V2.4A

Functions

Name	Operand	Functionality, Comment

INDEX( )	BEGIN-COLUMN=
	
	END-COLUMN=
		New operand: searches from this column position
		New operand: searches up to this column position

INTEGER-TO-X-LITERAL		New function: converts an integer to a 4-byte long X string
X-LITERAL-TO-INTEGER		New function: converts a string up to 4 bytes long to an integer

Commands

Name	Operand	Functionality, Comment
SELECT-VARIABLE	MESSAGE=	New operand: permits a message to be displyed at the start of a menu

SET-VARIABLE	FROM-INDEX=*LAST
	
	NUMBER-OF-ELEMENTS= *REST
		New operand value: specifies the last element in a list
		New operand value: specifies the number of elements from the start value up to the last element in the list

SHOW-VARIABLE	VARIABLE-NAME=*LIST(...)	New operand value: permits a list variable or individual list elements to be displayed
SHOW-VARIABLE-ATTRIBUTES	VARIABLE-NAME=*LIST(...)	New operand value: permits the attributes of a list variable or individual list elements to be displayed

1.3.1 README file

Information on any functional changes and additions to the current product version described in this manual can be found in the product-specific README file. You will find this README file on your BS2000 computer under the name SYSRME.*product.version.language*, and for SDF-P V2.4 under SYSRME.SDF-P.024.E . The user ID under which the README file is cataloged can be obtained from your system administrator. The full path name is also output using the following command:

```
   /SHOW-INSTALLATION-PATH INSTALLATION-UNIT=SDF-P,LOGICAL-ID=SYSRME.D
```

You can view the README file using the /SHOW-FILE command or an editor, and print it out on a standard printer using the following command:

```
/PRINT-DOCUMENT FROM-FILE = <filename>,LINE-SPACING = *BY-EBCDIC-CONTROL
```

2 Brief introduction to SDF-P

2.1 Introductory remarks for users familiar with non-S procedures

This section is intended for readers who are already familiar with non-S procedures; these are procedures that start with either of the commands PROCEDURE or BEGIN-PROCEDURE.

S procedures are essentially an enhancement of the existing procedure format that largely complies with the rules for high-level programming languages. Many non-S procedures can be converted to S procedures without major alterations (see chapter “Converting non-S procedures” on page 287ff). However, to make the most of the intrinsic advantages of the new procedure format, some of the programming methods used most frequently in the “old” procedures must be adapted to permit the utilization of the new constructs offered by SDF-P. The table below shows the resultant enhancements:

Feature	**Non-S procedures**	**S procedures**
Information storage	Procedure parameters (constants), job variables	S variables; job variables
Control flow processing	Conditional and unconditional branches, error handling using spin-off and the SET-JOB-STEP command	Condition blocks, loops, GOTO, error handling using spin-off or return codes and error handling blocks
Word processing	Editor call	String operators, predefined functions
Arithmetics	Editor call, special programs	Integer operators, predefined functions
Access to system output	Assignment of SYSOUT to file with subsequent editing	Structured output in S variables, predefined functions
Procedure integrity checking	Call in SDF test mode	Variable and type declarations, preanalysis of control structures at calling time, call in SDF test mode
Procedure debugging	———	Procedure test mode, tracing of individual steps

The section “Format of an S procedure” on page 30 contains a sample S procedure that outputs the last few lines of a text file to SYSOUT; both the name of the file and the number of lines to be output can be specified. To offer more or less the same functional scope, a non-S procedure would have to make use of certain TU programs, e.g. the EDT text editor. The sample procedure below shows how the functionality might be implemented in a non-S procedure and is intended to illustrate the contrast between non-S and S procedures:

```
/BEG-PROC LOG=*NO,PROC-PAR=(&FILE,&NUMBER=10),ESC-CHAR='&'
/ASS-SYSDTA *SYSCMD
/MOD-JOB-SW ON=(4,5)
/START-EXEC-PROG $EDT
@ SETSW OFF=4-5
@ PRO 1
@ 1.00
@N    Analyze FILE parameter:
@N    either  fully-qual. file name without gen/vers
@N    or      '[*lib-elem](library,element[(vers)][,typ])'
@@CR #S1 = '&FILE'
@@ON #S1 F '('
@@IF .F. GOTO 3
@ 2.00
@N    An element has been specified:
@N    extract library name
@@#I1 = #I0 + 1
@@#I2 = L #S1
@@CR #S1: #S1:#I1-#I2:
@@ON #S1 D R ')'
@@IF .F. GOTO 4
@@ON #S1 F','
@@IF .F. GOTO 5
@@#I3 = #I0 - 1
@@CR #S2: #S1:1-#I3:
@@#I3 = #I3 + 2
@@#I2 = L #S1
@@CR #S1: #S1:#I3-#I2:
@N    Load element
@@PRO 11
@@CR 1: '@COPY L=', #S2, '(', #S1, ')'
@@END
@@DO 11
@@IF NO ERRORS: @GOTO 7
@@GOTO 6
@ 3.00
@N    Load file
@@READ '&FILE'
@@IF NO ERRORS: @GOTO 7
@@GOTO 6
@ 4.00
```

```
@@CR #S1: 'Closing parenthesis after element spec is missing.'
@@P #S1 N
@@GOTO 6
@ 5.00
@@CR #S1: 'Comma between library name and element name is missing.'
@@P #S1 N
@@GOTO 6
@ 6.00
@@CR #S1: 'Error after loading "&FILE"!'
@@P #S1 N
@@GOTO 9
@ 7.00
@N    Now specify line number where
@N    output is to start (not < 1!)
@@#I1 = &NUMBER
@@IF #I1 < 1 GOTO 9
@@#L1 = $
@@#S1 = C #L1
@@#I2 = S #S1:1-4:
@@IF #I2 > #I1 GOTO 8
@N    Output all
@@P & N
@@GOTO 9
@ 8.00
@N    Output range
@@#I1 = #I1-1
@@#L1 = $-#I1
@@P #L1.-.$ N
@ 9.00
@@N
@ END
@ DO 1
@N    Cleanup (otherwise there will be a query upon @HALT)
@ DROP ALL
@ HALT
/SET-JOB-STEP
/MOD-JOB-SW OFF=(4,5)
/END-PROC
```

2.2 What is an S procedure?

In its simplest form, an S procedure is a sequence of ordinary commands that is stored sequentially (i.e. in the order in which the commands are to be executed) in a text file or a library element. Each command must be preceded by a slash. Statements for programs may be inserted at a suitable position between the commands; all program statements must be preceded by two slashes. Data lines (for input from SYSDTA) may also be inserted; these are identified by the absence of a slash in the first column.

An S procedure may additionally contain specific SDF-P commands and special character strings that may be used for specific processing control, to modify commands and to influence the sequence of command execution (i.e. for purposes of procedure programming proper). These are, however, optional. For instance, the following is a complete S procedure:

```
/CREATE-FILE #HELLO, SUPPRESS-ERRORS=*FILE-EXIST
/WRITE-TEXT 'Hello, world!'
```

The procedure can be called repeatedly. This may be done by means of the CALL-PROCEDURE command in which all that has to be specified is the name of the procedure file.

2.3 Rules for writing S procedures

2.3.1 Uppercase/lowercase notation

Uppercase and lowercase letters in commands are not distinguished, with the exception of operand values where the distinction is relevant for command execution. For instance, the uppercase and lowercase letters in the WRITE-TEXT command from the sample procedure given above are output as entered. Within this manual, however, commands are represented using mostly uppercase letters, as this is the traditional notation that is supported by all editors, terminals and output devices.

Where lowercase letters are used in command definitions, they often represent variable parts that have to be replaced by suitable current values (see the description of the metasyntax as of page 544).

2.3.2 Line format

In many cases, each text line will contain a command. The maximum line length is almost 4 KB, which should be sufficient even for complex commands.

However, since a much shorter line length is convenient for screen display and editing (in most cases 72 or 80 characters), long commands are usually split up into several lines. If a command extends over more than one text line, a hyphen must be written as the last character in all lines except the last line. All continuation lines begin with a slash, immediately followed by the continuation of the command. To permit continuation lines to be indented so that the result is easier to read without the runtime log having to contain the same number of blanks as represented by the indentation, the first slash of a continuation line may be preceded by blanks; these are not part of the command.

Using continuation lines and indentation is particularly advisable to improve the readability of nested operand structures such as the following:

```
/ MODIFY-FILE-ATTRIBUTES -
        /FILE-NAME = INFO.TXT, -
        /PROTECTION = *PARAMETERS(-
               /ACCESS = *READ, -
               /USER-ACCESS = *ALL-USERS), -
        /MIGRATE = *ALLOWED
```

On the other hand, it is also permissible to write several commands in the same line, provided they are separated by semicolons:

```
/ IF (MODE == 'DEBUG'); WRITE-TEXT 'check file &A'; END-IF
```

2.3.3 Comments

Comments can be any text enclosed in double quotes and may be written anywhere within a command except within names, keywords, operators, numbers or character constants (for syntactical reasons). A comment may also be written in an otherwise empty command line:

```
/   "Preparations for calling procedure P.YY"
/COPY-FILE OLD-FILE "created by program A.23" , -
/          NEW-FILE "required in procedure P.YY"
```

The two-character string &* causes any line contents that follow it to be ignored (end-of-line comment). It can thus be used to (temporarily) comment out a command line without requiring specific handling of any quotes contained in that line:

```
/     START-EXE YZ-ERZEUGEN
/ &*  CALL-PROC P.CONVERT,(YZ-FILE) "Convert umlauts"
/     START-PERCON
//    ASSIGN-INPUT-FILE DISK-FILE(...)
```

2.4 Creating S procedures

2.4.1 Format of an S procedure

If a procedure is to make use of the programming facilities offered by SDF-P, it has to contain certain declaration and control commands in addition to the execution commands proper. The sample procedure below is intended to illustrate this; the meaning of each of the commands is subsequently explained.

This sample procedure outputs the last lines of a text file to SYSOUT (default: last ten lines):

```
/SET-PROC-OPT IMPLICIT-DECLARATION=*NO
/BEG-PARAM-DECL
/  DECL-PARAM FILE(TYPE=*STRING)
/  DECL-PARAM NUMBER-LINES(TYPE=*INTEGER,INIT=10)
/END-PARAM-DECL
/
/"For FILE a fully-qualified file name is expected       "
/"(without generation/version) or an element             "
/"specification in the format                            "
/" '[*LIBRARY-ELEMENT](library,element[(version)],type)' "
/
/"Required auxiliary variables:"
/DECL-VAR I(TYPE=*INTEGER)
/DECL-VAR DATA(TYPE=*STRING),MULTIPLE-ELEMENTS=*LIST
/DECL-VAR LIB(TYPE=*STRING)
/
/"Check whether the file/element exists to enable"
/"output of a specific error message"
/I = INDEX(FILE,'(')
/IF (I > 0)   "Name contains '(', therefore library"
/  LIB = SUBLIST(SUBSTR(FILE,I),1)   "library name"
/  IF NOT IS-CATALOGED-FILE(LIB)
/    WRITE-TEXT 'library &LIB does not exist!'
/    EXIT-PROCEDURE ERROR=*YES
/  END-IF
/  IF NOT IS-LIBRARY(LIB)
/    WRITE-TEXT 'file &LIB is not a library!'
/    EXIT-PROCEDURE ERROR=*YES
/  END-IF
/ELSE-IF NOT IS-CATALOGED-FILE(FILE)
/  WRITE-TEXT 'file &FILE does not exist!'
/  EXIT-PROCEDURE ERROR=*YES
/END-IF
```

```
/
/"Read data to a variable list"
/READ-VAR *LIST(DATA),STRING-QUOTES=*NO,INPUT=&FILE
/
/I = SIZE('DATA') - NUMBER-LINES + 1 "Calculate start position"
/IF (I < 1); I = 1; END-IF         "not before first line!"
/FOR I = *COUNTER( FROM = I, TO = SIZE('DATA'), INCREMENT = 1 )
/  WRITE-TEXT &(TO-C-LITERAL(DATA#I))
/END-FOR
```

2.4.2 Procedure head

The first five lines of the sample procedure above constitute the procedure head. The procedure head begins with the SET-PROCEDURE-OPTIONS command, which contains global definitions of the procedure format and default settings for important procedure attributes; some of these can be modified dynamically by means of the MODIFY-PROCEDURE-OPTIONS command. Specifying IMPLICIT-DECLARATIONS=*NO prevents a variable from being created (unintentionally) merely by assigning it a value. This facilitates the detection of typing errors in variable names.

Declaration of procedure parameters

Procedure parameters are special variables that may be assigned variable values when the procedure is called, e.g. by means of the CALL-PROCEDURE command. The declaration of procedure parameters is part of the procedure head: DECLARE-PARAMETER commands declare the name, type, default value and type of value transfer of each procedure parameter. If there is more than one DECLARE-PARAMETER command, they must be enclosed in a block starting with a BEGIN-PARAMETER-DECLARATION command and ending with an END-PARAMETER-DECLARATION command.

The sample procedure declares two parameters: FILE and NUMBER-LINES. The latter can accept integer values only (TYPE=*INTEGER); if no current value is supplied with the procedure call, the preset value (INITIAL-VALUE) for this parameter applies (INIT=10). The FILE parameter, on the other hand, must be assigned a character string (TYPE=*STRING) with the procedure call since no INITIAL-VALUE is defined.

The parameter variables can be assigned current values when the procedure is called; this is done by specifying the parameter values (e.g. in the CALL-PROCEDURE command) in the order of the parameter declarations. As an alternative, values can be assigned to parameters via the parameter names, which may be abbreviated as long as the abbreviations are unequivocal:

```
/CALL-PROCEDURE P.SHOW-TAIL,(FILE=PROTO.L,NUMBER-LINES=20)
/CALL-PROCEDURE P.SHOW-TAIL,(PROTO.L,20)
/CALL-PROCEDURE P.SHOW-TAIL,(N-L=20,F=PROTO.L)
/CALL-PROCEDURE P.SHOW-TAIL,(PROTO.L)
```

The first three of these commands are equivalents since FILE is the first and NUMBER-LINES the second parameter declared and both F and N-L are valid abbreviations for the parameter names. In the last procedure call, the default value (INITIAL-VALUE) of 10 will apply for NUMBER-LINES since no value is specified for this parameter.

If an error occurs during parameter transfer or during processing of the procedure head, the procedure will not be executed. Such errors can be handled only by the caller and not by the called procedure. This also applies if a procedure parameter is not supplied with any value, neither explicitly via a specification in the procedure call command nor implicitly via a defined INITIAL-VALUE.

No parameter declaration commands are required at all if a procedure does not expect any parameters. The SET-PROCEDURE-OPTIONS command may likewise be omitted if the preset values are to apply. This means that a very simple procedure may have no procedure head at all.

2.4.3 Procedure body

The part of an S procedure that follows the procedure head is referred to as the procedure body. It can contain normal SDF (or ISP) commands. These may be influenced by what is referred to as & replacement, i.e. by the current values of procedure parameters, by the current values of other variables and by accessing environment variables (e.g. user ID or current date). The procedure body may also contain SDF-P control flow commands; these serve to modify the sequence of command processing in accordance with the values of variables and functions.

2.4.4 S variables

In addition to the procedure parameters which are generated and assigned their values when the procedure is called, an S procedure may contain so-called S variables; these may be generated anywhere within a procedure. There are three ways of generating an S variable: explicitly by means of the DECLARE-VARIABLE command, or implicitly via the first value assignment to that variable (provided this is not prevented as a result of IMPLICIT-DECLARATION=*NO in the SET-PROCEDURE-OPTIONS command), or automatically during execution of a few special commands (such as OPEN-VARIABLE-CONTAINER or EXECUTE-CMD).

As for parameters, explicit declaration permits a type and an initial value to be defined for the variables.

The names of (simple) S variables may be up to 20 characters long; according to the rules for the SDF data type <structured-name>, these may be alphanumeric characters and hyphens. Names starting with SYS are reserved for the system software; a few other names are reserved for arithmetic operators and Boolean constants.

Value assignment

The value of a variable can be modified by various commands and also by programs. This also applies to procedure parameters since these are treated like other S variables during procedure execution.

The SET-VARIABLE command, in particular, serves to assign a variable a new value. Since the name of this command may not only be abbreviated in accordance with the general rules but even omitted altogether, the following format, which is familiar from programming languages, may be used to write assignments of new values to variables:

```
/ variable-name = new-value
```

Constants

The values assigned to the variables may be constant character strings, numerics or Boolean values (true/false).

Constant character strings (referred to as values of type STRING) are enclosed in single quotes; any single quote or & character contained in a STRING value must be doubled. Uppercase letters are distinguished from lowercase letters.

Character strings may alternatively be specified in hexadecimal format (referred to as X strings); this permits non-printable characters to be represented. Thus, the following two commands are equivalents:

```
/ FILE = 'ARB.4'
/ FILE = X'C1D9C24BF4'
```

Integers (values of type INTEGER) may be represented as decimal constants from the value range -2^{31} .. $+2^{31}-1$, where the positive sign is optional.

Boolean constants (values of type BOOLEAN) are designated by the keywords TRUE and FALSE. As an alternative, the names YES and ON or NO and OFF may be used.

Variable types

The type of a variable is determined by the type of the value assigned to it (i.e. STRING, INTEGER or BOOLEAN). To ensure early detection of incorrect value assignments and thus improve runtime security for procedures, the TYPE operand can be specified in the variable declaration to permit only values of a particular type to be accepted for a variable; all other value assignments then result in an error.

Scopes of variables

The scope of a variable is normally restricted to the procedure in which it was declared (either explicitly or implicitly). When the procedure terminates, the lifetime of the variable ends with it. If several calls of the same or different procedures within a task make use of the same variable and the variable is to retain its value, a so-called task variable can be created by explicit declaration with SCOPE=*TASK. This specification ensures that the variable can be used by various procedures.

If one procedure calls another procedure and both use variables with identical names, the variables are created and maintained separately at all times. The only exception to this rule occurs when a procedure is called via the INCLUDE-PROCEDURE command; in this case, it can access the variables of the calling procedure and create or modify variables there, provided SCOPE=*PROCEDURE has been specified in the variable declarations.

Variable containers

Variable definitions and values are normally held in the privileged memory allocated to the task and deleted at the end of the variables' lifetime. However, there are two options that permit the use of other storage media: a variable declaration with CONTAINER=*JV causes the variable value to be stored in a job variable, enabling it to be retained for longer periods of time and making it available to other tasks for querying and modification via JV access.

If the variable declaration specifies a container in a library (via the OPEN-VARIABLE-CONTAINER command), both the variable description and its value can be stored in the library, either on request by means of the SAVE-VARIABLE-CONTAINER command or automatically at the end of the procedure or task; they will then be available in subsequent procedure runs.

2.4.5 Expressions

As an alternative to constants of one of the types STRING, INTEGER or BOOLEAN as representations of values, assignments can make use of already existing variable values and results from function calls; these can be linked to each other or to constants by means of various operations. Such complex representations are referred to as expressions; they resemble the corresponding constructs of higher-level programming languages in function as well as notation.
Each expression supplies a result of one of the three basic types. The simplest case is an expression consisting of a single constant, the name of a variable (which must have a simple value) or a function call. These so-called base terms can be linked via operators; for each operator, the permissible type(s) of operands and results supplied by it are defined.

Expressions can contain the following operators:

Arithmetic operators	`+, -, *, /, MOD`
Relational operators:	`<,  >,  <=, >=, ==, <>`
or (equivalent operators):	`LT, GT, LE, GE, EQ, NE`
Logical operators:	`NOT, OR, AND, XOR`
String operator (concatenation):	//

A more detailed description of these operators is given as of page 256.

Where relational operators are used, both operands must have the same type (STRING, INTEGER or BOOLEAN; in the latter case, they can only be compared to see whether they are equal or not equal.) The comparison is performed depending on the operand type: the expression 12 > 9 will supply the value TRUE, while the result of the string comparison ’12’ > ’9’ will be FALSE, since strings are compared character-by-character from left to right.

Since the hyphen can occur both in variable names and as an arithmetic operator (minus sign) in expressions, it must be preceded and followed by a blank when used as an operator in cases where it may be interpreted as a name part. For instance, the following command

```
/ I = I - 1
```

subtracts the constant 1 from the variable I and assigns the result to the same variable as its new value. The following command, however,

```
/ I = I-1
```

assigns the value of a variable I-1 to variable I.

The operators consisting of a string of letters such as MOD, EQ or AND must, of course, also be separated from any names or numbers preceding or following them. To enhance readability, it is generally advisable to write a blank before and after all operators, even in cases where syntactical analysis does not require it.

In addition to assignments by means of SET-VARIABLE, expressions may be used as operands in a few other commands (in particular those described in this manual).

2.4.6 Function calls

While a variable will return the value last stored in it unmodified each time it is accessed, a function call causes an internal SDF-P algorithm to be executed that will supply a result. The purpose of the function determines which information will be accessed: while string processing and data conversion functions calculate the result value from the parameters supplied with the function call, other functions may supply information about the runtime environment (e.g. user ID, terminal name, date, time, file attributes) or return codes from previous command calls which they find by means of internal operating system calls.

Within an expression, a function is addressed via its name, followed by a list of parameters enclosed in parentheses. The list may be empty if the function does not require any parameters (e.g. the function TSN( ) which supplies the current task's TSN) or if the preset parameter values are used. For instance, both function calls DATE( ) and DATE(FORMAT= *ISO) supply the current date in ISO format; the German format can be requested explicitly by means of the call DATE(FORMAT=*GERMAN). If the parameter list is empty, the parentheses may be omitted as well, provided the function name cannot be confused with a variable name.

Function parameters are specified in a similar way as SDF command operands: they can be addressed either via their position or their name, and their names may be abbreviated as long as they remain unequivocal. Parameter values may be keywords (prefixed by an asterisk) or expressions of one of the types STRING, INTEGER or BOOLEAN. Note that variable names are replaced by variable values when passed to a function. If the function is to access the variable name (e.g. to determine whether the variable has been defined), the name must be enclosed in quotes to identify it as a string constant; for example:

```
/ I = INDEX(FILE,'(')
/ B = IS-INITIALIZED('OUTPUT')
```

The first call accesses the contents of the string variable FILE and searches for the first occurrence of an opening parenthesis; the position of the parenthesis (or zero if no parenthesis is found) is assigned to variable I. The second command checks whether the variable OUTPUT has a valid value and assigns variable B either of the values TRUE or FALSE, depending on the result of the check.

In addition to the predefined functions supplied with SDF-P, there are system administration functions that may be implemented by the system administrator and made available to all users, if required.

The description of all predefined functions and their parameters starts on page page 347. A detailed description of the notation for function parameters is contained in section “Input parameters in function calls” on page 225ff.

2.4.7 Conditions and loops

The execution of an S procedure can be controlled via control flow commands. These can be used to build command sequences that are executed subject to a condition (IF blocks), and loops (WHILE, REPEAT-UNTIL and FOR blocks); both are similar to those used in higher-level programming languages. Each of these conditional or repetitive command sequences begins and ends with a specific command:

```
/IF (condition)
/  "then-command sequence"
/END-IF

/WHILE (condition)
/  "loop body"
/END-WHILE

/REPEAT
/  "loop body, executed at least once"
/UNTIL (condition)

/FOR variable = valuelist, CONDITION = (condition)
/  "loop body"
/END-FOR
```

The control structures may be nested in any order since each is identified unequivocally by its start and end commands. The IF block may contain additional queries of other conditions and/or an ELSE branch:

```
/IF (condition-1)
/  "commands executed ..."
/  "... if condition-1 is TRUE"
/ELSE-IF (condition-2)
/  "... if condition-1 is FALSE but condition-2 is TRUE"
/  "    (any number of ELSE-IF is possible"
/ELSE
/  "... if none of the conditions was fullfilled"
/END-IF
```

For further clarity, the beginning and end of a block can be identified by tags, whose correspondence will be checked by SDF-P:

```
/&* This command sequence removes all trailing blanks from
/&* a string variable TEXT. TEXT should contain at least
/&* one character that is not equal to ' '.
/BLANKS-OFF: WHILE (SUBSTR(TEXT,LENGTH(TEXT),1) == ' ')
/  TEXT = SUBSTR(TEXT,1,LENGTH(TEXT) - 1)  "truncate one byte"
/END-WHILE BLANKS-OFF
```

Such tags can be referenced by the EXIT-BLOCK command in order to exit an enclosing block before reaching its end. By referencing the tag of an enclosing loop, the CYCLE command can be used to initiate the next loop pass although the loop body has not yet been completely executed.

A BEGIN block resembles a loop in that it encompasses a command sequence, but it cannot influence the sequence of command execution. In conjunction with an EXIT-BLOCK, a BEGIN block can, for instance, be used to exit from a procedure section prematurely subject to certain conditions:

```
/DAILYTASK: BEGIN-BLOCK
/  "Tasks to be performed daily"
/  IF (DAY() == 'SAT' OR DAY() == 'SUN')
/    EXIT-BLOCK DAILYTASK
/  END-IF
/  "Tasks to be performed on weekdays only"
/  IF (SUBSTR(DATE(),9,2) <> '01')
/    EXIT-BLOCK DAILYTASK
/  END-IF
/  "Tasks to be performed only on the first day of the month"
/END-BLOCK DAILYTASK
```

This may improve the clarity of the procedure structure as compared with numerous nested IF blocks.

Those programmers who wish to continue using GOTO despite the command’s bad reputation will appreciate the fact that this command has been implemented as well; the branch destination should be specified as a tag prefixed to the destination command:

```
/IF (...); GOTO CLEANUP; END-IF
/"further commands"
/CLEANUP: DELETE-FILE ...
```

Note that the usual restrictions apply (e.g. no GOTO to enter a block).

2.4.8 & replacement

Variable values can be used to modify commands even if the use of expressions in their operand values is not permissible. In S procedures, SDF-P expressions are replaced by their value prior to command analysis proper and may occur in almost any position in command operands and command names (there are a few restrictions for some commands such as SDF-P control flow commands). The expression must be enclosed in parentheses at the appropriate position in the command and preceded by the escape symbol &. If the value of the expression is of type INTEGER or BOOLEAN, it is first converted to type STRING. The simplest expression consists of no more than the name of a simple variable; the parentheses may be omitted in this case and a period must be written to separate the variable name from any text following it unless it is followed by a blank or special character anyway:

```
/TIME = 40
/ENTER-JOB E.TEST1,CPU-LIMIT=&TIME           "Limit = 40 seconds"
/ENTER-JOB E.TEST2,CPU-LIMIT=&(TIME * 60)    "Limit = 40 minutes"
/ENTER-JOB E.TEST3,JOB-CLASS=JC&TIME.MAX     "Job class = JC40MAX"
```

Job variable values, whose syntax is similar to that in non-S procedures, are inserted in S procedures by calling the predefined function JV:

```
/&* check the following!
/SHOW-JV JV-CONTENTS=$USERXY.TSN
/&*  the JV is to contain the TSN of a running task
/SHOW-JOB-STATUS TSN=&(JV('$USERXY.TSN'))
```

In most cases, however, it will be easier to use the job variable as a container for an S variable:

```
/DECL-VAR DB-TSN(TYPE=STRING),CONTAINER=*JV($USERXY.TSN)
/  "..."
/SHOW-JOB-STATUS TSN=&DB-TSN
```

Note that recursive & replacement is not possible, i.e. if an & replacement generates another escape symbol &, this does not trigger another replacement.

2.4.9 Arrays and lists

In addition to the simple variables which can contain precisely one value of type STRING, INTEGER or BOOLEAN, there also exist variables which can contain numerous values of the same type. Arrays are one example: they offer a (predefinable) number of locations that can optionally be addressed via an index value:

```
/DECLARE-VARIABLE COST(TYPE=*INTEGER), -
/                 MULTIPLE-ELEMENTS = *ARRAY(1990,2099)
/COST#1996 = 8000 "cents per hour"
/YEAR = 1996
/WHILE (YEAR < 2005)    "loop over 10 years"
/  NEW-COST = (COST#YEAR * 105 + 50) / 100    "+ 5% per year"
/  YEAR = YEAR + 1
/  COST#YEAR = NEW-COST
/END-WHILE
```

The index value, specified as a number or the name of a simple integer variable, is linked to the variable name via the character #. Any number of array elements can be created, addressed and deleted independently of each other as long as they are taken from the valid index range as defined at variable declaration (see also the FREE-VARIABLE command).

Lists, on the other hand, always consist of a sequence of values that are consecutively numbered, starting with 1. Values can be added only at the end or the beginning of the list; in the latter case, the indexes of all existing values are automatically incremented by 1. Since the sequence of list elements without gaps is always ensured, lists are especially suitable for processing in loops (FOR command). In addition, the commands READ-VARIABLE and SHOW-VARIABLE support record-by-record transfer of file contents to and from lists. The sample procedure below writes the records of a file in reverse order to an output file, making use of two list variables:

```
/DECL-PARAM (FROM,TO)
/DECL-VAR (IN,OUT),MULT-ELEM=*LIST
/READ-VAR *LIST(IN),STR-QUOTES=*NO,INPUT=&FROM
/FOR LINE=*LIST(IN)
/  OUT = LINE, WRITE-MODE=*PREFIX
/END-FOR
/SHOW-VAR OUT,PREFIX=*NO,OUTPUT=&TO    "###"
```

The predefined array SYSSWITCH is a special case. Its 32 BOOLEAN-type elements (SYSSWITCH#0 through SYSSWITCH#31) are internally mapped to the job switches of the current job, enabling these to be set and queried directly:

```
/SW4-BEFORE = SYSSWITCH#4       "###"
/SYSSWITCH#4 = TRUE
/START-EXECUTABLE-PROGRAM $EDT
@...
@HALT
/SYSSWITCH#4 = SW4-BEFORE
```

2.4.10 Structures

In contrast to arrays and lists, which combine a number of values of the same type under a single variable name, a STRUCTURE-type variable permits several elements of any type to be combined, where each element is identified by its own name. The element names are appended to the name of the structure variable, separated from it by a period; this enables elements to be addressed individually:

```
/DECL-VAR SALES(TYPE=*STRUCTURE)
/SALES.HEAD = 'H. Haegar'
/SALES.NO-EMPLOYEES = 17
/SALES.TURNOVER = 99000
```

The structure elements may again be arrays, lists or structures:

```
/DECL-ELEM SALES.PERSONEL(TYPE=*STRUCTURE),MULT-ELEM=*LIST
/SALES.PERSONEL#1.NAME = 'S. Lucky'
/SALES.PERSONEL#1.PERS-ID = 13
/SALES.PERSONEL#2.NAME = 'D. Harry'
/SALES.PERSONEL#2.PERS-ID = 17
```

The example below illustrates how a list of structures can be used as a simple database. To create elements and assign values, a special form of the SET-VARIABLE command is used; it expects a string in the format of an SDF command substructure from which it generates individual value assignments. The entries in the “database” are processed sequentially by means of FOR loops, presenting the various possible inputs to the user who will then select the desired entry.

```
/DECL-VAR JOB-START(TYPE=*STRUCT),MULT-ELEM=*LIST
/DECL-VAR JOB(TYPE=STRUCT)
/
/&*     Set up list of job definitions
/JOB-START=*STR-TO-VAR('PAR(NAME=CTL,CPU=60,FILE=E.CONTROL)'), -
/          WRITE-MODE=*EXTEND
/JOB-START=*STR-TO-VAR('PAR(NAME=DB,CPU=999,FILE=$AV.DB-START)'), -
/          WRITE-MODE=*EXTEND
/"... any number of job definitions as required ..."
/
/&*     Show all defined jobs
/TEXT = '  Selection:'
/FOR JOB=*LIST(JOB-START)
/  TEXT = TEXT // ' ' // JOB.NAME
/END-FOR
/WRITE-TEXT '&TEXT'
/
/&*     Selection by the user
/READ-VAR JNAM,PROMPT=' Which job would you like?'
/
/&*     Search list for appropriate job and start
/FOR JOB=*LIST(JOB-START)
/  IF (UPPER-CASE(JNAM) == UPPER-CASE(JOB.NAME))
/    ENTER-JOB &(JOB.FILE),CPU-LIMIT=&(JOB.CPU)
/  END-IF
/END-FOR
```

Structures for which any number of elements may be created (and deleted) during the procedure run are referred to as “dynamic structures”. On the other hand, the complete layout of a structure, including the names and types of all its elements, can be defined at the time of variable declaration. The following commands create a layout under the name DEPARTMENT which is used exclusively for the declaration of a variable (as “static structure”):

```
/BEGIN-STRUCTURE DEPARTMENT
/  DECLARE-ELEMENT HEAD(TYPE=*STRING)
/  DECLARE-ELEMENT NO-EMPL(TYPE=*INTEGER)
/  DECLARE-ELEMENT TURNOVER(TYPE=*INTEGER)
/END-STRUCTURE DEPARTMENT
/DECL-VAR MEDICAL(TYPE=*STRUCT(DEFINITION=DEPARTMENT))
/MEDICAL.HEAD = 'Dr. Zook'
/MEDICAL.NO-EMPL = 2; MEDICAL.TURNOVER = 99001
```

SET-VARIABLE can be used to assign a complete structure to another; the WRITE-MODE operand then serves to determine whether elements of the destination structure that are not addressed are to be retained. If the destination structure is a static structure, those elements of the source structure not present in the destination variable will be ignored during assignment.

Some system commands can optionally reroute their output to structure variables. The SHOW-FILE-ATTRIBUTES command, for instance, can output a structure whose elements contain all catalog information for a file. This information can then be processed in an S procedure with direct access to the structure elements (see section “Structured output in S variables” on page 195 for further details).

2.4.11 Error handling

As in other BS2000 command sequences, an error in one of the commands in an S procedure will prevent execution of all subsequent commands. In non-S procedures, this behavior is referred to as “spin-off”, in S procedures as “SDF-P error handling”. Unless otherwise specified, it will cause the procedure run to be aborted; the caller will receive an error report. This again results in a spin-off or SDF-P error handling in the calling procedure, so that eventually the batch task will be aborted or, in interactive mode, the system prompt will appear on the screen.

If it is possible (or even necessary) to continue processing even after a command error has occurred in an S procedure, the error must be intercepted at an appropriate location in the procedure by means of an error handling block:

```
/DELETE-FILE FILE.A
/DELETE-FILE FILE.B
/DELETE-FILE FILE.C
/IF-BLOCK-ERROR
/  WRITE-TEXT 'At least one of the files cannot be opened.'
/END-IF
/"Continue processing"
```

The command sequence initiated by IF-BLOCK-ERROR is not executed unless one of the preceding commands triggers SDF-P error handling. The procedure may also contain an ELSE branch analogous to the IF block to be executed when no error occurs. This prevents the spin-off and procedure execution can be continued normally after the corresponding END-IF.

To supply more detailed information for evaluation than the mere “OK” or “not OK” returned by SDF-P error handling, each command reports the result of its execution in a standardized return code consisting of the components maincode, subcode1 and subcode2. These are automatically saved by SDF-P if an error occurs. If no error condition is set, the SAVE-RETURNCODE command can be used to save the return code from the command executed last. The saved values are made available for use in SDF-P expressions by the predefined functions MAINCODE, SUBCODE1 and SUBCODE2.

Unless otherwise specified, SDF-P error handling is triggered by the same situations which result in a spin-off in non-S procedures. Specifying ERROR-MECHANISM=*BY-RETURNCODE (SET-PROCEDURE-OPTIONS command) makes error handling dependent on the return code (subcode1 not equal to zero results in an error). For more details see section “Error handling” on page 69.

The command name IF-BLOCK-ERROR is meant to indicate that the error handling it initiates is block-oriented. It takes effect only if the error occurs in the same block or a block nested in that block. If an error occurs, any error handling contained in a new block that starts after the error will be ignored:

```
/DELETE-FILE FILE.A
/BEGIN-BLOCK
/  DELETE-FILE #TEMP.B
/  DELETE-FILE #TEMP.C
/  IF-BLOCK-ERROR
/    WRITE-TEXT 'error while deleting temporary files'
/  END-IF
/END-BLOCK
```

In this example, an error during deletion of FILE.A will cause the entire BEGIN block that follows to be skipped; the error handling takes effect only if an error occurs in the DELETE-FILE commands for the files #TEMP.B and #TEMP.C which are part of the same block.

To permit error handling to take effect for a single command without having to enclose the command in a separate BEGIN block, SDF-P provides the IF-CMD-ERROR command. This command initiates a block that is not executed unless an error occurs in the command immediately preceding the block.

2.4.12 Programming statement sequences

Normally, if a program attempts to read statements from an S procedure and the next line of the procedure contains a command, the EOF condition is set for statement input, causing the program to terminate. This behavior is meant to prevent that a missing END statement will cause the program to continue, which may result in uncontrolled termination when another program is loaded at a later time.

On the other hand, it may be desirable to control the processing sequence of commands as well as statements via SDF-P control structures. This is supported by a BEGIN block with PROGRAM-INPUT=*MIXED-WITH-CMD which may contain a mixture of commands and statements. If the program expects a statement and detects a command instead, it is interrupted and resumed with the next statement:

```
/DECL-PARAM CONTROLFILE
/&*     In S procedures, SYSDTA and SYSSTMT are
/&*     assigned to SYSCMD by default
/DECL-VAR ADDLIST,MULT-ELEM=*LIST
/READ-VAR *LIST(ADDLIST),STRING-QUOTES=*NO,INPUT=&CONTROLFILE
/SHOW-VAR ADDLIST
/BEGIN-BLOCK PROGRAM-INPUT=*MIXED-WITH-CMD
/  START-LMS
//   OPEN-LIB PROJECT-LIB,*UPDATE
/    FOR NAME=*LIST(ADDLIST)
//     ADD-ELEM PROJ.&NAME,*LIB-ELEM(*STD,&NAME(*INCR),S)
/    END-FOR
// END
/END-BLOCK
```

In this example, a number of names are read into a list variable. Subsequently, the LMS statement ADD-ELEMENT is executed once for each list element, the name being repeatedly inserted in the statement by means of & replacement.

By analogy, data records can be programmed in the same way as statements. The SEND-DATA command enables the S procedure syntax to be used for data lines as well (e.g. with respect to indentation or continuation handling) and ensures that the use of variables will not cause any problems (by default, no & replacement is performed in data records). SEND-DATA expects a string expression as operand; the value of this operand is passed as a data line to a program that reads from SYSDTA. As an alternative, the operand value *EOF can be used to set an end condition for SYSDTA.

2.4.13 Storing command output in variables

If the data and messages output by a command are to be collected for further analysis, output intended for SYSOUT may, for instance, be rerouted to a file or list variable and subsequently processed by string access functions. This method is, however, cumbersome and error-prone. Instead, the EXECUTE-CMD command can be used to output the text and return code supplied by any command to a variable directly. In addition, output from certain commands can be stored in variable structures in the form of pure user data that is then directly available for programming. The example below illustrates this method: a list of file names is compiled which is then used in a loop as input to the TRANSFER-FILE command:

```
/DECL-PARAM PATTERN
/DECL-VAR FILELIST(TYPE=*STRUCT),MULT-ELEM=*LIST
/DECL-VAR FILE(TYPE=*STRUCT)
/EXEC-CMD (SHOW-FILE-ATTR &PATTERN), -
/   TEXT-OUTPUT=*NONE,      "no output to SYSOUT" -
/   STRUCT-OUTPUT=FILELIST,  "struct. output in variable" -
/   RETURNCODE=*VAR(SUBCODE1=S1,     "== 0 if ok" -
/                   SUBCODE2=*NONE,  "not required" -
/                   MAINCODE=M)      "error code if any"
/IF (S1 > 0)
/  WRITE-TEXT 'error &M'
/  EXIT-PROC ERROR=*YES(MAIN=&M)   "return error to caller"
/END-IF
/FOR FILE = *LIST(FILELIST)
/  TRANSFER-FILE TO,HOSTXYZ,(&(FILE.F-NAME)),(...)
/END-FOR
```

The option of storing the messages output during command execution in variable structures is also provided. Message codes and variable inserts are stored separately, enabling them to be processed directly without analysis of the body of the message text.

A detailed description of the EXECUTE-CMD command is given on page 648. The output structures generated by SDF-P commands are described in section “Structured output in S variables” on page 195; information on other commands that support structured output can be found in the appropriate manuals.

2.4.14 S variable streams

While ordinary system files such as SYSDTA, SYSOUT and SYSLST merely transfer sequences of data records, where each data record corresponds to a text string, S variable streams transfer entire variable structures. The TRANSMIT-BY-STREAM command transfers such a structure to the stream; the stream must already be assigned to a suitable server that will accept the structure, process it and acknowledge it with return information.

FHS is available for use as such a server. FHS can output the structure contents to the terminal and return user input via predefined screen masks. A detailed description of "FHS as output server" is given in section "FHS as output server" on page 201.

A variable can in turn be used as a server that accepts the structures transferred via a stream for further analysis.

While output from a single command can be directed to a variable by means of the EXECUTE-CMD command, the predefined system streams SYSINF, SYSMSG and SYSVAR continually receive structured command and message outputs for various commands.

2.4.15 Using SDF-P interactively

Input in interactive mode is in many respects treated like a special S procedure: variables (including procedure-local variables) can be declared and created, a string of commands, separated by semicolons, can be entered as well as control structure blocks. For instance, if the WHILE command is entered at the terminal, no subsequent command will be executed immediately; they will be stored temporarily until the corresponding END-WHILE command is entered. The control structure can then be analyzed and executed. User support for interactive entry of control structures is provided by the fact that the system prompt always indicates which block needs to be closed next. This enables the interactive user to "try out" loops, condition constructs and variable access interactively prior to including them in a procedure.

There are, of course, a few restrictions that do not apply to "genuine" S procedures: for instance, GOTO cannot be used in interactive mode unless as part of block entries, continuation lines cannot start with insignificant blanks (since the slash need not be entered), and data lines in blocks can be created by means of SEND-DATA only (again because the slash, which serves to distinguish command lines from data lines, may be omitted).

The rules for interactive command input are described in more detail as of page 76.

2.4.16 Runtime security

The SET-PROCEDURE-OPTIONS command offers various options for protecting a procedure: both logging and interrupting can be prevented for a procedure. In combination with setting appropriate attributes for the procedure container (e.g. read access restricted via BASIC-ACL or GUARDS), these options can be used to prevent that procedure being read by an unauthorized caller or its execution being modified illegally.

The parameters transferred with a procedure call can either be thoroughly checked within the procedure (e.g. by the CHECK-DATA-TYPE function), or verified by SDF prior to procedure execution by embedding the procedure call in a command (SDF-A command definition with IMPLEMENTOR=*PROCEDURE). The SYSTEM-CALL function can be used to check whether SDF analyzed the call syntax using a system or group syntax file.

Generally speaking, runtime security can be enhanced for a procedure by (preferably) complete declaration of all variables used (including type definitions) and layouts. The implicit creation of variables, e. g. as a result of typing errors, can be prevented by specifying IMPLICIT-DECLARATION=*NO in the SET-PROCEDURE-OPTIONS command.

2.4.17 Testing procedures

To facilitate error detection and location in procedures, it is often advisable, during testing, to enable global logging of all procedures called by means of the following command (provided this is permitted by the procedures):

```
/ MODIFY-PROCEDURE-TEST-OPTIONS LOGGING=*YES
```

For the purposes of specific tracing, a procedure can be interrupted where appropriate by pressing the $\boxed{\text{K2}}$ key or inserting a HOLD-PROCEDURE command. The contents of the variables of the interrupted procedure can then be inspected (e.g. using the command SHOW-VARIABLE command) or modified in interactive mode. The procedure can subsequently be continued step-by-step using TRACE-PROCEDURE or up to its end or the next interrupt by means of RESUME-PROCEDURE.

3 The procedure concept in SDF-P

SDF-P introduces a new procedure format. Procedures with this format are called “structured procedures”, or S procedures for short. Procedures which do not comply with this format are called non-S procedures.

Procedures are always sequences of commands, statements and data records that are stored in a “procedure container”. SDF-P supports both BS2000 user files and library elements as procedure containers. This means that, like non-S procedures, S procedures can be stored in user files or in PLAM libraries. In addition, S procedures can also be stored in list variables.

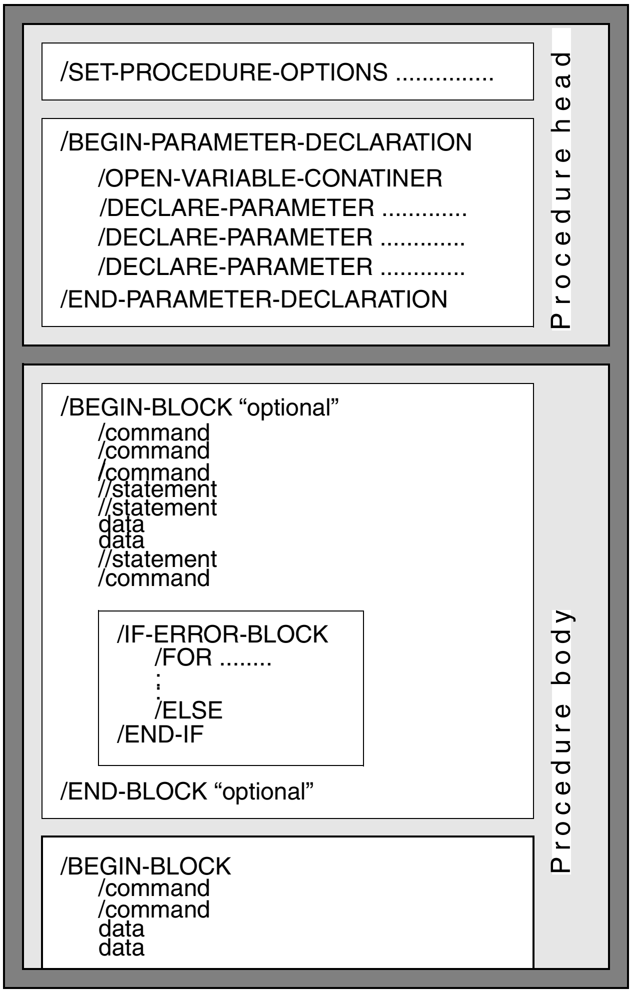
Apart from data and statements, commands are the most important element in S procedures. The other elements are variables, functions and expressions: variables are named data objects to which a content can be assigned. A function determines a unique result from input parameters, and this result is then used instead of the function. Expressions consist of operands and operators. The resulting value from the expression is used instead of the expression.

Unlike non-S procedures, S procedures do not begin with one of the commands /BEGIN-PROCEDURE or /SET-LOGON-PARAMETERS; consequently, when creating S procedures it is not necessary to take into account whether the procedure will later be invoked as a foreground or a background procedure.

An S procedure does not need to be terminated with a special procedure termination command. However, SDF-P does provide such a command, namely /EXIT-PROCEDURE. Using this command, execution of the procedure can be terminated at any desired point, and error information can be returned to the procedure caller. (For further information about the termination behavior of S procedures, see chapter “Calling and controlling procedures” on page 105.)

3.1 Structured procedure format

Under the structured procedure format, an S procedure consists of a procedure head and a procedure body. Within each of these parts, logically associated blocks can be defined. This principle not only results in procedures which are very clear, but it also embodies a practical functional element.



```
/SET-PROCEDURE-OPTIONS ...............

/BEGIN-PARAMETER-DECLARATION
    /OPEN-VARIABLE-CONATINER
    /DECLARE-PARAMETER .............
    /DECLARE-PARAMETER .............
    /DECLARE-PARAMETER .............
/END-PARAMETER-DECLARATION

/BEGIN-BLOCK “optional”
    /command
    /command
    /command
    //statement
    //statement
    data
    data
    //statement
    /command

    /IF-ERROR-BLOCK
        /FOR ........
        :
        /ELSE
    /END-IF

/END-BLOCK “optional”

/BEGIN-BLOCK
    /command
    /command
    data
    data
```

The procedure head always begins with the command /SET-PROCEDURE-OPTIONS, which is used to define the attributes of the procedure. This is followed, where necessary, by a declaration of the procedure parameters.

The procedure body contains the commands, statements and program data. Commands which are logically associated together form a command block, which in turn consists of a part containing the commands, statements and data, and an error-handling part. The command block is enclosed between a /BEGIN-BLOCK and an /END-BLOCK. It is possible to define particular attributes for this block, e.g. different handling of the commands and data. If an error occurs, the assigned /IF-ERROR-BLOCK will automatically be processed. The procedure should be terminated by the /EXIT-PROC command.

3.2 Conventions for S procedures

This section explains the structure of procedure lines, then deals with data and statements and finally describes the rules for expression replacement and & replacement together with the rules for reading in program data or statements.

3.2.1 Procedure lines

The procedure lines contain the commands, statements and data for a procedure.

The most important points to be taken into account are:

- procedure line length
- first character of procedure lines
- command length
- separation of commands
- continuation handling
- comment syntax
- tags

Procedure line length

The length in which procedure lines are evaluated is influenced by means of the INPUT-FORMAT operand of the SET-PROCEDURE-OPTIONS command (the SET-PROCEDURE-OPTIONS command is a component of the procedure head and is therefore described in detail in section “Creating the procedure head” on page 81).

The default setting for the length of procedure lines is INPUT-FORMAT = *FREE-RECORD-LENGTH. This means that in S procedures (in contrast to non-S procedures), the procedure lines are evaluated in their full length or up to the continuation character (see section “Continuation handling” on page 53).

For reasons of compatibility with non-S procedures, the operand INPUT-FORMAT = *BY-SDF-OPTION was introduced in the SET-PROCEDURE-OPTIONS command. The effect of this operand is that procedure lines containing commands are evaluated only up to column 72. The column that must then contain the continuation character depends on what has been set for the CONTINUATION operand in the SDF command MODIFY-SDF-OPTIONS.

Note

ISAM files are accepted as S procedures only if they have KEY-POS = 5 and KEY-LEN = 8.

First character of procedure lines

The following distinctions must be made:

– The first procedure line must begin with *a single* slash (/).
– All other procedure lines that begin with only one slash contain commands. SDF expects commands from the logical system file SYSCMD (see “Commands, Vol. 1-5”[3]).
– Procedure lines that begin with two slashes contain statements in the SDF format (to a program with SDF interface). SDF expects statements from the logical system file SYSSTMT, for which the same assignment applies as for the system file SYSDTA (see “Commands, Vol. 1-5” [3]).
– Procedure lines which do not start with a slash are data lines. These contain program input data.
– The first relevant character of any command continuation lines must also be a slash. The first relevant characters in each statement continuation line must be two slashes. The way that the continuation of data lines is handled will depend on the program doing the processing.
– Commands, statements and data must not be chained together in one procedure line.

Command length

If the command length is checked, you must consider whether an expression replacement is contained in the command call. Commands must not be longer than 16364 characters following an expression replacement.

Separation of commands

Each procedure line can contain several commands. These commands must be separated from one another by semicolons.

The first command is located at the beginning of a procedure line and must begin with a slash (/). If a procedure line contains several commands, these must be separated by semicolons and the individual commands must not be introduced by slashes.

Commands which are written after AID commands, and are separated from the latter by semicolons, will be processed as part of the AID command sequence; i.e. they are not treated as part of the procedure input, but instead are processed directly by AID.

Continuation handling

Commands, command sequences or statements can be distributed over several lines. “Continuation handling” determines how associated lines are recognized and evaluated.

In S procedures, up to 16364 characters (4090 characters for ISP commands) can be linked by means of continuation lines to form a command sequence.

The continuation character is a hyphen (–). The position in the procedure line at which the continuation character can be located is implicitly set by means of the INPUT-FORMAT operand in the SET-PROCEDURE-OPTIONS command.

Unless otherwise specified, the continuation character can be located in any column of the procedure line (INPUT-FORMAT = *FREE-RECORD-LENGTH). However, the continuation character must not be followed by any other characters; it must be the last character (apart from trailing blanks) in the procedure line.

If INPUT-FORMAT = *BY-SDF-OPTION applies, the setting of the CONTINUATION operand in the MODIFY-SDF-OPTION command must be taken into account:

– With CONTINUATION = *OLD-MODE, the continuation character for commands must be in column 72.
– With CONTINUATION = *NEW-MODE, the continuation character can be located in any column from 2 to 72.

Comments enclosed in quotation marks

Comments enclosed in quotation marks are used for the internal documentation of procedures. Comments can be used anywhere within a command or statement where blanks are permitted (except after a continuation character).

A comment can be any text that is enclosed in quotation marks (“ ”).

Even if a comment is written on a separate line, it must be enclosed in quotation marks, and follow the introductory (/). SDF-P then interprets such a procedure line as a command which consists solely of a comment. This will not be executed or logged. If such a “comment-command” is given a tag, only this tag will be logged. In addition, expression replacement is not performed for any such comments which are written on a separate line.

End-of-line comments

End-of-line comments are particularly important in the development of S procedures. End-of-line comments are marked by the character pair “&*”. Text after this character pair will be ignored when the procedure is evaluated. Consequently, any continuation characters, character separators (such as semicolons) and & characters must be written before the end-of-line character pair.

Examples

The character pair “&*” can be used to append a note after a command:

```
/PRINT-DOC &FILE &* My file is printed out here.
```

or it might be used to modify the input:

```
(1)  PRINT-DOC &FILE ,PRINT-JOB-NAME='&(SUBSTRING(FILE,1,8))'
```

```
(2)  PRINT-DOC &FILE &*,PRINT-JOB-NAME='&(SUBSTRING(FILE,1,8))'
```

In the case of (1) the PRINT-DOCUMENT command will be modified, whereas in the case of (2) the simple insertion of “&*” means that the default setting will be used instead. This allows the first form (1) to be very easily restored; all that has to be done is delete the “&*” again.

Notes

- Until now the character string “&*” was not permitted in S procedures - even when enclosed in quotation marks. The only place where it was previously not forbidden was in comment lines containing no operation names, e.g. “ cmt1 &* cmt2 ”.
  This should be taken into account if any incompatibilities arise with procedures which were created earlier.
- The character pair “&*” must not be split by blanks or any other characters.
- It is possible to incorporate the character pair “&*” as part of the input; this requires the & character to be duplicated: &&*.
  However, care must be taken to avoid repeating the & character too often. For example, “&&&*” will defeat the intention of canceling the special function of the character pair “&*”. In this case, the character string would again indicate an end of line comment.

Tags

Procedure lines can be provided with tags. These tags can then be used as branch destinations, for command block nesting or for branches made by branch commands. They are also known as S tags.

The following rules apply to S tags:

- SDF data type: <structured-name>
- Maximum length: 255 characters
- Character set: A...Z, 0...9, $, #, @, -
- First character: letter
- Last character: colon
- The colon must follow the tag directly, without spaces
- Tags must not be generated by expression replacement
- Tags must be located before the operation name and be separated from this name by at least one space. (The “operation name” is the name used to call a command.)

For information on non-S tag handling (.tag), also see the conversion instructions in chapter “Converting non-S procedures” on page 287ff.

3.2.2 Expression replacement

Expression replacement plays an important part in calling commands. It allows you to generate commands dynamically and modify the contents of procedure lines.
(For more information on expressions, see chapter “Expressions” on page 249.)

Expression replacement is carried out in such a way that the final inputs, or partial inputs, do not have to be specified, but only placeholders for them, which are then replaced by actual values at execution time.

Escape character

An escape character is the character that initiates expression replacement. This character immediately precedes the expression which is to be replaced and indicates that the subsequent characters do not represent the actual operand value, but only a placeholder.

A distinction must be made between two uses of the escape character:

- commands and statements
- data lines.

For commands and statements, the escape character is &.

The character that is to serve as an escape character in data lines can be determined in the procedure head using the SET-PROCEDURE-OPTIONS command (DATA-ESCAPE-CHAR operand). The characters &, #, *, @ and $ are available for selection.

Unless otherwise specified, expression replacement does not occur in data lines; DATA-ESCAPE-CHAR = *NONE applies.

Because the character & is the default escape character, expression replacement is also referred to below as & replacement.

It should be noted that if an expression or a variable is preceded by an escape character, the value of this expression or variable will not be processed directly by the command, but instead as though its value had been written in place of the escape character string in the input record. (Direct expression replacement, for example /SET-VARIABLE A=B, will only be performed by a command if the command supports an expression, otherwise escape character strings must by used)

The rules described in the sections below apply to expression replacement in commands and statements and in data lines.

Example

```
/SHOW-FILE-ATTRIBUTES &FILE
```

This displays the file attributes of a file whose name is stored in the FILE variable.

Syntax

In structured procedures, the escape character can be applied not only to variables (including procedure parameters), but also to functions, expressions and job variables. The following syntax applies for expression replacement:

&(expression)

or

&name

Where:

&	Escape character
expression:	An expression or the name of a job variable
name:	The name of a variable (if it contains no period) or the name of a predefined function without parameters

Rules

“expression” is evaluated, and the result value is converted to the STRING data type and then used as the current operand value.

If “expression” is a function call with the format &(function( )), this function is executed and the result of the function is used as the current parameter value. The function call can contain input parameters.

If “expression” is a name, a variable with the corresponding variable name is first searched for. As with non-S procedures, an entry &(name) is evaluated as follows:

1. A variable with this name is searched for.
2. If no such variable exists, a function with this name is searched for.
3. If neither a job variable nor a function with the specified name exists, then a search for a corresponding job variable will either be made or not, depending on the setting of the JV-REPLACEMENT operand in the SET-PROCEDURE-OPTIONS or MODIFY-PROCEDURE-OPTIONS command. If the value here is set to AFTER-BUILTIN-FUNCTION, a job variable will be sought; if the setting is *NONE, no search is made.
4. If the specified name does not designate a variable, function or job variable, error handling is activated.

The first character to follow the escape character can be:

– an open parenthesis (as separator (entry: &(expression))
– the first character of a variable or function name (entry: &name)

To ensure that & characters in text are retained, instead of acting as escape characters, single escape characters should be replaced by double ones (&&).

If “name” is followed by a period “.”, this period is lost in expression replacement (compatible with non-S procedures).
If “name” is followed by special characters, these characters are interpreted as separators. In contrast to non-S procedures, hyphens in the name are also translated and do not function as separators. (Example: &JOB-CLASS is replaced by the current job class, e.g. JCB00200.)

Commands or data lines may contain any number of expression replacements.

Expression replacements can be nested; however, recursive expression replacement is not possible. If, for example, &(expression) is replaced by the characters A + &B, &B is not evaluated further; the character & next to the B is retained.

Expression replacement cannot be executed before a tag or in a tag.

If an error occurs during expression replacement, error handling is activated, unless otherwise specified. Error handling can be suppressed for data lines by means of the operand DATA-ERROR-HANDLING = *NO in the SET-PROCEDURE-OPTIONS command.

Error handling will also be activated if a procedure line contains an & character by itself.

Restrictions

Expression replacement cannot be used to generate control flow commands. If a control flow command does contain an expression replacement in the command name, it is rejected at the time of procedure execution.

The following characters and names cannot be generated by expression replacement:

- continuation characters at the end of a procedure line
- an escape character that is then to function recursively
- the separator for commands or statements (semicolon “;”)
- branch tags or block names
- commands which cannot be generated by expression replacement:
- SDF-P control flow commands:
  ```
  BEGIN-BLOCK
  BEGIN-PARAMETER-DECLARATION
  CYCLE
  DECLARE-PARAMETER
  ELSE, ELSE-IF
  END-BLOCK, END-FOR, END-IF, END-WHILE
  END-PARAMETER-DECLARATION
  EXIT-BLOCK
  FOR
  GOTO
  IF, IF-BLOCK-ERROR, IF-CMD-ERROR
  INCLUDE-BLOCK
  REPEAT
  UNTIL
  WHILE
  ```
- AID commands that are followed by a command list or subcommand list (see the “AID” manual [6])
- SDF command SET-JOB-STEP (see “Commands, Vol. 1-5” [3])
- OPEN-VARIABLE-CONTAINER in the DECLARE-VARIABLE block before the first DECLARE-PARAMETER command.

3.2.3 Data and statements

In addition to commands, procedure lines can also contain data and statements. In S procedures, data can also be transferred with the SEND-DATA command and statements can be transferred with the SEND-STMT command.

This section deals with the following topics:

– reading in data
– generating an end-of-file condition
– mixing data, statement and command lines

Reading in data

There are various ways of transferring data lines to a program. For example, a file containing the data lines can be opened in the program, or data can be read from SYSDTA.

In order for an input file to be opened and data lines to be read in the program, this file must be assigned to the program. This operates in procedures just as it does on the system level. For a detailed description, see the manual entitled “Introductory Guide to DMS” [1].

The standard path for inputting data is SYSDTA. Like the system files SYSCMD, SYSLST SYSOPT and SYSOUT, SYSDTA belongs to the SYSFILE environment. SYSDTA is a logical file (system file) and designates the path by which data is forwarded from the system to a program.

In S procedures (in contrast to non-S procedures), SYSDTA is assigned by default to SYSCMD. For example, using the ASSIGN-SYSDTA command, SYSDTA can be assigned to a file from which the program then reads the data records sequentially (for information on ASSIGN-SYSDTA, see “Commands, Vol. 1-5” [3]).

If SYSDTA is assigned to a file in a procedure, it is exactly the same as assigning a file on the system level. However, the data lines can also be written directly into the procedure.

If SYSDTA is redirected using the command ASSIGN-SYSDTA TO-FILE=*SYSCMD, the program reads the data lines from the procedure.

In S procedures, the SEND-DATA command can be used to transmit data lines to the program which is loaded. In this case, each command call contains a data record or an expression that yields the data record when evaluated.

Data can also be written to separate data lines without the SEND-DATA command call. A procedure line then contains exactly one input data record. However, you must also make sure that the end-of-file condition is generated (for information on the end-of-file condition, see page 66). Any ISAM key contained in the records is ignored.

Nevertheless, entering data by means of the SEND-DATA command has several advantages:

– the procedure line “containing” the data record can be introduced by a tag
– the procedure line with the data record can contain a comment
– the data record can extend over several continuation lines
– the data and end-of-file condition are generated by means of a standard interface (see the section “Creating the end-of-file condition” on page 66).

Example

```
/ABC = 'Word '
/DEF = 'processing'
/START-EXE PROGRAM1                                     “Load program PROGRAM1”
/SEND-DATA ABC                                          “Input: Word”
/SEND-DATA 'INPUT'                                      “Input: INPUT”
/SEND-DATA 'This is a very long input record that -
/must always contain more than 72 characters'
/SEND-DATA ABC // DEF                                   “Input: Word
/SEND-DATA ...                                          processing”
```

In this example, different data records are read in consecutively. The fourth call of SEND-DATA contains as an argument an expression that yields the input “Word processing” when evaluated.

SEND-DATA can also be called within a loop in which input data records are generated successively.

Example

```
/SET-VARIABLE A = 'Text'
/SET-VARIABLE B = 'verarbeitung'
/SET-VARIABLE C = A // B
/BEGIN-BLOCK PROGRAM-INPUT=*MIXED-WITH-CMD
/START-EXE PROGRAM1                         "Load program PROGRAM1"
/FOR EINGABE = (A,B,C)                      "FOR loop in which the contents"
/                                           "of the variables A, B and C"
/                                           "are assigned sequentially“
/                                           "to the control variable INPUT"
/SEND-DATA EINGABE
/END-FOR
/END-BLOCK
```

Input records can also be written directly into a procedure line, without using the command call /SEND-DATA.

Example

```
/SET-PROCEDURE-OPTIONS DATA-ESCAPE-CHAR = *STD
/VAR1 = 'Word '
/VAR2 = 'processing'
/START-EXE PROGRAM1                                  “Load program PROGRAM1”
&VAR1
'INPUT'
&(VAR1 // VAR2)
 ...
```

In this example, the following data lines are transferred sequentially:

```
Text

'INPUT'

Word processing
```

Mixing data, command and statement lines

For forwarding commands and data lines, SDF-P provides the command SEND-DATA. This allows commands and data lines to be mixed as desired.

SDF-P also offers a separate command for forwarding statements to a program: SEND-STMT. When this command is used, the statement is transferred as an operand value in the command call directly or is transferred as an expression that yields the statement when evaluated.

Mixing of data lines and command lines can also be declared in the BEGIN-BLOCK command, using the operand PROGRAM-INPUT=*MIXED-WITH-CMD. This setting applies to the current command block and to all blocks nested within this block; it cannot be deactivated in a subordinate block.

PROGRAM-INPUT=*MIXED-WITH-CMD has the effect that the system handles data and command lines “analogously”. This means that the command line does not trigger an end-of-file condition. Instead, the system interrupts the program:

– The system cancels the program’s “ready-to-receive state”, which corresponds to interrupting a program using HOLD-PROGRAM.
– The system executes the commands until another data or statement line is read in.
– The system reactivates the program’s “ready-to-receive state”, which corresponds to resuming a program using RESUME-PROGRAM.
– Data or statement lines are then forwarded to the program until another command or a statement is read in or until the end-of-file condition is generated explicitly (for example, using /SEND-DATA *EOF).

Example

```
/BEGIN-BLOCK PROGRAM-INPUT=*MIXED-WITH-CMD
/                                    "DData and command lines can be mixed"
/START-EXE PROGRAM1                  "Start PROGRAM1"
/FOR V = *LIST(L)                    "FOR loop, control variable: V"
/                                    "is passed to PROGRAM1"
&V
/END-FOR
/END-BLOCK
```

In this example, the program PROGRAM1 is started up. As soon as data is requested from SYSDTA, the program is interrupted; the subsequent commands (BEGIN-BLOCK and FOR) are executed.

In the FOR loop, the contents of the first element of the list L are first assigned to the variable V.

The next line (&V) is a data line. The program is resumed and the data line (i.e. the contents of variable V) is transferred to the program.
As soon as data is again requested from SYSDTA, the program is executed. If the list variable L contains additional elements, the loop is resumed and the contents of the next element are assigned to the variable V.
Data input is terminated when the list variable L has been “processed” and the FOR loop has been terminated.

END-BLOCK cancels the PROGRAM-INPUT=*MIXED-WITH-CMD setting with the result that the next command generates the end-of-file condition.

If data, statement and command lines can be mixed, this also applies when calling nested procedures using CALL-PROCEDURE or INCLUDE-PROCEDURE. The program is interrupted and the called procedure is first processed in command mode. Before data is again forwarded to the requesting program, the RESUME-PROGRAM command must be called (the “ready-to-receive state” must be re-enabled). Note, however, that an implicit RESUME-PROGRAM command must not be used here.

Therefore, data procedures must begin with the RESUME-PROGRAM command (in addition, S procedures must begin with a slash!).

The HOLD-PROGRAM can be used to interrupt data input once again in order to allow an explicit or implicit EXIT-PROCEDURE to be executed.

Example

```
BEGIN-BLOCK PROGRAM-INPUT=*MIXED-WITH-CMD
/START-EXE PROGRAM2
/CALL-PROCEDURE DATA.PROC                "Data procedure call"
    -----> /RESUME-PROGRAM               "Program is ready to receive"
              input-1
              input-2
              ..
           /HOLD-PROGRAM                 "Program is interrupted"
           /EXIT-PROCEDURE
    <----
/...                                     "Superordinate procedure is resumed"
```

If the data procedure is not terminated with HOLD-PROGRAM, the end-of-file condition is generated after the last procedure line.

The setting PROGRAM-INPUT=*MIXED-WITH-CMD in the calling procedure does not apply in the data procedure. However, the /HOLD-PROGRAM command call does not yet generate an end-of-file condition. The procedure’s commands are executed, i.e. the procedure is terminated (implicitly if EXIT-PROCEDURE is omitted). In the calling procedure, however, the end-of-file condition is canceled, i.e. the program continues executing after the substituted RESUME-PROGRAM.

If an error occurs when a command is being executed, control is passed in command mode to the next error handling block (IF-BLOCK-ERROR).

Restrictions

The commands of the AID debugging utility must not be mixed with data. Mixing data lines with these commands may lead to unexpected program behavior since the program is resumed after each command (see the “AID” manual [6]).

The error status of statements is not always passed across to the command level. Consequently, it is possible that errors at statement level will not be processed until the end of the program when they are handled at command level by an IF-BLOCK-ERROR block. Setting BEGIN-BLOCK PROGRAM-INPUT=*STD/*MIXED-WITH-CMD has the following effects:

– With PROGRAM-INPUT=*STD, the input of a command (not HOLD-PROGRAM) causes an end-of-file condition for statement inputs, and particular program responses (e.g. for TERM with errors). Errors, if any, are passed to the command level when the program terminates. I.e. they can subsequently be processed further using IF-BLOCK-ERROR.

  Example

```
/BEGIN-BLOCK PROGRAM-INPUT=*STD
/START-EXE MY-UTILITY     "IF EOF DURING RDSTMT: TERMINATE"
//...      "FEHLER"
/IF-BLOCK-ERROR; WRITE-TEXT 'ERROR DUE TO EOF'; END-IF
/...      "ERROR PROCESSING"
/END-BLOCK
```

– PROGRAM-INPUT=*MIXED-WITH-CMD(PROPAGATE-STMT-RC=*STD) interrupts command inputs while program statements (and possible errors) are input. However, no error will be passed to the command level. In this case, IF-BLOCK-ERROR will not process the error at statement level. However, the error can be interrogated using the predefined function STMT-SPINOFF( ); i.e. instead of IF-BLOCK-ERROR, IF-STMT-SPINOFF( ) must be specified.

  Example

```
/BEGIN-BLOCK PROGRAM-INPUT=*MIXED-WITH-CMD
/START-EXE MY-UTILITY     "IF EOF DURING RDSTMT: TERMINATE"
//...      "ERROR"
/IF-BLOCK-ERROR; WRITE-TEXT 'NO TEXT WRITTEN'; END-IF
/...      "NO ERROR PROCESSING AT COMMAND LEVEL"
/END-BLOCK
/ "EOF IS NOW REPORTED, BECAUSE NO FURTHER DATA INPUT TAKES PLACE."
/ "PPROGRAM TERMINATES."
/ "N ERROR IS PASSED TO THE COMMAND LEVEL AFTER BLOCK END"
```

– The operand setting PROGRAM-INPUT=*MIXED-WITH-CMD(PROPAGATE-STMT-RC=*TO-CMD-RC) interrupts the command input while program statements (and possibly errors) are entered. Return codes from program statements are transferred to the command level. In this manner, any errors occurring in the statement level can be handled with the IF-BLOCK-ERROR command and the predefined functions SUBCODE1( ), SUBCODE2( ) and MAINCODE( ).

  Example

```
/BEGIN-BLOCK PROGRAM-INPUT=*MIXED-WITH-CMD -
                           /(PROPAGATE-STMT-RC=*TO-CMD-RC)
/START-LMS
//...                "ERROR AT STATEMENT LEVEL"
//...
/IF-BLOCK-ERROR
/...                 "ERROR PROCESSING AT COMMAND LEVEL"
/END-IF
//END
/END-BLOCK
```

Creating the end-of-file condition

Whether the system will create the end-of-file condition or execute a command depends on the setting of the PROGRAM-INPUT operand in the BEGIN-BLOCK command. For the default setting, BEGIN-BLOCK PROGRAM-INPUT=*STD, the end-of-file condition will be created if a command is called in the data stream. That is to say, data and command lines must not be mixed here - unless SEND-DATA is specified.

Example

```
/PROG: BEGIN-BLOCK
/ABC = 'Word '
/DEF = 'processing'
/START-EXE PROGRAM1            “Load PROGRAM1”
&ABC
INPUT
&(ABC // DEF)
...
/WRITE-TEXT ...
/END-BLOCK PROG
```

In this example, the command WRITE-TEXT terminates the data input. From the initial slash, the system recognizes the line as a command line, and generates the end-of-file condition. After the program has terminated, the command is executed.

If the SEND-DATA command is used with the default setting of BEGIN-BLOCK to pass data to the program then - in order to generate the end-of-file condition - the operand RECORD = *EOF must be specified in the last SEND-DATA command.

Example

```
 /PROG: BEGIN-BLOCK
 /VAR1 = 'Word '
 /VAR2 = 'processing'
 /START-EXE PROGRAM1              “Load PROGRAM1”
 /SEND-DATA VAR1                  “Input: Word “
 /SEND-DATA 'INPUT'               “Input: INPUT”
 /SEND-DATA VAR1 // VAR2          “Input: Word processing”
 /SEND-DATA RECORD=*EOF           “End of input: EOF”
```

The program can read the data records in sequentially.

Example

```
/PROG: BEGIN-BLOCK
/SET-VARIABLE A = 'Word '
/SET-VARIABLE B = 'processing'
/SET-VARIABLE C = A // B
/BEGIN-BLOCK PROGRAM-INPUT=
              *MIXED-WITH-CMD
/START-EXE PROGRAM1                 “Load PROGRAM1”
/FOR INPUT = (A,B,C)                “FOR loop in which the contents of ”
/                                   “variables A, B, C are assigned in ”
/                                   “turn to the control variable”
/SEND-DATA INPUT                    “INPUT”
/END-FOR
/SEND-DATA *EOF                     “End of input: EOF”
/END-BLOCK
```

As described above, specifying BEGIN-BLOCK PROGRAM-INPUT=*MIXED-WITH-CMD allows data and command lines to be mixed without additionally specifying a SEND-DATA command. However, it is still necessary to specify SEND-DATA=*EOF to generate the end-of-file condition.

Example

One command block is nested within another. In the superordinate block, PROGRAM-INPUT=*MIXED-WITH-CMD is set; this also affects the subordinate block.

```
/BEGIN-BLOCK PROGRAM-INPUT=*MIXED-WITH-CMD
/...

/COMP: BEGIN-BLOCK
/START-EXE $RZ.FOR1
   PROGRAM STATIST
   ...
   END
/SEND-DATA *EOF
/START-EXE $BINDER
...
/END-BLOCK COMP
...
/END-BLOCK
```

In the subordinate block, the FORTRAN compiler FOR1 reads in a FORTRAN program line by line until the system detects a command call.

The program is interrupted and the command is executed. The end-of-file condition is generated, thus activating end-of-file handling in the compiler. The FORTRAN program is compiled.
The static linkage editor BINDER is then called. The compiler FOR1 is unloaded and BINDER is loaded and started. Statements to the BINDER then follow.

If the /SEND-DATA *EOF command call were to be omitted in this example, the following would occur:

– Data input would be interrupted by the first command, without an end-of-file condition starting the compiler, because in this case PROGRAM-INPUT=*MIXED-WITH-CMD was specified.
– This command (START-EXE $BINDER) would be executed, i.e. the compiler FOR1 would be unloaded (without first writing the results of the compilation) and the BINDER program would be loaded.

Since no end-of-file condition was generated, the program was not compiled. Consequently, BINDER cannot access a newly compiled program.

3.3 Error handling

In contrast to non-S procedures, S procedures permit selective error handling for commands and program statements. Since SDF-P interprets return codes from statements as return codes from commands, no distinction is made between the two types of return codes in the following sections (see also the description of the BEGIN-BLOCK command). SDF-P provides commands for this purpose that react to an error situation or can be used to analyze an error. In addition, there are functions that support error handling (see chapter “Functions” on page 223).

Error handling in S procedures is block-oriented, i.e. it is performed on block level. In the case of blocks nested inside one another, error handling can be performed on each level. However, it can be called for higher, surrounding blocks only. The errors that occur in inside blocks are then also transferred and processed.

The error handling of individual commands is a special case in block-oriented error handling. In this case, the command is set internally in a BEGIN block.

Error handling itself is performed in error handling blocks. A distinction must be made between two such error handling blocks:

– IF-BLOCK-ERROR
– IF-CMD-ERROR

SDF-P also supports error analysis on the command return code level (with the SAVE-RETURNCODE command and predefined functions).

For the sake of compatibility with non-S procedures, SDF-P still supports the SET-JOB-STEP command.

Note
The following commands are not executed if an error occurs:
ABEND, ABORT, CANCEL-PROCEDURE, END-PROCEDURE, ENDP, EXIT-JOB, EXIT-PROCEDURE, LOGOFF.

When an error occurs, the components of the return code can be queried using the built-in functions (see chapter “Functions” on page 223). If the return code is to be checked even when an error has not occurred, the SAVE-RETURNCODE command must first be called. Only then will the return code be made available for evaluation by means of the predefined functions.

Error handling is automatically activated when command execution returns a return code with an error indication. SDF-P then branches to the next IF–BLOCK-ERROR or IF–CMD-ERROR command.

SAVE-RETURNCODE

The SAVE-RETURNCODE command is used to save the current return code. The return code can then be evaluated by means of the predefined functions. Thus, for example, the programmer can react to situations in which command execution was terminated without error (subcode1 = 0) but subcode2 and the maincode must be evaluated.

3.3.1 Error handling blocks

The commands for error handling initiate IF blocks that are processed like “normal” IF blocks. ELSE and END-IF commands are also associated with these IF blocks.

IF-BLOCK-ERROR block

If a command return code with an error flag is returned, SDF-P automatically branches to the next IF-BLOCK-ERROR block; other blocks on lower nesting levels are skipped.

The IF-BLOCK-ERROR command can also be called if an error has not occurred. In this case, the ELSE branch of the IF-BLOCK-ERROR block is executed.

If no IF-BLOCK-ERROR block is present between the command that led to the error and the procedure end, the procedure is terminated and the caller is notified of the error. The error code is transferred to the caller in the same way as for procedure termination with EXIT-PROCEDURE ERROR = *YES.

Example

```
/BEGIN-BLOCK
/"commands"
/  IF-BLOCK-ERROR
/  "Error handling
/  ELSE
/  "No error"
/  END-IF
/END-BLOCK
```

IF-CMD-ERROR block

The IF-CMD-ERROR block can be used to perform error handling for the command that directly precedes it but not for block initiation or termination commands. This means that IF-CMD-ERROR cannot be applied to the FOR, IF, REPEAT or WHILE commands or the associated termination commands.

This permits specific error handling for this command and prevents execution of block error handling.

Like the IF-BLOCK-ERROR block, the IF-CMD-ERROR block can contain an ELSE branch that is executed if the IF-CMD-ERROR command is called and the preceding command was executed without error.

Example

```
/command
/IF-CMD-ERROR
/  "Error handling"
/ELSE
/  "No error in command sequence"
/END-IF
```

Regardless of whether or not an error occurred, the command return code of the command which preceded IF-CMD-ERROR is always available. This permits warnings to be evaluated, even if there was no error, both in the ELSE branch and after command error handling (END-IF).

3.3.2 Command return codes

Error handling in S procedures is based on the fact that commands and statements supply a defined return code. Using this return code, SDF-P can determine whether an error occurred during the execution of a command and what that error was.

The return code from a command always comprises three components:

Subcode1	Error class
Subcode2	Additional information on subcode1
Maincode	Error/message code

For more information on the return codes delivered by SDF-P commands, see section “Command return codes” on page 559.

The components of the return code can be queried by means of the predefined functions: SUBCODE2( ), SUBCODE1( ) and MAINCODE( ); the error message text can be queried using the predefined function MSG( ) (see chapter “Predefined functions” on page 347).

If the return code is to be checked even if no error has occurred, the SAVE-RETURNCODE command must first be called (except where IF-CMD-ERROR was specified). This makes the return code available so that it can be evaluated by means of the predefined functions.

Error handling is automatically activated when a command is executed and returns a return code with an error flag. SDF-P then branches to the next IF-BLOCK-ERROR or IF-CMD-ERROR command.

SAVE-RETURNCODE

The SAVE-RETURNCODE command serves to save the current return code so that it can be evaluated with the predefined functions. Thus, for example, the programmer can react to situations in which command execution was terminated without error (subcode1 = 0) but subcode2 and maincode must be evaluated.

Example

As a result of the command sequence below, the return code of the preceding command in a procedure can be evaluated if no error occurred.

```
/"Command which returns a warning but no error"
/SAVE-RETURNCODE
/WRITE-TEXT &(TO-C-LIT(MSG(MAINCODE())))
```

Error handling blocks

Error handling in structured procedures is block-oriented, i.e. it is applied to command blocks.

Error handling itself also takes place in blocks, called error-handling blocks. These blocks are described in detail in section “Creating the procedure body” on page 92.

SET-JOB-STEP command

For the sake of compatibility with non-S procedures, SDF-P supports the SET-JOB-STEP (or STEP) command; however, this command should not be used in S procedures. It has the same effect as an empty error handling block (/IF-BLOCK-ERROR; END-IF), but in addition it resets some of the task switches. There are also some restrictions with SET-JOB-STEP which must be noted (see chapter “Converting non-S procedures” on page 287).

3.3.3 Error condition when reading in data lines

The following errors can occur when data lines are read in:

– A procedure line contains data where a command was expected.
– The expression cannot be replaced by a variable or function.
– A procedure line contains a single escape character by itself.

If one of these errors occurs, error handling is normally initiated.

The SET-PROCEDURE-OPTIONS and MODIFY-PROCEDURE-OPTIONS commands provide the DATA-ERROR-HANDLING operand, which can be used to specify that error handling should not be activated (for further details see section “Activation of error handling” on page 84).

3.3.4 Error messages

If an error occurs during procedure execution, error messages identifying the error are output. More information can be requested in interactive mode with the aid of the HELP-MSG-INFORMATION command.

For a summary of the error messages that are output by SDF-P, see chapter “Messages” on page 797.

3.3.5 Error transfer

“Error transfer” refers to the fact that information on errors that occur is transferred from a subordinate procedure to the superordinate procedure.

The way that this is done differs, depending on whether the procedures are called in the foreground or in the background. For details, see chapter “Calling and controlling procedures” on page 105.

3.4 Procedure compiler

SDF-P provides a procedure compiler. This compiler is responsible for ensuring the portability of S procedures on systems in which the SDF-P subsystem is not loaded. This means that the procedure compiler converts S procedures into an intermediate format, thus allowing these alone to be executed by means of SDF-P-BASYS.

The following points are important in this context:

The procedure compiler only converts (compiles) the SDF-P control structures. This means that while control structures are converted in the intermediate format generated by the compiler (also referred to as a compiled procedure or object procedure), other commands and statements are identical to the source procedure. They are executed in exactly the same way as in the original S procedure.

Compilation is performed with the COMPILE-PROCEDURE command, which is part of the (chargeable) SDF-P subsystem. The final product of the compilation process is the intermediate format (compiled procedure) mentioned earlier, which can be stored in a file or library element. The compiled procedure can be called like an S procedure using the command CALL-PROCEDURE, INCLUDE-PROCEDURE or ENTER-PROCEDURE. However, unlike in the first two of these commands, no element type can be specified in the command ENTER-PROCEDURE FROM-FILE=*LIBRARY-ELEMENT(..). SDF-P-BASYS in this case expects the compiled procedure to be stored as a SYSJ-type or J-type element.

Notes

Conversion using COMPILE-PROCEDURE and execution on installations without the SDF-P subsystem is also an option for S procedures which do not contain any chargeable SDF-P functions, or which only include a few such functions.

Appropriate use of compiled procedures

Compiled procedures should only be used in installations in which the chargeable part of SDF-P has not been loaded. On installations on which the chargeable part is available and has been loaded and in which S procedures are updated regularly, the use of compiled procedures is not recommended.

The fact that only the SDF-P control structures are compiled and the major part of the S procedure is not interpreted until runtime results in the following typical applications:

- Procedures with chargeable SDF-P functions can be passed to a user who has not installed the chargeable part of SDF-P; e.g. an external software developer.
- If a computer center with multiple BS2000 computers possesses a software licence for the chargeable part of SDF-P for one computer only:
  S procedures can be developed, tested and compiled on this computer and the compiled procedures can be used on the other computers even though the chargeable part of SDF-P is not present.
- If a computer center only possesses a software licence for the chargeable part of SDF-P as part of a current project, e.g. only during the development of certain procedures:
  the required S procedures can be developed, tested and compiled during this period and the compiled S procedures can be used even after the licence for the chargeable part of SDF-P has expired.

3.5 SDF-P commands in interactive mode

Not only can SDF-P commands be called from a procedure environment, but they can also be entered interactively at the data display terminal.

Generally speaking, it is always practical to use SDF-P interactively in situations where a job occurs in this exact form only once or only very seldom. Jobs that occur again and again should be executed in procedures.

The following are examples of practical uses of SDF-P commands in interactive mode:

- Checking the syntax of command calls:
  For example, when generating a procedure; in this way, syntax errors can be avoided or the correction of syntax errors can be checked before the next test run.
- Before a procedure call, prefilling variables that are then used within the called procedure:
  For example, variables that are passed as procedure parameters.
- Programming loops interactively:
  For example, when several files are to be processed consecutively.
- Starting up long program runs in a dialog block in which the program run is monitored:
  For example, long compiler runs in program test phases; the results are compiled and evaluated automatically; the programmer can use this time for other jobs and does not have to remain seated at the data display station.

3.5.1 Rules for entering SDF-P commands interactively

Input line structure

In general, the same syntax rules apply to commands as when they are used in a procedure, with the exception of the initial slash and the end-of-line character.

The initial slash at the beginning of the input line can be omitted.

This applies both when entering individual SDF-P commands and in command blocks.

End of line

Input lines can be terminated with the $\boxed{\text{DUE}}$ (send) key or with a logical end of line ($\boxed{\text{LZE}}$ key). The following generally applies: as soon as the $\boxed{\text{DUE}}$ key is pressed, input is considered to be terminated.

The way in which an input line must be terminated depends on various factors:

– If input consists of a single line only, it is terminated with $\boxed{\text{DUE}}$.
The command is analyzed and immediately executed.
– If several associated input lines are to be entered together but the commands are not embedded in a command block, each input line must be terminated with a *logical end of line* character ($\boxed{\text{LZE}}$ key). The $\boxed{\text{DUE}}$ key must not be pressed until the last input line has been terminated.
– If several associated input lines are to be entered together and the commands are embedded in a command block, each line can be terminated with $\boxed{\text{DUE}}$.
As each line is entered, its contents are then preanalyzed and buffered but not yet executed. As soon as an error is detected (for example, an incorrect sequence of control flow commands), the block is canceled. The entire command block is not executed until the command block has been terminated with the block termination command (and $\boxed{\text{DUE}}$).
If command blocks are entered interactively, the block identifier appears in the operating system prompt. The normal operating system prompt does not reappear until the block is terminated or canceled due to an error.

Data and statements

In order for it to be possible within command blocks to differentiate between data and commands, data must be entered as arguments of the SEND-DATA command (see page 732).

If a line is to contain a statement, it must begin with two slashes “//” or it must be entered as the argument of a SEND-STMT command.

Before the program is called, /ASSIGN-SYSDTA *SYSCMD must be set so that the program reads the data or statements from the dialog block.

Expression replacement

Expression replacement is possible in both guided and unguided dialog. However, actual replacement does not occur until the command is executed.

In contrast to & replacement in S procedures, the expression to be replaced is left unchanged if an error is detected during evaluation.

Example

```
/WRITE-TEXT '&(1 // 2)'
```

The expression 1 // 2 is invalid because numbers must not be concatenated. In an S procedure, error handling is initiated in this case; in a dialog, the character string &(1 // 2) is output unchanged.

Job variable replacement is also active by default in the dialog.

Interrupt/cancel

The K1 or K2 key can be used to cancel input within a command block at any time. The normal operating system prompt then reappears.

Command block execution can be interrupted at any time with the K2 key and resumed with RESUME-PROCEDURE.

Guided/unguided dialog

When SDF-P commands are entered individually in interactive mode, guided dialog can be used.

Commands can be entered in command blocks in unguided dialog only; even the “command?” entry is buffered. Guided dialog is not initiated until command execution. Preset entries are then taken into account.

Guided dialog is active in dialog blocks only if it has been switched on with the command MODIFY-SDF-OPTIONS PROCEDURE-DIALOGUE=*YES.

Expression replacement can be used in guided dialog.

3.5.2 Example

3.5.2.1 Compiling and linking a program

The COBOL.PROG program is to be compiled. It is a very complex program and compilation takes a relatively long time. The programmer would like to use this “waiting time” for other activities and starts compiling in an SDF-P command block in interactive mode (the programmer does not write a procedure because it is certain that the program no longer contains any errors and, therefore, this exact command sequence will not have to be entered again).

The programmer enters the following commands interactively:

```
/BEGIN-BLOCK
/ASSIGN-SYSDTA *SYSCMD
/START-EXE $COBOL85, MONJV = MJV
/SEND-DATA '*COMOPT ...'
/SEND-DATA '*END'
/IF (SUBSTR(JV('MJV'), 1, 2) EQ '$T')
/START-LMS
//...
//...
//END
/ELSE; PRINT-DOC COMPIL.LIST
/END-IF
/END-BLOCK
```

Since the entire command sequence is embedded in a BEGIN block, each line can be terminated with DUE.

Program compilation is to be monitored by the monitor JV MJV (MONJV=MJV).

Following compilation, the contents of the monitor JV are to be evaluated and checked in the IF request. If the monitor JV contains $T, compilation was error-free and the program can be imported to the program library: the LMS utility is called.

If an error occurred during compilation (contents of MJV do not equal $T), the compilation log must be printed out.

The END-IF and END-BLOCK commands terminate the two command blocks nested one inside the other. Following END-BLOCK, command block execution is activated.

3.6 Calling command sequences from a program

The INCLUDE-CMD command provides a means of calling command sequences or a procedure from within a program. The commands to be executed are transferred as the value of the CMD operand. INCLUDE-CMD thus provides the same functionality as the INCLUDE-PROCEDURE command (see page 686), albeit with the following restrictions:

- The commands to be executed are not read from a file but are specified directly in the CMD operand, which assumes the role of a virtual SYSCMD file. In conjunction with the CMD macro call, it is therefore possible to make a command procedure available in main memory.
- INCLUDE-CMD does not support any of the operands of the INCLUDE-PROCEDURE command. Only the default values are used.
- Unlike INCLUDE-PROCEDURE, INCLUDE-CMD does not terminate a program when it is called by the CMD macro. The program is also not terminated if a procedure is called in the command sequence transferred in the CMD operand (see the example on page 685).
- INCLUDE-CMD may be executed in the CMD macro (TU program) and in EXECUTE-SYSTEM-CMD statements.

The system rejects the following operations during execution of the INCLUDE-CMD command to avoid possible inconsistencies since this command is called in program mode:

- Start and terminate program: START utility, LOAD-/START-EXECUTABLE-PROGRAM (or LOAD-/START-PROGRAM), RESTART-PROGRAM (see CALL-PROCEDURE ...UNLOAD-ALLOWED=*NO).
- Resume program: AID commands, RESUME-PROGRAM, EXIT-PROCEDURE RESUME-PROGRAM=*YES, ENDP-RESUME, INFORM-PROGRAM (and SEND-MSG ...,TO=*PROGRAM).
- Abort procedure: CANCEL-PROCEDURE, K2 when prompted for parameters.
- Call INCLUDE-CMD recursively.
- BEGIN-BLOCK PROGRAM-INPUT=*MIXED-WITH-CMD
- SET-JOB-STEP if a program is loaded.

4 Creating S procedures

This chapter describes how the procedure head, procedure body and branches are correctly defined in the course of creating S procedures.

4.1 Creating the procedure head

The procedure head consists of the SET-PROCEDURE-OPTIONS command, which is used to define the attributes of the procedure, and the DECLARE-PARAMETER block, in which the procedure parameters are declared.

The procedure head is always located at the beginning of the procedure. When a procedure call is issued, the commands contained in the procedure head are executed first, thus setting the procedure attributes before the commands in the procedure body are analyzed and processed (see chapter “Calling and controlling procedures” on page 105).

The following paragraphs describe how the procedure parameters are set and which standard attributes can be used. This is followed by an explanation of how procedure parameters are declared.

4.1.1 Setting procedure attributes

The procedure attributes are first set in the procedure head, using the SET-PROCEDURE-OPTIONS command. If different attributes are to apply when the procedure is executed, they may then be modified with the MODIFY-PROCEDURE-OPTIONS command (this does not apply to the SYSFILE environment).

The SET-PROCEDURE-OPTIONS command must be the first command in the procedure head. If SET-PROCEDURE-OPTIONS is used with no operands, then the default values set in the current syntax file will apply. If the default values described in the manual for the procedure attributes are to be used, then SET-PROCEDURE-OPTIONS must be omitted.

The SET-PROCEDURE-OPTIONS command can be used to set the following attributes:

– which command can be used to call the procedure
– whether a separate SYSFILE environment has to be set up
– the length in which procedure records are evaluated
– whether logging occurs and what can be logged
– whether the procedure can be interrupted
– whether error handling is activated when data is read in instead of a command
– what character is to serve as an escape character in data records
– whether the implicit declaration of variables is allowed
– whether job variables are taken into account in expression replacement
– whether the error handling is compatible with the spin-off response or is to be activated depending on the return code
– whether certain SDF-P messages are to be suppressed

Each of these procedure attributes corresponds to an operand in the SET-PROCEDURE-OPTIONS (or MODIFY-PROCEDURE-OPTIONS) command. The sections below provide a short explanation of the meaning of these attributes and the function of the corresponding attributes.

4.1.1.1 Defining the procedure call command

There are three SDF-P commands which can be used to call S procedures: CALL-PROCEDURE, INCLUDE-PROCEDURE and ENTER-PROCEDURE. The command for calling the procedure is defined in the CALLER operand of the SET-PROCEDURE-OPTIONS command. The operand values which can be specified for CALLER are *ANY, *CALL and * INCLUDE.

– If *ANY is set, any of the calling commands may be used to call the procedure. *ANY is the default setting.
– If *CALL is set, the procedure may be called by either CALL-PROCEDURE or ENTER-PROCEDURE.
– If *INCLUDE is set, the procedure may only be called by INCLUDE-PROCEDURE.

Further details about calling procedures will be found in chapter “Calling and controlling procedures” on page 105.

4.1.1.2 Defining the SYSFILE environment

The SYSTEM-FILE-CONTEXT operand of the SET-PROCEDURE-OPTIONS command is used to set the system file environment in which the procedure is to be executed.

The term SYSFILE environment refers to the system files of the BS2000, regardless of whether these files have their primary allocation or are assigned to a BS2000 file.

Procedures can be executed in a separate SYSFILE environment or can adopt the system file environment of the caller. This choice is set by the SYSTEM-FILE-CONTEXT operand.

The operand values which can be specified for SYSTEM-FILE-CONTEXT are *STD, *OWN and *SAME-AS-CALLER.

- If *STD is set, the procedure will be allocated a separate SYSFILE environment in which the caller's assignments are adopted for all system files except SYSDTA. In the default configuration, SYSDTA is assigned to the procedure (i.e. *SYSCMD).
- If *SAME-AS-CALLER is set, the procedure adopts the caller's system file environment, including the SYSDTA assignment. If assignments are modified within the procedure, this has a corresponding effect on the caller's system file environment.
- If *OWN is set, a separate system file environment is set up for the procedure; in this, all the caller's assignments are adopted, including that for SYSDTA. However, if assignments are changed within the procedure, this does not affect the caller's system file environment.

The system file environment which is set applies for the entire procedure, and cannot be modified by MODIFY-PROCEDURE-OPTIONS.

4.1.1.3 Defining the length of procedure lines

The INPUT-FORMAT operand of the SET-PROCEDURE-OPTIONS command is used to specify where in the procedure line the continuation character must be placed if the command or statement sequence extends over several lines.

The operand values which can be specified for INPUT-FORMAT are *FREE-RECORD-LENGTH and *BY-SDF-OPTION.

If *FREE-RECORD-LENGTH is set, the continuation character may appear in any position in the procedure line. However, it must be the last relevant character in the line. *FREE-RECORD-LENGTH is the default setting.

If *BY-SDF-OPTION is set, the positioning of the continuation character depends on the setting of the CONTINUATION operand in the MODIFY-SDF-OPTIONS command. If CONTINUATION =*NEW-MODE is specified, the continuation character may be anywhere between columns 2 and 72. If CONTINUATION = *OLD-MODE is specified, the continuation character must be in column 72.

The setting *BY-SDF-OPTION ensures the compatibility of non-S procedures with S procedures, thus facilitating conversion.

The value set for the INPUT-FORMAT operand applies for the entire procedure, and cannot be modified by MODIFY-PROCEDURE-OPTIONS.

4.1.1.4 Setting the logging

The LOGGING-ALLOWED operand of the SET-PROCEDURE-OPTIONS command has two functions: to specify whether the procedure may be logged, and to specify whether commands and/or data should be logged.

The operand values which can be specified for LOGGING-ALLOWED are *YES, *NO and *PARAMETERS.

– If LOGGING-ALLOWED=*YES is set, both commands and data may be logged.
– If the setting is *NO, no logging is permitted.
– If *PARAMETERS is set, the logging of commands and data can be separately controlled. *PARAMETERS is the default setting.

The settings for the logging parameter can be changed in the MODIFY-PROCEDURE-OPTIONS command.

Logging is activated by the appropriate specification for the LOGGING operand in the calling command.

4.1.1.5 Defining the interruptibility of a procedure

The INTERRUPT-ALLOWED operand of the SET-PROCEDURE-OPTIONS command is used to specify whether procedure execution can be interrupted by the $\boxed{K2}$ function key.

The operand values which can be set for INTERRUPT-ALLOWED are *YES and *NO.

– If *YES is set, the procedure may be interrupted. Such procedures are described as interruptible. They can be continued after an interruption using the RESUME-PROCEDURE command. *YES is the default setting.
– If *NO is set, the procedure is uninterruptible. The $\boxed{K2}$ key has no effect.

The effects of a procedure interruption are described in section “Procedure interruption” on page 125.

4.1.1.6 Activation of error handling

The DATA-ERROR-HANDLING operand is used to specify whether error handling in data records should be activated if an error situation occurs.

The default setting is *YES, which means that error handling will be activated if a procedure reads data where commands are expected, or if expression replacement in data cannot be resolved.

If the setting is *NO, no error handling will be activated in such cases. The setting *NO is compatible with the spin-off behavior of non-S procedures.This facilitates the conversion of non-S procedures into S procedures.

4.1.1.7 Setting the type of error handling

Error handling can be activated either in a way that is compatible with the previous spin-off behavior, or on the basis of the command return codes. It is performed by error handling blocks defined in the procedure body.

The ERROR-MECHANISM operand of the SET-PROCEDURE-OPTIONS command can be used to set the applicable type of error handling. The possible operand values are *SPIN-OFF-COMPATIBLE and *BY-RETURNCODE.

– If *SPIN-OFF-COMPATIBLE is set, error handling will be activated in a manner compatible with the spin-off behavior. Subcode1 will be ignored. This ensures that the error behavior of S procedures which were created under BS2000 V10.0 remains compatible. *SPIN-OFF-COMPATIBLE is the default setting.

– If the setting is *BY-RETURNCODE, error handling will be activated if Subcode1 of the last command return code is not zero. No account is taken of the spin-off behavior. The setting *BY-RETURNCODE is appropriate for procedures in which commands are used that have their own command return codes. This enables differentiated programming to be used.

Example 1 (for ERROR-MECHANISM=*SPIN-OFF-COMPATIBLE)

```
/SET-PROCEDURE-OPTIONS ERROR-MECHANISM=*SPIN-OFF-COMPATIBLE
/...
/PRINT-FILE DOES-NOT-EXIST
/...
/
/IF-BLOCK-ERROR
/                  “IF-BLOCK-ERROR is not initiated, since /PRINT-FILE ”
/                  “(with correct syntax) does not cause a spin-off”
/END-IF
/...
```

Example 2 (for ERROR-MECHANISM=*BY-RETURNCODE)

```
/SET-PROCEDURE-OPTIONS ERROR-MECHANISM=*BY-RETURNCODE
/...
/PRINT-FILE DOES-NOT-EXIST
/...
/IF-BLOCK-ERROR
/                  “IF-BLOCK-ERROR is initiated in accordance”
/                  “with the return codes for /PRINT-FILE”
/END-IF
/...
```

Error behavior when changing versions

Although the behavior of most commands is related to the current version, conversion of a command to defined command return codes does not depend on the BS2000 version, but rather on the version of the system product or subsystem.

If ERROR-MECHANISM=*BY-RETURNCODE is specified, the conversion of a system product to command return codes may affect the error behavior of the related commands in the same BS2000 version.

Error behavior of programs

Error handling also affects user programs; such programs can return their own return codes. For programs which do not return their own command return codes, SDF simulates the command return code in the same way as for spin-off handling.

By default, error handling by SDF-P does not take place until after program termination. The default setting *SPIN-OFF-C0MPATIBLE causes error handling to be performed in accordance with the UNIT=STEP parameter of the TERM macro which triggers spin-off, rather than on the basis of the command return codes. (Refer to the manual “Executive Macros” [7] for more details about the TERM macro, and to the description of the CMDRC macro in the manual “SDF-A” [16] for more information about command return codes.)

Setting PROPAGATE-STMT-RC=*TO CMD-RC in the BEGIN block (see the BEGIN-BLOCK command) permits error handling and evaluation of return codes by SDF-P to be performed during program execution. Error handling is in this case based exclusively on the return codes; the setting for EROR-MECHANISM is irrelevant.

4.1.1.8 Defining the escape character in data records

The DATA-ESCAPE-CHARACTER operand of the SET-PROCEDURE-OPTIONS command is used to specify which character is used as an escape character for expression replacement in data records. The operand values which DATA-ESCAPE-CHARACTER may take are *NONE, *STD, &, #, *, @ and $.

– *NONE is the default setting, i.e. there is no escape character. (However, an escape character **must** normally be declared for utility routines without an SDF interface if replacement within data records is to be carried out.)

– If the setting is *STD, the & character is used as the escape character.

– By specifying &, #, *, @ or $, the corresponding character can be defined as the escape character.

For further details see section “Expression replacement” on page 55.

4.1.1.9 Setting implicit declaration of variables

The IMPLICIT-DECLARATION operand of the SET-PROCEDURE-OPTIONS command is used to specify whether simple variables can be declared implicitly. The default setting is IMPILICIT-DECLARATION=*YES. This means that simple variables can be implicitly declared when they are first assigned, without having to be explicitly declared with a DECLARE-VARIABLE command.

If the setting is IMPLICIT-DECLARATION=*NO, simple variables must be explicitly declared.

Complex variables must always be explicitly declared. Further details on this will be found in chapter “Using variables in S procedures” on page 135.

4.1.1.10 Setting job variable replacement

The JV-REPLACEMENT operand of the SET-PROCEDURE-OPTIONS command is used to specify whether job variables are also to be replaced during expression replacement. The possible operand values for JV-REPLACEMENT are *NONE and *AFTER-BUILTIN-FUNCTION.

– The default setting is *NONE. This means that names are not to be interpreted as job variable names during expression replacement.

– If the setting is *AFTER-BUILTIN-FUNCTION, then names will be interpreted as job variable names during expression replacement.

The settings can be altered by a MODIFY-PROCEDURE-OPTIONS command.

4.1.1.11 Suppressing specific SDF-P messages

In the SUPPRESS-SDP-MSG operand of the SET-PROCEDURE-OPTIONS command you specify if the output of certain SDF-P messages (from the SDP message class) are to be suppressed. The settings are only valid in the calling procedure (and are not inherited). SUPPRESS-SDP-MSG provides the *NONE and <structured-name 7..7> operand values:

– The default setting is *NONE. This means that the message output is not suppressed and all SDF-P messages will be output.

– When a message code from the SDP message class is explicitly specified, the output of this message is suppressed. More than one message code can be specified in a list.

The settings can be changed in the MODIFY-PROCEDURE-OPTIONS command:

– SUPPRESS-SDP-MSG=*NONE switches off the suppression of messages and all SDF-P messages are output again.
– With SUPPRESS-SDP-MSG=*ADD(...) you can add one or more messages to the list of messages to be suppressed.
– With SUPPRESS-SDP-MSG=*REMOVE(...) you can remove one or more messages from the list of messages to be suppressed.

4.1.2 Adopting the default values for procedure attributes

There are two different sets of default values:

– global default values, which are set by the syntax file
– specific SDF-P default values (see the description of the SET-PROCEDURE-OPTIONS command, page 734).

If the default values from the syntax files are to be adopted for a procedure, the procedure head must begin with a SET-PROCEDURE-OPTIONS command with no operand values. This is followed by any required procedure parameter declarations.

If the SDF-P-specific default values are to be adopted for the procedure, the SET-PROCEDURE-OPTIONS command must be omitted. If procedure parameters are to be declared, the procedure head then begins with the declaration of these parameters. If no procedure parameters are to be declared, the procedure begins with the first command of the procedure body. The procedure head is then considered to be implicitly declared.

SDF V4.1 permits task-specific default values to be defined. Although these default values may be defined in S procedures, they take effect with interactive input only. They are ignored in procedure mode; in this mode, the default values set in the syntax files apply. Task-specific default values must not be specified in control flow commands nor in the procedure head. The manual “SDF Introductory Guide” [20] contains more information about task-specific default values.

4.1.3 Declaring the procedure parameters

To enable parameters to be passed to a procedure when it is called, these parameters must be declared in the called procedure.

Procedure parameters are declared in the DECLARE-PARAMETER block by a DECLARE-PARAMETER command. If it is possible to declare all the procedure parameters within one command, the DECLARE-PARAMETER block may consist solely of a DECLARE-PARAMETER command.

If the parameter declarations require several DECLARE-PARAMETER commands, the DECLARE-PARAMETER block must be introduced by a BEGIN-PARAMETER-DECLARATION command and terminated by an END-PARAMETER-DECLARATION command. Between these two commands there may be any required number of DECLARE-PARAMETER commands

Apart from procedure parameters, variable containers can be opened within DECLARE-PARAMETER blocks by an OPEN-VARIABLE-CONTAINER command, making available within the DECLARE-PARAMETER block variables for use in setting the initial values for procedure parameters. The OPEN-VARIABLE-CONTAINER commands must appear before the first DECLARE-PARAMETER command.

The following attributes of the procedure parameters can be set in the DECLARE-PARAMETER command:

– parameter name
– initial value (if specified)
– data type
– type of parameter transfer

Each of these attributes is defined by means of an operand of the DECLARE-PARAMETER command.

4.1.3.1 Defining the parameter name

The parameter name is defined with the NAME operand. Since procedure parameters are classed as variables, the same rules apply to parameter names as to variable names, i.e. either simple names or complex names may be used (see section “Variable names” on page 150).

The parameter name can then be used in the procedure call as a keyword for parameter transfer.

4.1.3.2 Specifying the data type

The data type is defined in the TYPE operand of the DECLARE-PARAMETER command. The same data types apply for procedure parameters as for variables: *ANY, *STRING, *INTEGER and *BOOLEAN. The data type that is set here must be taken into account when passing parameters and when assigning an initial value.
If TYPE = *ANY is specified, then the parameter value will be interpreted as *STRING, regardless of whether it is specified with or without apostrophes in the call.

4.1.3.3 Assigning an initial value

The INITIAL-VALUE operand of the DECLARE-PARAMETER command specifies whether an initial value is to be assigned to the procedure parameter. The possible operand values for INITIAL-VALUE are *NONE, *PROMPT and the direct specification of an initial value.

– *NONE is the default setting. It means that no initial value is assigned to the parameter. However, a value must be assigned to the parameter when the procedure is called.

– If the setting is *PROMPT, a value will be requested when the procedure is executed, unless one is passed with the procedure call. However, it can also be specifically requested via READ-VARIABLE.

Example

```
/SET-PROCEDURE-OPTIONS
/DECLARE-VARIABLE NAME(INITIAL-VALUE=*PROMPT)
/...
/IF (NOT IS-INITIALIZED('NAME'))
/ WRITE-TEXT 'PLEASE ENTER YOUR NAME HERE!'
/ READ-VARIABLE NAME,INPUT=*TERMINAL
/END-IF
```

The third possibility is to use INITIAL-VALUE to directly specify an initial value. The data type of the value specified must match that of the procedure parameter. The initial value will apply unless another value is passed with the call.

4.1.3.4 Specifying the type of parameter transfer

When a procedure is called, the parameter argument which is passed can be either a direct value or the name of a variable which contains the appropriate value. For this reason, the way in which this argument is to be interpreted must be defined in the called procedure as part of the parameter declaration. This is done in the TRANSFER-TYPE operand of the SET-PROCEDURE-OPTIONS command.

- If TRANSFER-TYPE =*BY-VALUE is set, the value passed in the procedure call is directly assigned to the procedure parameter; if the procedure parameter had already been initialized with another value, this previous value is overwritten. TRANSFER-TYPE= *BY-VALUE is the default setting. *BY-VALUE must be set to enable parameters to be passed for a procedure called in the background.
- If TRANSFER-TYPE = *BY-REFERENCE is set, the value passed is interpreted as the name of a variable which the caller has declared and initialized. This variable then serves as a container for the procedure parameter.

The effects of parameter transfer during the procedure call are described in section “Passing procedure parameters” on page 106.

4.1.3.5 Initializing procedure parameters with permanent variables

Before a parameter is first declared, it is possible to use OPEN-VARIABLE-CONTAINER within the first DECLARE-PARAMETER block in the procedure head to open a variable container, holding permanent variables. The variables declared here can then be used in initializing the relevant procedure parameters. (For further details, see section “Variable containers” on page 162.)

4.2 Creating the procedure body

The procedure body follows directly after the procedure head. It consists of a series of commands, statements and data that is executed when the procedure is run. Execution of the procedure can be controlled by control structures and branch commands.

The control structures include single command blocks, loops and branches. Branches include also the error handling blocks.

Each control structure is opened by an introductory command and closed by a termination command.

Control structures can also be nested.

Branch commands enable the sequential processing of commands to be interrupted by a jump to a defined point in the procedure.

The following sections describe first the construction and usage of control structures, and then the use of branch commands.

4.2.1 Defining single command blocks

In a single command block, commands can be assembled to form a logical unit. The block is introduced by a BEGIN-BLOCK command and terminated by an END-BLOCK command. Between these two commands are the commands which are to be executed in the block. Single command blocks are also referred to as BEGIN blocks.

The command line of the BEGIN-BLOCK command can begin with a tag, in accordance with the rules for S tags. This tag directly links the END-BLOCK command to the BEGIN-BLOCK command; it can also be used by other commands, as a branch destination.

If the BEGIN-BLOCK command line contains no tag, the implicit assignment between BEGIN-BLOCK and END-BLOCK commands takes effect: i.e. the END-BLOCK command always refers implicitly to the last BEGIN-BLOCK command that has not yet been terminated by END-BLOCK.

Layout of a BEGIN block without a tag

```
/BEGIN-BLOCK
[command sequence]
/END-BLOCK
```

Layout of a BEGIN block with a tag

```
/tag: BEGIN-BLOCK
[command sequence]
/END-BLOCK [BLOCK = tag]
```

4.2.2 Defining conditional branches

IF blocks can be used to generate branches during procedure execution. The sequence of commands in an IF block is executed or not, depending on the result of a condition.
An IF block begins with the IF command, and ends with the END-IF command. It may be given a tag.

If there is only one condition to be tested, the IF block consists of an IF command defining the condition, the associated sequence of commands and an END-IF command. The condition must be specified as a Boolean value.

```
/IF condition
  command sequence1

/END-IF
```

Here, if the condition in the IF command is satisfied, then `command sequence`$_1$ will be executed. If the condition is not satisfied, the IF block will be terminated. The command which follows the END-IF will then be executed.

As in higher programming languages, a distinction can be made between THEN and ELSE branches in IF blocks. `command sequence`$_1$ in the above example is the THEN branch, and the jump to the END-IF is the ELSE branch.

If another command sequence is to be executed as an alternative to `command sequence`$_1$, the IF block must contain an explicit ELSE branch. This ELSE branch contains the alternative command sequence. It begins with the ELSE command and ends with the END-IF command.

```
/IF condition
  command sequence1

/ELSE
  command sequence2

/END-IF
```

If the condition in the IF command is satisfied, `command sequence`$_1$ is executed and the block is terminated. If the condition is not satisfied, `command sequence`$_2$ is executed.
It is also possible to test several conditions consecutively within an IF block.

```
/IF condition
  command sequence1

/ELSE-IF condition
  command sequence2

/ELSE-IF condition
  command sequence3
...
/ELSE
  command sequencei

/END-IF
```

If the condition in the IF command is met, `command sequence`$_1$ is executed. If it is not met, the condition in the first ELSE-IF command is tested. If this condition is met, `command sequence`$_2$ is executed; if not, the condition in the next ELSE-IF command is tested, and so on. If no conditions are met, then `command sequence`$_i$ which follows the ELSE command is executed.

Each of the command sequences between the IF, ELSE-IF, ELSE and END-IF commands forms a separate block.

Example

Depending on the value of variable A, a file is to be created whose file name is to contain the value of the variable as a subname:

– If the value of variable A is less than 11, the file FILE.SMALL.&A is to be created.
– If the value of variable A is from 11 to 100, the file FILE.MEDIUM.&A is to be created.
– If A is greater than 100, the file FILE.LARGE.&A is to be created.

```
/DECLARE-PARAMETER A(INITIAL-VALUE=*PROMPT,TYPE=*INTEGER)
/IF (A < 11)
/   CREATE-FILE FILE.SMALL.&A,SUPPORT=*PUB-DISK(SPACE=*RELA(PRIM-ALLOC=30))
/ ELSE-IF (A < 101)
/   CREATE-FILE FILE.MEDIUM.&A,SUPPORT=*PUB-DISK(SPACE=*RELA(PRIM-ALLOC=60))
/ ELSE
/   CREATE-FILE FILE.LARGE.&A,SUPPORT=*PUB-DISK(SPACE=*RELA(PRIM-ALLOC=90))
/END-IF
```

The files are created by means of the CREATE-FILE command operand
SUPPORT = *PUB-DISK(SPACE = *RELA(PRIM-ALLOC = ...)):

– SUPPORT = *PUBLIC-DISK means that the file is created as a disk file on a public disk.
– SPACE = *RELATIVE means a relative memory allocation.
– PRIMARY-ALLOCATION means that the file is allocated n PAM pages of memory.

For a detailed description of how files are generated, see the manual entitled “Introductory Guide to the DMS” [1]. This manual also explains the terms “disk file”, “public disk”, “(relative) memory allocation” and “PAM page”.

4.2.3 Defining loops

A loop is used to execute repeatedly a sequence of commands, subject to a condition. Loops are formed using command blocks. A distinction is made between three types of loop block which are named after the initiating command: FOR blocks, WHILE blocks and REPEAT blocks.

FOR block

In the FOR block, which is also called a FOR loop, a command sequence is executed for each of the values which is assigned to a control variable.

The FOR block begins with the FOR command and ends with the END-FOR command. The FOR command assigns one of the following to a control variable

– a counter,
– the value of a list variable,
– an expression,
– or the values of the elements in a list of expressions, list variables and/or counters.

```
/FOR variable = counter / list-variable / expression [,CONDITION = condition]
[command sequence]
/END-FOR
```

The user defines a control variable; the number of loop passes and the value of the control variable are determined by the specification to the right of the equals sign (expression, list-variable, counter):

– counter
  The number of loop passes is determined by the initial value, the final value and the increment. The control value always contains the current loop value.
– list-variable
  The number of loop passes is determined by the number of list elements. With each loop pass, the control variable is assigned the value of the next list element, the elements of the list variable being processed in ascending order.
– expression
  The number of loop passes is determined by the number of elements in the value list. With each loop pass, the control variable is assigned the next expression (string, arithmetic, Boolean, ... expression), working from left to right.
– condition
  As an option, a condition can be specified in the form of a logical expression. Each loop pass will then be preceded by a check of the defined condition. If the condition is no longer met, the loop is terminated.

If a mixed list of counter, list variable and expression is specified, it will be processed from left to right.

Expression replacement (&...) in any of the FOR operands takes place only upon entering the FOR loop and not with each loop pass.

The control variable can be used in the FOR block to modify or replace commands by variable replacement, for instance.

The FOR block can have a tag.

Example 1

```
/DECLARE-VARIABLE A, MULTIPLE-ELEMENTS=*LIST  ——————————————————————————  (1)
/DECLARE-VARIABLE L  ———————————————————————————————————————————————————  (2)
/SET-VARIABLE A=1436,WRITE-MODE=*EXTEND  ———————————————————————————————  (3)
/A=1455,WRITE-MODE=*EXTEND  ————————————————————————————————————————————  (4)
/A=1577,WRITE-MODE=*EXTEND  ————————————————————————————————————————————  (5)
/FOR L=*LIST(A)  ———————————————————————————————————————————————————————  (6)
/ CANCEL-JOB JOB-ID=TSN(&L)  ———————————————————————————————————————————  (7)
/END-FOR
```

Explanations:

(1) Declaration of list variable A (DECLARE-VARIABLE: see page 541).

(2) Declaration of control variable L.

(3) The value 1436 is assigned to variable A (SET-VARIABLE: see page 541). WRITE-MODE=*EXTEND means that the list is extended by one element. The value is assigned to this new, last element. In this case, the last element is also the first element of the list.

(4) The value 1455 is assigned to variable A as a second value (the command name can be omitted for SET-VARIABLE).

(5) The value 1577 is assigned to variable A as a third value.

(6) The FOR loop begins with control variable L and the contents of variable A as a value list.

(7) For each value in variable A, a CANCEL-JOB command is issued to the TSN that is contained in variable L.

The following commands are issued by the FOR loop:

```
/CANCEL-JOB JOB-ID=*TSN(1436)
/CANCEL-JOB JOB-ID=*TSN(1455)
/CANCEL-JOB JOB-ID=*TSN(1577)
```

Example 2

```
/FOR I = *COUNTER(1,8)
/   ENTER-PROCEDURE ENTERFILE.&I
/END-FOR
```

The FOR loop creates eight ENTER-PROCEDURE commands for the files ENTERFILE.1 through ENTERFILE.8.

WHILE block

In the WHILE block, which is also called a WHILE loop, a command sequence continues to be executed until a condition defined by the user is no longer met. The WHILE block begins with the WHILE command and ends with the END-WHILE command.

The WHILE command contains the condition for traversing the loop; this condition is checked before each loop pass. If the condition is met, the command sequence within the block is executed. If the condition is not met, the loop is terminated and procedure execution is resumed at the command following the END-WHILE command.

The condition in the WHILE command is specified as a logical expression (see chapter “Expressions” on page 249).

```
/[tag:]WHILE condition
[command sequence]
/END-WHILE [tag]
```

Example

```
/COND = 1
/WHILE (COND < 9)
/   ENTER-PROCEDURE ENTERFILE.&COND
/   COND = COND + 1
/END-WHILE
```

The WHILE loop generates eight ENTER-PROCEDURE commands for the files from ENTERFILE.1 to ENTERFILE.8.

REPEAT block

In the REPEAT block, which is also called a REPEAT loop, a command sequence continues to be executed until a condition defined by the user is met. The block begins with the REPEAT command and ends with the UNTIL command.

Unlike the FOR and WHILE loops, the condition for terminating the loop is not contained in the introductory command but in the block termination command, UNTIL. For this reason, the loop is executed at least once.

At the end of each loop pass, the condition in the UNTIL command is checked. As long as the termination condition is not met, the command sequence within the block is again executed; as soon as the termination condition is met, the commands that follow the UNTIL command are processed.

The condition in the UNTIL command is specified as a logical expression (see chapter "Expressions" on page 249).

```
/REPEAT
[command sequence]
/UNTIL condition
```

Example

```
/DECLARE-VARIABLE A
/DECLARE-VARIABLE SWITCH-1(TYPE=*BOOLEAN)
/SET-VARIABLE A = 5
/SET-VARIABLE SWITCH-1 = ON
/REPEAT
/  A = A + 10
/  IF (A > 50)
/     SET-VARIABLE SWITCH-1 = OFF
/  END-IF
/  UNTIL (SWITCH-1 = OFF)
/SHOW-VARIABLE A
A = 55
```

4.3 Defining branches

Branches are executed using the branch commands, which terminate the sequential processing of commands, “branch” to a defined location in the procedure and resume procedure execution at that location.

The following branch commands are available in SDF-P:

- EXIT-BLOCK
- CYCLE
- GOTO
- INCLUDE-BLOCK

Branch commands can be used to address only those branch destinations that are contained in the current command block or in command blocks that surround the current command block.

It is not possible to branch to a command block that is nested more deeply by one or more levels.

Branch destinations are tags that are set in command calls. These tags can be used in the EXIT-BLOCK, CYCLE, GOTO and INCLUDE-BLOCK branch commands.

The branch commands can be divided into three groups:

- commands whose branch destination is always the begin of the block (INCLUDE-BLOCK)
- commands whose branch destination is always the end of the block (EXIT-BLOCK, CYCLE)
- commands that address any branch destinations (GOTO and, compatible with non-S procedures, SKIP-COMMANDS).

4.3.1 The beginning of a block as a jump destination

The INCLUDE-BLOCK command in a procedure jumps to the command line with the specified tag (tag:). The tag must mark the start of a BEGIN block (i.e. a BEGIN-BLOCK command must follow the tag). The execution of the procedure continues with the processing of the BEGIN block. When the end of the block is reached, processing continues with the command that follows the INCLUDE-BLOCK command.

Example

```
/ASSIGN-SYSLST TO=PROT.EINGABE,SYSLST-NUMBER=1
/...
/IF (EING='*START')
/ INCLUDE-BLOCK INFO-1
/END-IF     &* Continue here after executing the subprocedure INFO-1
/...
/IF (EING='*END')
/ INCLUDE-BLOCK INFO-1
/END-IF     &* Continue here after executing the subprocedure INFO-1
/...
/...
/...
/INFO-1: BEGIN-BLOCK    &* Start of the subprocedure INFO-1
/      WRITE-TEXT '&(TIME()): &(EING) was entered',OUTPUT=*SYSLST(1)
/END-BLOCK &*End and return to command line following INCLUDE-BLOCK
```

4.3.2 End of block as branch destination

4.3.2.1 Branch to end of random command block

The EXIT-BLOCK command terminates the processing of the command sequence within the block and branches to the block termination command. Procedure execution is then resumed with the command that follows the block termination command.

The block that is to be terminated can be addressed in the EXIT-BLOCK command call either implicitly by the preset value *LAST or explicitly via the name of the tag that precedes a block initiation command.

Example

```
/LOOP: WHILE (COND < 9)
/...
/IF (INP='*END')
/ EXIT-BLOCK LOOP
/END-IF
/...
/END-WHILE
/"Following EXIT-BLOCK, procedure execution is resumed here"
```

When branching with EXIT-BLOCK, the tag specified can be either that of the current block (as in the example above) or that of a surrounding block.

Example

```
/LOOPWH: WHILE (COND<9)
/...
/LOOPFOR: FOR I=*LIST(LIST)
/...
/          IF (COND=TRUE)
/...
/            EXIT-BLOCK LOOPWH
/...
/          END-IF
/...
/        END-FOR
/...
/END-WHILE
/"Execution resumes here after EXIT-BLOCK"
```

With the *ALL operand it is possible to terminate all surrounding blocks. Procedure execution is then resumed with the command following the END command that terminates the outside block on the highest procedure level. Using EXIT-BLOCK *ALL is practical, for example, in error handling blocks when the procedure is to be correctly terminated rather than resumed following an error.

Note

If /EXIT-BLOCK is written without a destination tag in an IF block, the procedure is continued after the associated END-IF.
The presetting BLOCK=*LAST should therefore not be used to address the block to be terminated unless the branch condition is specified in the EXIT-BLOCK command.

4.3.2.2 Branch to end of loop

The CYCLE command can be used only in loops. It terminates the processing of the command sequence in a loop and branches to the relevant end-of-loop command, where procedure execution is resumed.

In the case of REPEAT loops, the UNTIL end-of-loop command contains the loop condition; this condition is checked and, if appropriate, the next loop pass is initiated. In the case of FOR and WHILE loops, the end-of-loop commands (END-FOR, END-WHILE) contain the branch back to the beginning of the loop. There, the loop condition is rechecked and, if appropriate, the next loop pass is initiated.
The loop to which the command call is to refer can be addressed in the CYCLE command call either implicitly by the preset value *LAST or explicitly using the name of the tag that precedes the loop initiation command.

Example

```
/LOOP: WHILE (COND < 9)
/...
/IF (INP='*IGN')
/ CYCLE LOOP
/...
/END-WHILE  "Executed after CYCLE, i.e. return to WHILE"
```

Note

If - as in this example - the CYCLE command is used in an IF block (or in a single block), then the CYCLE command must specify a tag to identify the relevant loop block.

4.3.3 Random branch destinations

Branch to an S tag

The GOTO command branches to the command line with the specified tag (TAG:) within the procedure. The procedure run is then resumed with this command call.

The GOTO command can be used only to exit a command block or within a command block. It is not possible to branch into a command block from the outside using GOTO, i.e. GOTO can address branch destinations on superordinate nesting levels but not on subordinate levels.

Example

The lines below are examples illustrating what uses of the GOTO branch command are allowed and what uses are not allowed.

```
/LOOP1: WHILE (A < B)
/ADD1: X = X + A
/LOOP2: WHILE (X < Y)
/ADD2: A = A + 1
/  GOTO ADD1                 “Allowed; goes to surrounding loop”
/  END-WHILE LOOP2
/  GOTO ADD2                 “Not allowed; goes to inside loop!”
/END-WHILE LOOP1
```

Branch to a non-S tag

The SKIP-COMMANDS and WAIT-EVENT commands are SDF commands of the BS2000 basic configuration that are supported for the sake of compatibility with non-S procedures.

As branch destinations, these commands recognize tags in non-S format only (.tag). Non-S tags must not be used within command blocks; they can be used only on the highest procedure level, i.e. on nesting level 0.

SKIP-COMMANDS can be used to perform conditional and unconditional branches. The branch destination is the command with the specified tag, where procedure execution is resumed.

WAIT-EVENT can be used to make the chronogical execution of a job dependent on the user switch settings (only in batch jobs) or on the status of a job variable (JV).

Both commands are described in “Commands, Vol. 1-5” [3].

5 Calling and controlling procedures

S procedures can be called and started both from the system level and from within procedures. They can be executed in the foreground, i.e. synchronously with the calling task, or in the background, i.e. asynchronously in a batch task. The command used for the procedure call must be selected appropriately. The commands CALL-PROCEDURE and INCLUDE-PROCEDURE are provided for the former situation, and for the latter there is the ENTER-PROCEDURE command.

5.1 Calling S procedures in the foreground

The term “calling an S procedure in the foreground” means that the procedure is executed under the control of the task in which the procedure call command is issued: no new task is created. Such procedures are also referred to as synchronously called procedures, or as “interactive procedures”, because interactions with the user are possible if the procedure is called in dialog. Procedure calls in the foreground can be issued both from the system level and from within a procedure. S procedures are called in the foreground by a CALL-PROCEDURE command or by an INCLUDE-PROCEDURE command.

Procedures that are called by a CALL-PROCEDURE command are also referred to as call procedures; in the same way, procedures called by an INCLUDE-PROCEDURE command are referred to as include procedures.

Note

CALL-PROCEDURE can be used both to call S procedures in non-S procedures and to call non-S procedures in S procedures.

5.1.1 Choosing the call command

The command which may be used to call a procedure is specified in the procedure head, in the CALLER operand of the SET-PROCEDURE-OPTIONS command. If CALLER = *CALL is specified, the procedure may only be called by a CALL-PROCEDURE (similarly, for CALLER = *INCLUDE only by an INCLUDE-PROCEDURE). Only if CALLER = *ANY is specified may the procedure optionally be called by CALL-PROCEDURE or INCLUDE-PROCEDURE.

CALL-PROCEDURE and INCLUDE-PROCEDURE have exactly the same operands. However, they differ in their effect on the visibility of variables when the procedure is called from within another procedure. In procedures called with INCLUDE-PROCEDURE, unless otherwise specified, all variables are visible that are also visible in the calling procedure. In procedures that are called with CALL-PROCEDURE, unless otherwise specified, only variables local to the procedure are visible in the current procedure. (For more information, see section “Scope of variables” on page 157.)

Thus, the selection of a call command depends on whether the variable environment of the called procedure is to be retained.

Any procedure call must first name the procedure container. The caller can then define how the procedure is to be logged, whether a program loaded at execution time can be unloaded and whether the procedure is to be executed immediately in its entirety or should be interrupted for testing. The caller can also pass procedure parameters in the procedure call.

All these attributes are controlled by the operands of the procedure call commands, as described below. There is a special feature in the case of logging, which can also be set when the procedure is called; however, whether logging is performed and what is logged also depends on the settings that are valid within the procedure during its execution.

5.1.2 Specifying procedure containers

The procedure container is specified in the procedure call by means of the FROM-FILE operand. A procedure container can be a BS2000 file, a library element or a list variable. FROM-FILE = *VARIABLE(VARIABLE-NAME = ...) designates a list variable; each element of this list then contains a procedure line.

5.1.3 Passing procedure parameters

When a procedure is called with CALL-PROCEDURE or INCLUDE-PROCEDURE, the calling procedure can use the PROCEDURE-PARAMETERS operand to pass parameters to the called procedure. This operation is referred to as “parameter transfer”.

These procedure parameters must be declared in the procedure head of the called procedure by means of the DECLARE-PARAMETER command (see chapter “Creating S procedures” on page 81).

SDF-P handles the procedure parameters declared with DECLARE-PARAMETER as variables local to the procedure, i.e. as variables with SCOPE = *CURRENT (see the description of the DECLARE-VARIABLE command in page 607).

Procedure parameters can be specified as keyword parameters or as positional parameters.

Depending on the type of parameter transfer specified for the procedure in the DECLARE-PARAMETER command, either values or names of variables containing values can be specified for the parameters in the call.

5.1.3.1 Passing procedure parameters as positional parameters

Positional parameters are specified without the name of the corresponding parameter and are separated from each other by commas.

If procedure parameters are passed as positional parameters, the order in which the procedure parameters are declared in the called procedure must be taken into account. The parameters are evaluated in the order in which they are declared with DECLARE-PARAMETER commands.

This means that if only one procedure parameter is declared in each DECLARE-PARAMETER command, the first procedure parameter specified is mapped onto the parameter (name) that is declared as the first in the first DECLARE-PARAMETER command, the second is mapped onto the parameter name in the second DECLARE-PARAMETER command, etc.

If several procedure parameters are declared in one DECLARE-PARAMETER command, the procedure parameters that are passed are mapped onto them in the order in which the parameter names are specified.

When procedure parameters are declared, a valid value can already be preassigned to these parameters, i.e. they can be initialized. These procedure parameters can be “skipped” when parameters are passed in the procedure call if their initial value remains valid. When passing positional parameters, you must note where such “skipped” procedure parameters are located in the declaration sequence.

In the procedure call, procedure parameters that are already initialized do not have to be taken into account if they are the last procedure parameters in the declaration sequence.

If, however, these procedure parameters that are already initialized are followed by other procedure parameters, they must be passed as empty parameters; a single comma must be passed for the parameter to be skipped.

Example

PROC1 procedure

```
/...
/CALL-PROCEDURE PROC2, PROCEDURE-PARAMETERS = (MILLER, EDWARD, HARPER -
/AVENUE, , , 1234567)
```

PROC2 procedure

```
/SET-PROCEDURE-OPTIONS
/BEGIN-PARAMETER-DECLARATION
/   DECLARE-PARAMETER (LAST-NAME (*NONE, *STRING), FIRST-NAME (*NONE,-
/   *STRING))
/   DECLARE-PARAMETER STREET(*NONE, *STRING)
/   DECLARE-PARAMETER CITY('CHICAGO', *STRING)
/   DECLARE-PARAMETER (AREA-CODE('312', *STRING), TEL(*NONE, *STRING))
/END-PARAMETER-DECLARATION
```

With the procedure call, six procedure parameters are passed to PROC2: last name (MILLER), first name (EDWARD) and street (HARPER AVENUE). No values are passed for the parameters CITY and AREA-CODE; for these parameters, commas are inserted. As a final value, the telephone number is passed. The procedure parameters CITY and AREA-CODE are preassigned with the values CHICAGO and 312 (because, for example, this is the address list of a company in Chicago). Since TRANSFER-TYPE = *BY-VALUE is preset, the specified procedure parameters are handled as value entries.

5.1.3.2 Passing procedure parameters as keyword parameters

Parameters can also be passed as keyword parameters. A keyword is the name by which the procedure parameter is declared in the DECLARE-PARAMETER command. In accordance with the rules for abbreviating keyword parameters in commands, it is also possible here for keywords to be abbreviated down to the shortest unique abbreviation.

When procedure parameters are passed as keyword parameters, they can be in any order, since the parameter names must be unique.

Parameters that are initialized in the called procedure do not need to be taken into account unless their initial value is to be overwritten.

Example

If procedure parameters were passed as keyword parameters, the procedure call in the previous example would look as follows:

```
/CALL-PROCEDURE PROC2,-
/PROCEDURE-PARAMETERS = (LAST-NAME = MILLER, FIRST-NAME = EDWARD,-
/STREET = HARPER AVENUE, TEL = 1234567)
```

5.1.3.3 Mixing positional and keyword parameters

Positional parameters and keyword parameters can be passed in the procedure call together. In this case, the positional parameters must be specified first, followed by the keyword parameters.

Example

The parameter transfer for the Chicago address list could then look like this:

```
/CALL-PROCEDURE PROC2, PROCEDURE-PARAMETERS = (MILLER, EDWARD, HARPER -
/AVENUE, TEL = 1234567)
```

The last name, first name and street are passed as positional parameters and the telephone number is passed as a keyword parameter. The CITY and AREA-CODE parameters do not need to be taken into account since they were initialized when they were declared.

5.1.3.4 Type of parameter transfer

When procedure parameters are declared with DECLARE-PARAMETER, the handling of the character string passed for the procedure parameter is also determined, in the TRANSFER-TYPE parameter.

If TRANSFER-TYPE = *BY-VALUE, the character string which is passed is assigned to the corresponding procedure parameter as a value.

Example

```
/CALL-PROCEDURE PROC2, PROCEDURE-PARAMETERS = (MILLER, EDWARD, HARPER -
/AVENUE, TEL = 1234567)
```

In this example, the procedure parameter LAST-NAME is assigned the value MILLER, the procedure parameter FIRST-NAME is assigned the value EDWARD, etc.

If TRANSFER-TYPE = *BY-REFERENCE, the characters string which is passed is evaluated as a variable container for the formal procedure parameter. Consequently, the called procedure can also return results to the caller using this procedure parameter.

Example

Calling procedure:

```
/DECLARE-VARIABLE LAST-NAME('MILLER',*STRING)
/DECLARE-VARIABLE FIRST-NAME('EDWARD',*STRING)
/DECLARE-VARIABLE STR('HARPER AVENUE',*STRING)
/DECLARE-VARIABLE TELEPHONE('1234567',*STRING)
/....
/CALL-PROCEDURE PROC2, PROCEDURE-PARAMETERS=(LAST-NAME,FIRST-NAME, STR,,,TEL)
```

Called procedure PROC2:

```
/SET-PROCEDURE-OPTIONS
/BEGIN-PARAMETER-DECLARATION
/DECLARE-PARAMETER LAST-NAME (*NONE, *STRING,-
/  TRANSFER-TYPE = *BY-REFERENCE)
/DECLARE-PARAMETER FIRST-NAME (*NONE, *STRING,-
/  TRANSFER-TYPE = *BY-REFERENCE))
/DECLARE-PARAMETER STR(*NONE, *STRING,-
/  TRANSFER-TYPE = *BY-REFERENCE))
/DECLARE-PARAMETER CITY('CHICAGO', *STRING,-
/  TRANSFER-TYPE = *BY-VALUE)
/DECLARE-PARAMETER AREA-CODE('312', *STRING,-
/  TRANSFER-TYPE = *BY-VALUE)
/DECLARE-PARAMETER TEL(*NONE, *STRING,-
/  TRANSFER-TYPE = *BY-REFERENCE))
/END-PARAMETER-DECLARATION
```

Variable names are passed as procedure parameters. Since TRANSFER-TYPE = *BY-REFERENCE applies for the procedure parameters in PROC2, the transferred string is not assigned to the procedure parameter as a value; instead, the variables are linked together.

Example

Procedure P contains the following lines:

```
/SET-PROCEDURE-OPTIONS
/BEGIN-PARAMETER-DECLARATION
/DECLARE-PARAMETER TOTAL (TRANSFER-TYPE=*BY-REFERENCE)
/DECLARE-PARAMETER (P1(TYPE=*INTEGER),P2(TYPE=*INTEGER))
/END-PARAMETER-DECLARATION
/TOTAL = P1+P2
```

The following commands are entered interactively:

```
/DECLARE-VARIABLE S
/CALL-PROCEDURE P, (S,3,5)
/SHOW-VARIABLE S
```

This results in the output:

```
S = 8
```

Variable names are transferred as procedure parameters by means of a “container mechanism”; the procedure parameter serves as a container for the variable so that the called procedure can access this variable and evaluate its contents.

Variable names can be used as procedure parameters only if the variable and the procedure parameter are declared with the same data type (the variable with DECLARE-VARIABLE in the calling procedure and the procedure parameter with DECLARE-PARAMETER in the called procedure). For example, they must both be declared with TYPE = *INTEGER. One cannot be declared with TYPE = *INTEGER and the other with TYPE = *ANY.

5.1.4 Requesting logging

The SET-PROCEDURE- OPTIONS command serves to define whether logging is performed or not, and what is to be logged during the procedure run. Once this general definition has been made, the caller can set more detailed options in the CALL-PROCEDURE/ INCLUDE-PROCEDURE command.

Logging is affected by the settings of various commands, which can be grouped as below for foreground procedures in respect of the procedure and job level:

- SDF-P commands, which are used to define within the procedure whether logging is allowed: SET-PROCEDURE-OPTIONS and MODIFY-PROCEDURE-OPTIONS, LOGGING-ALLOWED operand
- SDF-P commands for the procedure call in which the caller can request a log: CALL-PROCEDURE and INCLUDE-PROCEDURE, LOGGING operand
- SDF-P commands for the procedure’s test phase: TRACE-PROCEDURE and MODIFY-PROCEDURE-TEST-OPTIONS.
- Commands that affect logging on a job level or for SDF: MODIFY-JOB-OPTIONS and MODIFY-SDF-OPTIONS.

The MODIFY-JOB-OPTIONS command affects the logging of job execution (for example, whether a supplementary SYSLST log or hard copies are to be generated). MODIFY-SDF-OPTIONS affects the log format. These two commands are not SDF-P commands. They are described in detail in “Commands, Vol. 1-5” [3].
This section deals only with the effects of SDF-P commands on logging. To find out whether logging is currently being performed, use the LOGGING-MODE( ) function (see page 444).

5.1.4.1 Permissibility of logging

Whether logging is allowed is controlled in part by the protection mechanisms of the data management system. The procedure can be assigned a read password. (Read protection can be canceled only if the appropriate password is specified.) ACL or BASIC-ACL can be used to grant execution and read authorization separately.

Whether logging is allowed is also controlled within the procedure with the SET-PROCEDURE-OPTIONS and MODIFY-PROCEDURE-OPTIONS commands, in each case by means of the LOGGING-ALLOWED operand. A distinction is made between logging commands and logging data.

If logging is to be prohibited for commands and/or data from the beginning of procedure execution, the procedure must begin with the SET-PROCEDURE-OPTIONS command. There, the operand LOGGING-ALLOWED = *PARAMETERS(...) must be specified with CMD = *NO and/or DATA = *NO.

If command and/or data logging is to be prohibited for only part of the procedure, the MODIFY-PROCEDURE-OPTIONS command must be called in the procedure body and the corresponding value specified in the operand LOGGING-ALLOWED = *PARAMETERS(...). The (*YES/*NO) setting is retained until it is changed by another MODIFY-PROCEDURE-OPTIONS command.

If logging is prohibited for parts of the procedure, this setting cannot be changed “externally”, i.e. it cannot be changed using procedure call commands or interactive commands.

Logging should be prohibited, for example, if the caller is not supposed to “see” certain commands or data.

5.1.4.2 Logging the normal execution of a procedure

For foreground procedures, the caller specifies in the procedure call whether a log should be created for the execution of the procedure, using the LOGGING operand. When this is done, it makes no difference whether the procedure is called by a CALL-PROCEDURE or an INCLUDE-PROCEDURE, because the LOGGING operand has the same function for both CALL-PROCEDURE and INCLUDE-PROCEDURE: the caller specifies whether commands are to be logged by means of CMD = *YES/*NO, and whether data is to be logged by DATA = *YES/*NO.

5.1.4.3 Logging procedure test runs

The MODIFY-PROCEDURE-TEST-OPTIONS command can be used to set logging globally for all the procedures of a task. This setting applies to all nesting levels.

Using the TRACE-PROCEDURE command in interactive mode, the user can trace procedure execution step by step. All commands that are executed from the time TRACE-PROCEDURE is called until the next interruption are automatically logged (provided that logging is allowed).

5.1.4.4 Restrictions for procedures with a read password

An attempt to log a procedure is regarded as a read access. If the file to be logged has a read password, then this password must have been entered in the current password list with ADD-PASSWORD. If this is not the case, a counter for unsuccessful attempts in the system is incremented and the task is aborted if the limit specified in the class 2 option for PWERRORS is reached. However, such behavior could be an unacceptable restriction for the user if, for example, he or she activates logging globally with MODIFY-PROCEDURE-TEST-OPTIONS and his/her calling hierarchy includes password-protected procedures whose passwords he/she does not know.

For this reason, procedures protected by read passwords can never be logged with MODIFY-PROCEDURE-TEST-OPTIONS, even if their passwords are in the current password list. This ensures that no unsuccessful read access attempts are executed and the error counter is not incremented. Such procedures can thus be logged only directly with the aid of the LOGGING operand in CALL-PROCEDURE or in INCLUDE-PROCEDURE. If this is done, an access attempt with an invalid password will again increment the error counter. The same applies to the TRACE-PROCEDURE command.

5.1.4.5 Contents of the logging records

The specification of whether the current entry should apply to commands, to data, or to both is possible within the procedure and when calling background procedures. The LOGGING-ALLOWED and LOGGING operands have the same format for all commands.

If the entry is to apply both to commands and data, the operand must be specified as LOGGING[-ALLOWED] = *YES (or *NO).
If the entry is to apply to commands only, the operand value must be specified as PARAMETERS(CMD = *YES) or (CMD = *NO).
In the same way, if the entry is to apply to the logging of data only, the operand value must be PARAMETERS(DATA = *YES) or (DATA = *NO).

The table below shows the interplay between the settings in the SET-PROCEDURE-OPTIONS, MODIFY-PROCEDURE-OPTIONS and CALL-PROCEDURE/INCLUDE-PROCEDURE commands.

It is possible to query whether commands or data are being logged by means of the predefined function LOGGING-MODE( ) with the operand STREAM = *CMD/*DATA.

SET-PROC-OPT MOD-PROC-OPT	**CALL-PROCEDURE / INCLUDE-PROCEDURE Operand LOGGING =**					
Operand LOGGING-ALLOWED=	*YES	*NO	*PARAM (CMD = *Y)	*PARAM (CMD = *N)	*PARAM (DATA = *Y)	*PARAM (DATA=*N)
*YES	C/D	-/-	C/*	-/*	*/D	*/-
*NO	-/-	-/-	-/-	-/-	-/-	-/-
*PAR(CMD= *Y)	C/D	-/-	C/*	-/*	*/D	*/-
*PAR(CMD = *N)	-/D	-/-	-/*	-/*	*/D	-/-
*PAR(DATA = *Y)	C/D	-/-	C/*	-/*	*/D	*/-
*PAR(DATA = *N)	C/-	-/-	C/*	-/-	*/-	*/-

C/D	Commands and data are logged
-/-	Neither commands nor data are logged
C/-	Only commands are logged
-/D	Only data is logged
*	Whether commands/data are logged depends on the setting in the MODIFY-PROCEDURE-TEST-OPTIONS command.

Notes

- If S procedures are logged, the log always refers to the line number and procedure level, so that it is easy to locate any errors which occur.
- When loops are logged, the command which initiates the loop is logged only once. The command that terminates the loop is logged for each pass through the loop.
- If a command call is preceded by an S tag (tag:), the command and the tag are logged separately.
- Comment commands are not logged. (A comment command is a command line that consists only of a comment.)

5.1.5 Unloading programs

The UNLOAD-ALLOWED operand allows the user to determine whether the program loaded at execution time can be unloaded.
If an attempt is made to unload a program when it is not allowed, an error occurs.

If the program cannot be unloaded (UNLOAD-ALLOWED = *NO), no command that unloads a program can be called during procedure execution.

Such a command would be, for example, START-EXECUTABLE-PROGRAM, which starts up a program. However, only one program can be loaded at any one time. If another program is already loaded, it is first unloaded, before the second program is loaded and started up with START-EXECUTABLE-PROGRAM.

5.1.6 Setting the execution mode

The EXECUTION operand in the CALL-PROCEDURE and INCLUDE-PROCEDURE commands can be used to set the execution mode for the procedure.

EXECUTION = *YES is the default setting: the procedure is executed immediately after the preanalysis.

The setting EXECUTION = *NO is useful for preventing procedures which are being tested from immediately being executed in full. With this setting, the commands in the procedure head are executed first. The procedure body is then preanalyzed, i.e. it is checked to see whether the control structures are correct. However, the procedure body is not yet executed.

5.1.7 Error transfer

“Error transfer” refers to the fact that information on errors that occur is transferred from a subordinate procedure to the superordinate, calling, procedure.

In foreground procedures, error information can be transferred from the subordinate, called procedure to the superordinate, calling procedure by means of various mechanisms.

– the EXIT-PROCEDURE command
– the transfer of error information using variables.

Transfer using EXIT-PROCEDURE:

1. In the command call, error information is specified for the error classes Subcode1, Subcode2 and Maincode (e.g. by variable replacement: the variables contain the command return code or user-defined error information for case differentiation).
2. In the calling procedure, these components are then evaluated by the built-in functions SUBCODE1( ), SUBCODE2( ) and MAINCODE( ).

The ERROR operand in the EXIT-PROCEDURE command can serve to return information to the caller on errors that occurred during the procedure run and were intercepted by the internal error handling. If the procedure is canceled as errored, the error information is automatically transferred, since the error situation is not terminated.

Transfer using variables:

1. The variable must be visible: In CALL procedures, the variable must be imported as task-global; in INCLUDE procedures, variables of the calling procedure can be accessed directly.
2. If an error occurs, an appropriate value is assigned to the variable.
3. An IF block decides whether procedure execution should be resumed or terminated immediately.
4. After the subordinate procedure has been terminated, the variable in the superordinate, calling procedure is evaluated.

5.1.8 Procedure termination

Procedures can be terminated in several different ways:

– with the EXIT-PROCEDURE command
– with the END-PROCEDURE command
– with the CANCEL-PROCEDURE command

The END-PROCEDURE command at the end of the procedure is supported only for reasons of compatibility with non-S procedures. When a procedure is terminated with END-PROCEDURE, error information cannot be transferred or a program cannot be resumed.

The CANCEL-PROCEDURE command can be used to cancel procedure execution entirely. SYSCMD is reset to the primary allocation. If CANCEL-PROCEDURE is called in a subordinate procedure, all superordinate procedures are likewise canceled.

If a procedure does not contain a termination command, it is automatically terminated after the last command is executed. In the event of an error, the error code is returned to the caller.

Exiting with EXIT-PROCEDURE

S procedures are always terminated by the EXIT-PROCEDURE command. EXIT-PROCEDURE terminates procedure execution, supplies error information to the caller and also causes a program to be resumed, if appropriate.

EXIT-PROCEDURE is executed only if the procedure has been executed correctly up to this command call, i.e. if it has not been canceled by an error or the CANCEL-PROCEDURE command.

The RESUME-PROGRAM operand can be used to resume a program that is loaded when the procedure is terminated.

Because the caller’s SYSFILE environment can be amended in foreground procedures, (SYSTEM-FILE-CONTEXT=*SAME-AS-CALLER) the caller must ensure, after the procedure has terminated, that the correct SYSFILE environment is in effect.

For information on how error information is supplied to the caller, see section “Error transfer” on page 116.

5.2 Calling S procedures in the background

Calling an S procedure in the background, also referred to as an asynchronous procedure call, has the effect that it is executed independently of the calling job; a new job is generated with its own task sequence number (TSN).
In many respects, background procedures behave like foreground procedures. Hence, the description which follows deals only with the differences.

5.2.1 The ENTER-PROCEDURE call command

If an S procedure is to be started as a background procedure, it must be called by an ENTER-PROCEDURE command. An ENTER file is created internally when this is done, and the file is started with the ENTER-JOB command.

Method

1. A copy of the procedure file is made under the name `S.PROC.tsn.date.time`, with *date* in the format yyyy-mm-dd and *time* in the format hh.mm.ss.
2. An ENTER file with the following contents is created under the name `S.E.tsn.date.time`:

```
/SET-LOGON-PARAMETERS
:
:
:
 /CALL-PROCEDURE FROM-FILE=S.PROC.tsn.date.time, -
/                         PROCEDURE-PARAMETERS=(parameter)
:
:
:
/EXIT-JOB SYSTEM-OUTPUT=option
```

   The value of *parameter* corresponds to the specification in the operand PROCEDURE-PARAMETERS. (Procedure parameters can only be passed as values (*BY-VALUE).) After execution of the procedure, the copy of the procedure file is deleted (if the background procedure is not to be repeated). The value of *option* corresponds to the specification in the SYSTEM-OUTPUT operand.
3. The ENTER file is started with ENTER-JOB. The specifications for the operands PROCESSING-ADMISSION, JOB-CLASS, JOB-NAME, MONJV, JV-PASSWORD, JOB-PRIORITY, RERUN-AFTER-CRASH, FLUSH-AFTER-SHUTDOWN, START, REPEAT-JOB, RESOURCES, LISTING and JOB-PARAMETER are taken from the entries in the ENTER-JOB command.

The operands of the ENTER-PROCEDURE command that determine the attributes of the background procedure are described later on in section “Job attributes” on page 120.

Notes

- “Remote Enter” is supported. This means that ENTER-PROCEDURE can also be used to call the procedure on a remote computer.
- The ENTER-PROCEDURE command cannot be called from a console.

5.2.2 Specifying procedure containers

The procedure container is named in the FROM-FILE operand. Only a BS2000 file or a library element can be specified as a procedure container. Procedures that are to be called in the background cannot be transferred in list variables, since the variable would be known only in the calling procedure and not in the executing procedure.

If the procedure is stored in a file under a foreign user ID, it must be cataloged as being shareably executable, otherwise, it cannot be accessed.

If the procedure file is protected by a password, this password must be specified in the PROCEDURE-PASSWORD operand or in an ADD-PASSWORD command.

5.2.3 Passing procedure parameters

In the PROCEDURE-PARAMETERS operand, procedure parameters can be passed as values only (declared in the DECLARE-PARAMETER command with TRANSFER-TYPE = *BY-VALUE). Variable names cannot be passed, since variables are known only in the calling job and not in the executing job.

Whether the actual and formal parameters match is not checked until execution time.

5.2.4 Requesting logging

The LOGGING operand in the call command for background procedures (i.e. ENTER-PROCEDURE) has a different format than in the CALL-PROCEDURE and INCLUDE-PROCEDURE commands, and hence different effects.

In the case of ENTER-PROCEDURE, this operand has the following function: the caller uses *YES and *NO to switch logging of job execution on and off and uses *STD to switch on logging only for the case where the file is not read-protected.

Unless otherwise specified, procedures that are called in the background are logged on SYSOUT. The number of records to be written to SYSLST (or SYSOUT) can be set in the ENTER-PROCEDURE command with the SYSLST-LIMIT (or SYSOUT-LIMIT) operand.

5.2.5 Job attributes

The ENTER-PROCEDURE command contains a number of operands that relate to the background procedure. These operands can be combined in the following groups:

– job attributes (JOB-CLASS, JOB-PRIORITY, JOB-NAME, JOB-PARAMETER operands)
– monitoring job variables (MONJV, JV-PASSWORD operands)
– startup behavior (START, REPEAT-JOB, RERUN-AFTER-CRASH, FLUSH-AFTER-SHUTDOWN operands)
– resources (operand RESOURCES(RUN-PRIORITY, CPU-LIMIT, SYSLST-/SYSOPT-LIMIT))

5.2.5.1 Setting the job attributes (including monitoring job variables)

The job attributes for background procedures comprise:

– job class
– job priority
– job name
– job parameters
– monitoring job variables

These attributes are set by corresponding operands. The user can determine which values it is permissible to set by means of a SHOW-USER-ATTRIBUTES or SHOW-JOB-CLASS command (for a description of this command, see “Commands, Vol. 1-5” [3]).

Notes
– The user can issue any arbitrary job name, but it must not be longer than eight characters.
– The job parameters are used to identify further job class attributes, the meaning of which is defined by the system administrator.

5.2.5.2 Setting the startup behavior

The startup behavior of a background procedure can be set by means of the following operands: START, REPEAT-JOB, RERUN-AFTER-CRASH, FLUSH-AFTER-SHUTDOWN.

The START operand is used to define the time point when the procedure is to be started. It is possible to specify an absolute date or a relative time, e.g. immediately, as soon as possible, within a specified time interval, etc.

The REPEAT-JOB operand can be used to define a cyclic interval at which the procedure is to be repeatedly started., e.g. daily, weekly, etc.

Using the RERUN-AFTER-CRASH operand, it is possible to specify whether a procedure should be automatically restarted if it has been terminated as a result of a system error or shutdown.

The FLUSH-AFTER-SHUTDOWN operand is evaluated if the background procedure was not started during the current session. FLUSH-AFTER-SHUTDOWN can be used to specify whether a background procedure is to be removed from the job queue if it has not been started by the end of the session.

5.2.5.3 Specifying resource usage

The RESOURCES operand limits the use of system resources. It determines the run priority, the maximum amount of CPU time that the background procedure is allowed to use and the number of records that are output to SYSLST and SYSOUT files.

5.2.6 Error transfer

Procedures called in the background are executed in separate jobs; for this reason, error information cannot be transferred by means of variables or command return codes.

In this case, job variables are provided for storing error information. Compatibly with non-S procedures, user switches can also be used in S procedures. Other jobs can access both job variables and user switches.

Differentiated error information can be transferred by means of job variables, since the contents of a job variable can be up to 256 characters long. (For more information on job variables, see the “Job Variables” manual [5].)

5.2.7 Terminating procedures

Background procedures are also terminated by an EXIT-PROCEDURE, an END-PROCEDURE or CANCEL-PROCEDURE.

After termination of a procedure that was called in the background, the batch job in which it was executed is also terminated. If execution was error-free, the monitoring job variables are set to $T; if execution was aborted due to error, they are set to $A. “Abortion due to an error” is in this case a cancellation of the procedure with CANCEL-JOB or its explicit termination with EXIT-JOB MODE = *ABNORMAL (or the ISP command ABEND), but not a call of the command EXIT-PROCEDURE ERROR = *YES!

5.3 Nesting S procedures

When a procedure is called within another procedure, this type of procedure call is also referred to as a nested call. By extension, the resulting procedure link is called procedure nesting.

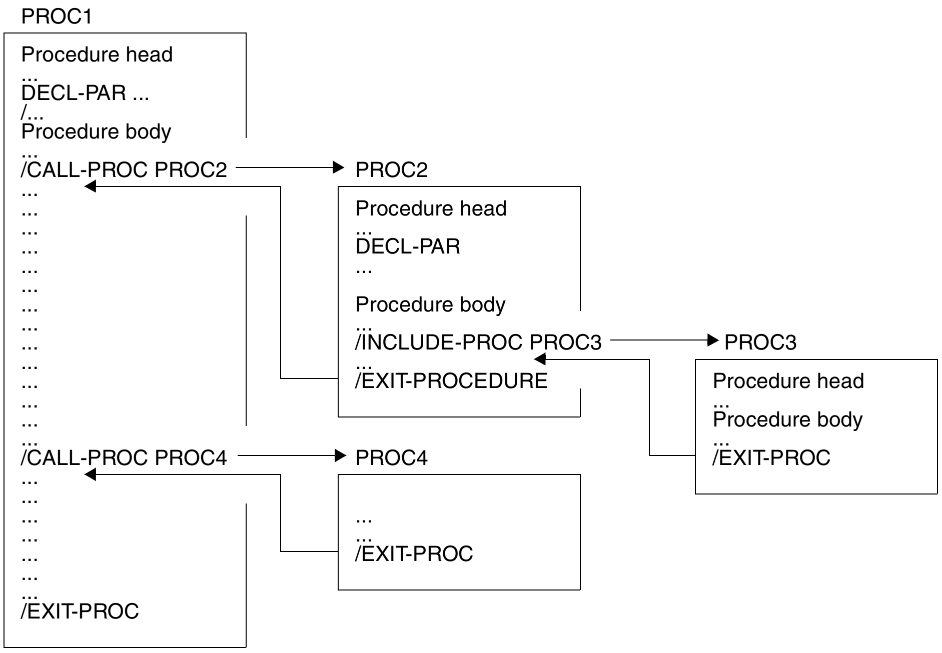
Note that, although nested procedures can be called, one procedure cannot be “written” inside another. This means that each procedure container can contain one procedure only.

If nested procedures are called, the “inner” (=called) procedure is completely processed before control is returned to the calling (= superordinate) procedure. The called procedure is subordinate to the calling procedure.

There are three commands for procedure nesting:

– CALL-PROCEDURE and INCLUDE-PROCEDURE for the procedure call
– EXIT-PROCEDURE for procedure termination

The diagram below shows an example of procedure nesting.



In this example, PROC1 is called on the system level; PROC1 is the superordinate procedure for PROC2, PROC3 and PROC4. PROC2, PROC3 and PROC4 are called from within a procedure, i.e. they are subordinate procedures; PROC2 and PROC4 are subordinate to PROC1 only and PROC3 is subordinate to PROC2 and PROC1. PROC2 contains PROC3.

Nested procedures are always called in the foreground. Hence the same rules apply for the call (or start), execution and termination as for foreground procedures which are called on the system level.

Nested procedures may call background procedures. However, these procedures are not nested within the caller hierarchy, but instead initiate a new job with its own independent procedure nesting.

It is important when executing nested procedures that there be mutual access to variables (i.e. to variables that are declared in the superordinate or in the subordinate procedure).

Whether variables can be accessed in nested procedures, i.e. are visible, depends on the declaration of the variable scope and on the command used to call the procedure.

SDF-P provides the following options for accessing variables from superordinate procedures:

– Passing a variable name as a procedure parameter with the procedure call.
– Redeclaring task-global variables that are not visible. This means that each of these variables must be declared using a DECLARE-VARIABLE command and with exactly the same attributes as in the superordinate procedure.
– Importing task-global variables using the IMPORT-VARIABLE command.

For a detailed description of the relationship between the visibility of variables and the procedure call command, as well as the “redeclaration” and importation of variables, see section “Scope of variables” on page 157.

For a description of how variable names are passed as procedure parameters, see section “Passing procedure parameters” on page 106.

5.4 Internal subprocedures

A command block that starts with a BEGIN-BLOCK command and ends with a END-BLOCK command (also called a BEGIN block, see page 568) can be used within the procedure as a subprocedure. Such a procedure is called with the INCLUDE-BLOCK command (see page 681). The execution of the procedure jumps to the command line that starts with the specified tag and that contains the corresponding BEGIN-BLOCK command. After processing the BEGIN block, the procedure continues with the command that follows the INCLUDE-BLOCK command.

Subprocedures are used in a manner similar to that of subroutines in higher level programming languages. They promote clarity and make the procedure easier to maintain and update. The calls for such procedures execute faster than the calls for external procedures in which an additional procedure container must first be opened.

5.5 Procedure interruption

The procedure interruption feature applies only to procedures which have been called in a dialog or in the foreground (using CALL-PROCEDURE or INCLUDE-PROCEDURE).
A distinction must be made between interruptions on the system level and interruptions within procedures.

Interruptions within procedures can be triggered at any time by means of the HOLD-PROCEDURE command. A procedure can be interrupted on the system level with the K2 function key, whenever this is allowed. Whether or not a procedure is interruptible can be set in the procedure head by means of the INTERRUPT-ALLOWED operand in the SET-PROCEDURE-OPTIONS command. This setting can subsequently be modified in the procedure body by means of the MODIFY-PROCEDURE-OPTIONS command, again via the INTERRUPT-ALLOWED operand. In this way, interruptibility can be enabled or disabled for individual parts of the procedure. The setting for interruptibility cannot be modified when the procedure is called.

In both cases (interruption with the HOLD-PROCEDURE command and interruption with the function key K2), procedure execution is resumed when the RESUME-PROCEDURE command is entered at the system level.

If an interruption of procedure execution is requested by means of the K2 key during a procedure that is not interruptible, the request is ignored.

During procedure interruption, the user can, for example, call commands in interactive mode on the system level in order to check or modify the SYSFILE environment or can access all variables that are local to the interrupted procedure. Thus, the interruption dialog is part of the visibility range of the interrupted procedure.

During an interruption dialog, variables can also be declared which are then considered to be variables local to the interrupted procedure. If, for example, an attempt is made within a procedure to access a variable that is not declared, the procedure can be interrupted within the framework of the relevant error handling routine and the missing variable can be declared on the system level.

5.6 Uninterruptibility

Protecting procedures and programs from being interrupted

By setting the operand value INTERRUPT-ALLOWED=*NO in SET-PROCEDURE-OPTIONS or in MODIFY-PROCEDURE-OPTIONS, a procedure can be protected against interruptions by commands input in dialog mode.

If a program is activated in a procedure, the procedure must also be protected against uncontrolled command inputs; e.g. if the program is processing data or statements, and some of these request the program to execute commands (CMD macro) or to issue an interrupt (BKPT) which is not provided for in the procedure.

Example

```
/SET-PROCEDURE-OPTIONS ...,INTERRUPT-ALLOWED=NO
/ASSIGN-SYSDTA TO=*PRIMARY
/START-LMS
                       ----------> Dialog at the terminal:
                                         //
                                         //...
                                         //END
/MODIFY-JOB-SWITCHES ON=(4,5)
/START-EDT
                       ----------> Dialog at the terminal:
                                          *
                                            * ...
                                          *
/EXIT-PROC
```

If a procedure is protected against interruption, then normally any program which is executed in this procedure is also protected against interruption by the $\boxed{K2}$ key. However, if a STXIT routine is defined in the program for $\boxed{K2}$ interruptions (ESCPBRK), then the $\boxed{K2}$ event is passed to this STXIT routine. Responsibility for handling this event then lies with the program. This will in any case lead to problems, if the program reacts with a BKPT macro.

Example

```
/SET-PROCEDURE-OPTIONS ...,INTERRUPT-ALLOWED=NO
/START-EXECUTABLE-PROGRAM ...

//
//...                                            <--------- K2
K2-STXIT>  Output PC
                     BKPT
                     “Return to the command level”
/EXIT-PROCEDURE
```

5.6.1 Implicit procedure protection

If a program is to be activated in a procedure, the program should check whether the procedure is protected against interruptions; i.e. the program should protect the procedure implicitly against interruptions which the program could cause during the course of its execution.

It is nevertheless possible for actions such as CMD, BKPT, STXIT, etc. to be executed by the program: first, for internal purposes; and secondly if there are user requests to this effect.
The program should interrogate the procedure attribute INTERRUPT-ALLOWED by means of a CLIGET macro call before performing any of these actions. If this attribute is set to *NO, the action should be rejected.

Programs can always be interrupted by procedure commands such as HOLD-PROGRAM.

5.6.2 Protecting programs explicitly

User programs which process security-related data must be protected against uncontrolled input of commands. This protection must be provided in all input modes (for foreground and background procedures). To achieve this, the settings must be made within any programs which contain security-critical parts. This is also referred to as explicit program protection. It is activated by a CLISET macro call. (For further details see also chapter “Program interfaces” on page 305). This makes the programs themselves responsible for calls such as CMD, BKPT, K2-STXIT, etc., when security-related information is being processed.

Acceptance and rejection of events

Program	**Non-interruptible**	**Interruptible**	
Procedure	**Arbitrary**	**Non-interruptible**	**Interruptible**
K2 key	z	r	a
K2-STXIT	a,re	a,ri	a
CMD macro	a,re	a,ri	a
BKPT macro	a,re	a,ri	a
other macros	a,re	a,ri	a
//EXEC-SYS-CMD	r	c	a
//HOLD-PROGRAM	r	c	c
/HOLD-PROGRAM	r*	a*	a*

Key:

a:	accepted by the system
re:	should be rejected by the program for explicit program protection (CLISET)
ri:	soll vom Programm bei impliziten Programmschutz zurückgewiesen werden (CLIGET)
r:	rejected by the system
c:	rejected by the system if SYSSTMT is not SYSCMD
a*:	regarded by the system as data if SYSSTMT or SYSDTA, as applicable, is not SYSCMD. Otherwise, it will be accepted by the system.
r*:	regarded by the system as data if SYSSTMT or SYSDTA, as applicable, is not SYSCMD. Otherwise, it will be rejected by the system..

re and ri are the responsibility of the program.

Rejection of events

- Interruption by K2 is simply ignored, the processes continue running unimpaired.
- /HOLD-PROGRAM, //HOLD-PROGRAM and PROGRAM-INPUT=*MIXED-WITH-CMD return EOF to the other program.
- //EXECUTE-SYSTEM-COMMAND is rejected, and spin-off is activated for the statements.
- //HOLD-PROGRAM is always rejected if SYSSTMT is not assigned to SYSCMD.

Coexistence of different protection modes

The two protection modes (implicit and explicit) are two different functions. They coexist, and for some events they overlap.

A protection mode cannot be inherited. Nevertheless, there are effects which are similar enough to make use of the term inheritance appropriate.

Namely

- explicit program protection includes implicit program protection,
- implicit program protection includes procedure protection,
- and procedure protection includes implicit program protection: however, only if this protection is provided by the program itself at the request of the procedure, by a CLIGET macro call.

Notes

- Since explicit program protection is implemented by an SVC, the program can be interrupted in command mode by the K2 key before the SVC is executed: e.g. if the K2 key is pressed during execution of the LOAD-/START-EXECUTABLE-PROGRAM (or LOAD-/START-PROGRAM).
  If pressing the K2 key must not be allowed to interrupt an SVC in a program, it is necessary to activate a STXIT routine which intercepts the K2 event.
- To avoid the SVC being terminated by test functions (e.g. AID), the program must be protected against read access. In this case, only the RESUME-PROGRAM command is allowed after the K2 key has been pressed.
- Explicit program protection can also be set in non-procedure mode (for foreground and background processes).
- Implicit program protection is only relevant in procedure mode. This option cannot be set in non-procedure mode.
- Where programs support implicit program protection, a number of actions which affect interruptions are prohibited - in conjunction with the procedure settings for INTERRUPT-ALLOWED. Compatible behavior must therefore be correctly provided in the programs themselves.
- Implicit program protection can be activated in the procedure by a switch, by a program statement or a parameter file.
- To ensure that program protection is maintained, the program should interrogate the INTERRUPT-ALLOWED setting in the procedure before each action which affects interruptions (CMD, STXIT, BKPT etc.).
- A procedure can be terminated during foreground processes, whereas a program is interrupted by /HOLD-PROGRAM or //HOLD-PROGRAM. In this case, the program can restore the implicit interruption protection by continually interrogating the CLIGET interface before requesting an action which affects interruptions.

- Implicit program protection against interruptions must be documented in the program specification. If it is not, any use of the program in uninterruptible procedures should be recorded.
- A procedure can be protected against interruption by means of a procedure-internal program which calls BKPT in a $\boxed{K2}$-STXIT routine. SDF-P program functions can be used to intercept the interruption and to restart the program.

Example

```
/SET-PROCEDURE-OPTIONS INTERRUPT-ALLOWED=*NO
/ASSIGN-SYSOUT TO=*DUMMY        "No affect on EDT for write-read"
/DECLARE-VARIABLE OPS(TYPE=*STRUCTURE),-
                 /MULTIPLE-ELEMENTS=*LIST
/LOAD-EXE FROM-FILE=*LIB(LIB=&(INSTALLATION-PATH( -
                                  /LOGICAL-ID='EDT', -
                                  /INSTALLATION-UNIT='SYSLNK', -
                                  /VERSION=*STD, -
                                 /DEFAULT-PATH-NAME='EDT')), -
                         /ELEM=EDTSTRT,TYPE=L)
/EXECUTE-CMD CMD=(SHOW-JOB-STATUS),TEXT-OUTPUT=*NONE,-
            /STRUCTURE-OUTPUT=OPS,RETURNCODE=*NONE
/SHV OPS#.PROG-FILE;SHV OPS#.PROG-NAME
/WHILE (OPS#.PROG-FILE NE '')
/   RESUME-PROGRAM
/   FREE-VARIABLE OPS
/   EXECUTE-CMD CMD=(SHOW-JOB-STATUS),TEXT-OUTPUT=*NONE,-
/   STRUCTURE-OUTPUT=OPS,RETURNCODE=*NONE
/END-WHILE
```

5.7 Internal processing of S procedures

5.7.1 Analyzing procedures

Before a procedure is executed, SDF-P first transfers it to the SDF-P procedure interpreter. The procedure head is executed. The procedure body is preanalyzed and then executed as well.

5.7.1.1 Procedure interpreter

The procedure interpreter checks the following attributes:

– Is the first character in the procedure a slash (/)?
  Procedures must begin with a command call, i.e. with a slash (/).
  If the first character is not a slash, it is not a procedure. Execution is aborted.

– Is the slash followed by a SET-LOGON-PARAMETERS (or LOGON) command?
  If so, the procedure is a batch job and is rejected.

– Is the slash followed by the BEGIN-PROCEDURE or PROCEDURE command?
  BEGIN-PROCEDURE (or PROCEDURE) introduces non-S procedures; thus, the called procedure is not an S procedure. It is processed as compatible (see “Commands, Vol. 1-5” [3]).

5.7.1.2 Preanalysis

If the analysis of the first procedure line has shown that the called procedure is an S procedure, the procedure interpreter begins processing the procedure.

S procedures are processed in the following steps:

1. If the SET-PROCEDURE-OPTIONS command is present, it is executed.
   (The SET-PROCEDURE-OPTIONS command can be called only as the first command of an S procedure.)

2. Any commands in the DECLARE-PARAMETER block are executed.
   Note: multiple calls of the DECLARE-PARAMETER command must be preceded by the BEGIN-PARAMETER-DECLARATION command and must be terminated by END-PARAMETER-DECLARATION. Any OPEN-VARIABLE-CONTAINER command in the DECLARE-PARAMETER block must also be placed between these two commands.

3. The procedure body is read in and processed:
   a) All procedure lines are read in and continuation lines are evaluated immediately, ignoring comments. The continuation character is replaced by the next line and the slash (/) at the beginning of the continuation line is deleted.
   b) Command sequences containing commands separated by semicolons (;) are divided up into individual commands.
      Exception: Command sequences in an AID command block. These commands are not processed until the AID interactive debugging aid executes the command block (for more information on the AID utility, see the “AID” manual [2]).
   c) A check is run to determine whether the block structures are syntactically correct and whether the destination tags for the GOTO, EXIT-BLOCK and CYCLE branch commands exist.

5.7.2 Procedure processing and execution

Once the entire procedure has been read in and analyzed, the procedure is started up and the individual commands are executed.

First, the expressions are replaced within the command (= expression replacement); then the command is analyzed and, finally, executed.

In the case of expression replacement, some restrictions must be noted as a result of the sequence of processing steps described here. These are described in section “Expression replacement” on page 55.

Procedure execution

The execution of S procedures is controlled within the procedures by means of control structures. The internal control of procedure execution also includes intercepting and evaluating errors so that procedure execution can then be correctly terminated or resumed. Other important aspects of procedure execution are logging and procedure interruption. Whereas control structures, logging and error handling can play an important part in both foreground and background procedures, only procedures that are called in the foreground can be interrupted.

Control structures are loops or branches that are implemented as command blocks. Each structure is initiated and terminated by a pair of associated control flow commands. Branch commands also belong to the control flow commands. The concept of control structures or control flow commands applies equally to procedures that are called in the foreground and in the background. This is described in chapter “Creating S procedures” on page 81. For a detailed description of the commands, see chapter “SDF-P commands” on page 541.

5.8 Compiled procedures

Identifying a compiled procedure

The COMPILE-PROCEDURE command writes the following identification as text in the first line of a compiled procedure:

```
/ ///COMPILED–PROCEDURE///
```

The code of the intermediate format is generated directly after this line. SDF-P-BASYS uses this identification to distinguish between compiled procedures and S procedures.

The identification also serves to signal to any other system components, utility routines or user applications that the object being read is to be considered a procedure which can be called with CALL-PROCEDURE. S procedures and compiled procedures have the following in common (see also page 131): they must start with `/<command>`, and the command may not be a SET-LOGON-PARAMETERS (or LOGON) or BEGIN-PROCEDURE (or PROCEDURE) command.

Creating a compiled procedure with variable elements

Unlike source procedures, compiled procedures cannot be modified using a text editor. Therefore end users cannot tailor compiled procedures to meet their particular needs. This is why the developer of the text procedure should include independent customization mechanisms in the procedure such as:

– use of procedure parameters
– ways of reading in variables from the terminal (using READ-VARIABLE)
– ways of reading in multiple variables from a text file (using READ-VARIABLE)
– integration of variable containers with simple variables
– integration of procedure exits for S procedures by using non-chargeable functions in the compiled procedure (e.g. with CALL-PROCEDURE or INCLUDE-PROCEDURE)
– use of job variables, etc.

Notes

– S procedures created by means of compiled procedures in list variables or with a text editor at runtime are not, themselves, compiled procedures. In other words, if compiled procedures are to be executed without the SDF-P subsystem, they can generate other S procedures which use chargeable SDF-P functions, but they cannot execute these procedures at runtime.
Similarly, no chargeable SDF-P functions can be used if a compiled procedure is interrupted during execution with the $\boxed{\text{K2}}$ key or HOLD-PROCEDURE and the SDF-P subsystem is not loaded.

- Only errors reported to SYSOUT during the analysis of SDF-P control structures are detected. This means that only the structures in the procedure are checked, but not the input commands.
- If the syntax files of the SDF-P flow control commands are modified after compilation but before execution, incompatible modifications may cause the compiled procedure to crash. The consequences are the same as for text procedures but the error is not reported when the procedure is called. Instead it is reported when the commands in the compiled procedure are executed.
- The procedure compiler stores compiled procedures in containers. These procedures can also be copied into other containers, except variable containers, without affecting their execution.
- Non-S procedures cannot be compiled.

6 Using variables in S procedures

This chapter begins with a description of the concept of variables in SDF-P. This is followed by a presentation of how variables are declared in S procedures (which also includes a description of the attributes of variables), and how variables are processed.

6.1 Variable concept

For the programming of procedures, SDF-P provides a complex variable concept that takes into account the BS2000 procedure parameters and the job variables of the job variable system.

This section provides a brief introduction to the variable concept. First the basics of the variable concept are presented, followed by an explanation of the differences between procedure parameters and S variables and, finally, a comparison of S variables and job variables.

6.1.1 Basics of the variable concept

Variables are placeholders for data which has been stored, or is to be stored, and as such are important components of procedures. They can take on various values.

Variables in SDF-P, which are also called S variables, are uniquely identified by their variable name and their scope.The scope determines where a variable can be accessed, whether in the current procedure only or throughout the entire task. Another important attribute of variables is the data type, which determines the values that a variable can assume.

A distinction is made between simple and complex variables: simple variables cannot be “divided” further, while complex variables consist of variable elements (variable elements can themselves be simple or complex variables). There are three types of complex variables: lists, arrays and structures.

Important features that contribute to programming convenience are:
– implicit variable declaration
– dynamically changing data type
– dynamic expansion of complex variables

On the other hand, runtime security is ensured through:

- explicit variable declaration
- fixed data type
- static structures

The variable concept of SDF-P takes all these features into account. A brief description of each is given in the sections below.

6.1.1.1 Simple procedure creation

S procedures have a standard format that is set up to facilitate the programmer's work with variables as much as possible, through implicit variable declaration, a dynamically changing data type and the dynamic expansion of complex variables.

Implicit variable declaration

"Implicit declaration" means that variables do not have to be declared explicitly by means of a command. They are automatically declared when they are assigned a value.

Unless otherwise specified, variables can be declared implicitly in S procedures; they are then created with standard attributes, as simple variables with a dynamically changing data type (see below) and the scope "current procedure". This means that only those variables can be accessed that are contained in the current procedure and in procedures that are called within this procedure with the INCLUDE-PROCEDURE command (for further details, see section "Scope of variables" on page 157).

Dynamically changing data type

The data type determines the values that a variable can assume, such as an integer value (data type = integer) or a random string (data type = string).
Unless otherwise specified, variables in S procedures are created with a dynamically changing data type.

"Dynamically changing data type" means that the data type is not determined when the variable is declared; it is first determined when a value is assigned. It can then change with each assignment, i.e. a variable can, for example, be assigned an integer value, a string, and again an integer value, one after the other.

The current data type can be queried by means of the predefined function CURRENT-TYPE() (for information on predefined functions, see page 364).

Dynamic expansion of complex variables

Complex variables are variables that are made up of several variable elements. Complex variables must be declared explicitly. For this explicit declaration, the variable elements do not have to be known, nor do their number of elements or their names or data types.

Unless otherwise specified, complex variables are declared as dynamically extendable in S procedures. “Dynamically extendable” means that if, during processing, these complex variables require more elements than are currently available, elements are added automatically (the complex variable is extended).

6.1.1.2 Runtime security in procedures

When working with comprehensive, complex procedures, the speed at which they are generated in the foreground is less important than runtime security, clarity and ease of maintenance. Therefore, SDF-P offers programmers the option of declaring variables explicitly with a fixed data type. In the case of complex variables of the type STRUCTURE, SDF-P permits explicit declaration of variables with a fixed number of variable elements, each with a fixed data type (= static structures).

Explicit variable declaration

“Explicit variable declaration” means that simple and complex variables are declared by means of the appropriate commands (DECLARE-VARIABLE (see page 607), DECLARE-ELEMENT (see page 594), DECLARE-CONSTANT (see page 589)). When variables are declared explicitly, unique variable attributes can also be defined at the same time.

The implicit declaration of variables can be disabled in S procedures by means of the SET-PROCEDURE-OPTIONS command in the procedure head (cf. chapter “Creating S procedures” on page 81).

Fixed data type

When variables are declared explicitly, a variable can be assigned a specific data type. Later on, this data type cannot be changed dynamically.

“Fixed data type” means that the variable can be assigned values of a specific data type only (for example, integer values only). Assignments of wrong data types are rejected and an error message is issued.

Static structures

Complex variables of the type structure can be declared as static. This means that the number of variable elements is already defined when the variable is declared and cannot subsequently be changed.

6.2 Variable declaration

Following the initial general introduction to working with variables, in the previous section, we now explain in detail how variable declaration takes place in SDF-P. Each of the following subsections deals with one aspect which must be taken into consideration in such declarations:

– The first subsection presents the different “variable types” in SDF-P, i.e. it defines the terms that apply to S variables as a whole.
– The next subsection, “Variable names”, describes variable name syntax.
– The third and fourth subsections describe the possible “data types” for S variables and the way in which S variables are initialized (“Initial value”).
– The next subsection, “Scope of variables”, describes the various scopes and their effect on the “visibility” of S variables.
– Finally, the last two subsections describe the container mechanism for S variables (“Variable containers”), and the possibility of multiple declaration.

6.2.1 Variable types

Variables in S procedures are named data objects, to which values can be assigned. They are addressed by means of their variable names. Variables in S procedures are also referred to as S variables.

A distinction must be made between two types of “variable” (or “variable types”):

– simple variables
– complex variables

Simple variables are not divisible, while complex variables are made up of one or more elements. The elements of complex variables (= variable elements) can themselves be simple or complex variables.

The term “variable element” is applied only to those elements on the highest level of a complex variable. If the variable elements themselves are complex variables, their elements are not considered to be elements of the superordinate complex variables.

There are three different types of complex variables:

– lists (= list variables)
– arrays
– structures

These complex variables differ in their variable names, internal structure and the way in which the variable elements are addressed.

The sections below first describe simple variables, then complex variables in general and, finally, the various types of complex variables (lists, arrays, structures).

6.2.1.1 Simple variables

Simple variables are uniquely named data objects that are assigned a value and whose value can be changed. They are addressed by means of a variable name. Simple variables do not contain variable elements.

Variable elements that are not themselves complex variables are also called simple variables.

Declaring simple variables

Simple variables that are not variable elements can be declared as follows:

– implicitly, the first time a value is assigned with the SET-VARIABLE command, provided that implicit declaration is allowed;
– explicitly, using the command DECLARE-VARIABLE ... TYPE = *ANY / *INTEGER / *BOOLEAN / *STRING, MULTIPLE-ELEMENTS = *NO

The way in which simple variables that are variable elements are declared depends on whether they are elements of a list, an array or a structure:

– Elements of lists are created either implicitly at declaration time of later by dynamic extension of the list. They can be addressed by a name, which is determined for each element by its position in the list.
– Elements of arrays can also be created implicitly when the array is declared or later on when the array is dynamically extended. They can be addressed by means of a separate name that is already preset when the array is declared (see page 142).
– Elements of structures can be declared implicitly if this was determined when the structure was declared. However, they can also be declared explicitly (see page 143).

6.2.1.2 Complex variables

Complex variables are variables that are made up of several elements. These elements can be addressed by means of the common variable name, i.e. the name of the complex variable.

Complex variables must generally be declared explicitly with the DECLARE-VARIABLE command. The particular operands of the DECLARE-VARIABLE command that must be used for declaration depend on whether the complex variable is a list, an array or a structure.

Arrays and lists are declared using the operand MULTIPLE-ELEMENTS = *ARRAY(...)/*LIST(...).

There are also some commands (such as SHOW-VARIABLE or EXECUTE-CMD) which implicitly create lists. Structures are declared using the operand TYPE = *STRUCTURE of the DECLARE-VARIABLE command (for this reason, structures are also known as variables with the data type STRUCTURE). The three types of complex variables are described below.

Lists

Lists are also known as list variables; this term is used particularly when there is a risk of confusing them with SDF lists (an SDF list is a string which is interpreted in accordance with the syntax rules for operand lists in commands).

Lists in SDF-P are complex variables whose elements all have the same data type. List elements can be accessed either sequentially or directly.

List elements can either be simple variables or complex variables of the type structure.

Lists have only a “relative” element name at the user interface. To generate this, the # character is appended to the list name, followed by the number of the element. The element number is the result of the sequence of elements in the list. The first list element (the list header) is always element number 1 (<name>#1). The other list elements are numbered consecutively, beginning with this list header. The element number may be omitted when addressing the list header (<name>#).

Lists can be processed sequentially in FOR loops: if the current list element contains a valid value, the FOR loop is executed. The loop is repeated until the list has been processed (for details of FOR loops, see section “Defining loops” on page 96).

List declaration

Lists are explicitly declared by means of the operand MULTIPLE-ELEMENTS = *LIST(...) in the DECLARE-VARIABLE command. The variable attributes declared in this apply for all the elements in the list.

Lists can be implicitly and dynamically extended in the SET-VARIABLE command, by appending a new element at the beginning or end of the list. However, it should be noted that there must be no gaps in the sequence of list elements.

If a restriction is to be put on the number of list elements, this must be effected by means of the LIMIT operand in the DECLARE-VARIABLE command.

Releasing a list

A single list element or a contiguous section of list elements can be released. After releasing the elements, the numbering for the list is updates so that numbering starts at 1 and continues without any gaps.

Example

```
/DECLARE-VARIABLE L,MULTIPLE-ELEMENTS=*LIST
/L=*STRING-TO-VAR('(1,2,3,4,5,6,7,8)')
/SHOW-VARIABLE L,LIST-INDEX-NUMBER=*YES
L#1 = 1
L#2 = 2
L#3 = 3
L#4 = 4
L#5 = 5
L#6 = 6
L#7 = 7
L#8 = 8
/FREE-VARIABLE L#3
/SHOW-VARIABLE L,LIST-INDEX-NUMBER=*YES
L#1 = 1
L#2 = 2
L#3 = 4
L#4 = 5
L#5 = 6
L#6 = 7
L#7 = 8
/FREE-VARIABLE *LIST(LIST-NAME=L,FROM-INDEX=4),NUMBER-OF-ELEMENTS=3)
/SHOW-VARIABLE L,LIST-INDEX-NUMBER=*YES
L#1 = 1
L#2 = 2
L#3 = 4
L#4 = 8
```

Outputting a list

It is possible to output individual list elements by means of a FOR loop or using the command `/SHOW-VARIABLE <name>#<number>`.

Arrays

Arrays are complex variables whose elements are all declared with the same data type.

Array elements can themselves be simple variables or complex variables of the type structure. They can be addressed directly by means of their array element names, which are composed of the array name and an integer array index (see section “Variable names” on page 150).

Example 1

ACCOUNT is a simple array and comprises the following elements:

```
ACCOUNT#-12
ACCOUNT#-1
ACCOUNT#0
ACCOUNT#1
ACCOUNT#2
ACCOUNT#123
ACCOUNT#1234
```

Example 2

CLIENT is a complex array. The client numbers serve as the index; each array element is a structure containing a client’s address.

```
CLIENT#123.SURNAME
CLIENT#123.FORENAME
CLIENT#123.STREET
...
CLIENT#358.SURNAME
CLIENT#358.FORENAME
CLIENT#358.STREET
...
```

Array declaration

Arrays are declared using the DECLARE-VARIABLE command, with the operand MULTIPLE-ELEMENTS = *ARRAY(...). The other variable attributes declared here apply to all the elements of an array.

Array elements can only be declared implicitly. Explicit declaration of individual elements is not possible.

Arrays are always dynamically extendable. A value range can be defined for the array index by means of the UPPER-BOUND and LOWER-BOUND operands, thus limiting the number of array elements.

Structures

Structures are complex variables whose elements can have different data types and can be accessed directly by means of their alphanumeric element names.

Simple or complex variables of the type list, array or structure can serve as structure elements.

The structure element name is made up of the variable name of the structure and a subname of the SDF data type <structured-name>; these two components are separated by a period (see section “Variable names” on page 150).

Structures can be either statically or dynamically extendable, this being specified in the declaration.

Note

Structures play an important part in structured output using structured variable streams. For detailed information, see chapter “S variable streams” on page 187.

Example

ADDRESS is a structure in which of the following simple variables are the elements:

```
ADDRESS.SURNAME
ADDRESS.FORENAME
ADDRESS.TITLE
ADDRESS.HOUSE-NO
ADDRESS.STREET
ADDRESS.CITY
ADDRESS.STATE
ADDRESS.ZIP
```

The structure elements SURNAME, FORENAME, TITLE, STREET, CITY and STATE can be declared with the STRING data type; the structure elements HOUSE-NO and ZIP can be declared with the INTEGER data type.

Structure declarations

Complex variables of the type STRUCTURE are declared using the DECLARE-VARIABLE command, in the NAME operand with TYPE = *STRUCTURE(DEFINITION=). At this point, it is also determined whether the structure is dynamically extendable or static.

Dynamic structures

A structure is “dynamic” or “dynamically extendable” if the elements are not declared explicitly by means of a structure layout or with *BY-SYSCMD.

Dynamic structures are declared explicitly using the DECLARE-VARIABLE command with the operand NAME = ...(TYPE = *STRUCTURE(DEFINITION=*DYNAMIC)).

The elements of dynamic structures can be declared explicitly or implicitly. The setting of the IMPLICIT-DECLARATION operand in the SET-PROCEDURE-OPTIONS command in the procedure head has no effect; it is always possible to declare the elements of dynamic structures implicitly.

Example 1

The dynamic structure DYN-STR is to be declared:

```
/DECLARE-VARIABLE DYN-STR (TYPE = *STRUCTURE(*DYNAMIC))
```

An element of this structure is subsequently initialized in an assignment:

```
/DYN-STR.STR2.ARR#123 = 'ABC'
```

This assignment yields the following results:

– The structure element DYN-STR.STR2 is created as a dynamic structure.
– The structure element DYN-STR.STR2.ARR is created as an array with the data type TYPE = *ANY.
– The array element DYN-STR.STR2.ARR#123 is created as a simple variable with the data type TYPE = *ANY.

Example 2

```
/DECLARE-VARIABLE S1 (TYPE = *STRUCTURE(*DYNAMIC))
```

An element of this structure is initialized in an assignment:

```
/S1.S2.ARR#1.S3 = 'ABC'
```

This assignment has the following consequences:

– The structure element S1.S2 is created as a dynamic structure.
– The structure element S1.S2.ARR is created as an array with the data type TYPE = *STRUCTURE(*DYNAMIC).
– The array element S1.S2.ARR#1 is created as a dynamic structure with the data type TYPE = *STRUCTURE(*DYNAMIC).
– The structure element S1.S2.ARR#1.S3 is created as a simple variable with the data type TYPE = *ANY.

Example 3

```
/DECLARE-VARIABLE DYN-STR (TYPE = *STRUCTURE(*DYNAMIC))
/DECLARE-ELEMENT DYN-STR.S.NUMBER(TYPE=*INTEGER)
```

In this example, the elements are explicitly declared: DYN-STR.S as a dynamic structure and DYN-STR.S.NUMBER as a simple variable with TYPE=*INTEGER.

Example 4

```
/DECLARE-VARIABLE S1 (TYPE = *STRUCTURE(*DYNAMIC))

/S1#123 = 'ABC' → Error
```

The assignment is rejected as errored, since S1 is not declared as an array.

Static structures

Static structures are structures that are not dynamically extendable. A distinction must be made as to whether the static structure was created with *BY-SYSCMD or was defined by means of a structure layout. The section below first introduces the declaration using the structure layout; it then describes the declaration of structures that are used for the output of BS2000 commands.

After the structure declaration has been terminated, no more elements can be added to the structure.

Rules for static structures

Structure declarations must be complete within a procedure, i.e. they must be terminated in the procedure in which they began. This applies to both CALL and INCLUDE procedures.

Declaration blocks can contain control flow commands or procedure calls.

For a procedure call within a structure declaration, the following must be noted:

– Procedures may be freely called within a declaration block
– If a procedure is called by means of the INCLUDE-PROCEDURE command, the elements that were declared before the call are not visible. Furthermore, no other elements of the interrupted structure can be declared in the INCLUDE procedure.
– If an incomplete structure or an incomplete structure layout is accessed in an include procedure, an error occurs.

Abortion of the structure declaration due to end of procedure:

- If a procedure is canceled while a structure is being declared, incomplete structures whose scope is PROCEDURE or INCLUDE are automatically deleted and an appropriate warning is issued.
- Incomplete structures and structure layouts whose scope is TASK are retained. The structure declaration is implicitly terminated to allow subsequent accessing of the structure or previously declared structure elements. However, no further elements can be declared.

Using *BY-SYSCMD to declare structures

If a static structure is declared with DEFINITION = *BY-SYSCMD, the declaration command for the structure must immediately follow the declaration of the structure elements.

The structure elements must be declared in a structure declaration block. This block is initiated with the BEGIN-STRUCTURE command and terminated with the END-STRUCTURE command. No structure layout name may be specified in the BEGIN-STRUCTURE command.

Consequently, the following steps are required for the declaration:

1. Declare structure explicitly.

   The DECLARE-VARIABLE command must contain the following information:

   NAME = variable name
   TYPE = *STRUCTURE(DEFINITION = *BY-SYSCMD)

2. Initiate structure declaration block.

   The DECLARE-VARIABLE command must immediately follow the BEGIN-STRUCTURE command. The command must not contain a structure layout name (NAME =) or a scope (SCOPE =).

3. Declare structure elements.

   The structure elements must be declared individually by calling the DECLARE-ELEMENT command separately for each one. Structure elements can be declared as simple or complex variables with any data type. If the structure elements are simple variables, they can be initialized with INITIAL-VALUE. All attributes that are not explicitly declared in the DECLARE-ELEMENT command are transferred from the superordinate structure. The NAME operand specifies only the element name and not the structure name.

4. Terminate structure declaration block.

   The declaration block is terminated with the END-STRUCTURE command.

Example 1

Declaration of a structure with simple variables as structure elements:

```
/DECLARE-VARIABLE M (TYPE = *STRUCTURE(*BY-SYSCMD))
/BEGIN-STRUCTURE
/   DECLARE-ELEMENT A
/   DECLARE-ELEMENT B
/END-STRUCTURE
```

Example 2

Declaration of a structure with a complex variable as a structure element:

```
/DECLARE-VARIABLE M (TYPE = *STRUCTURE(*BY-SYSCMD))
/BEGIN-STRUCTURE
/   DECLARE-ELEMENT A (TYPE = *STRUCTURE(*BY-SYSCMD))
/   BEGIN-STRUCTURE
/      DECLARE-ELEMENT B
/   END-STRUCTURE
/END-STRUCTURE
```

The static structure M now exists with the element M.A.B. Since A was also declared as a structure with *BY-SYSCMD, BEGIN-STRUCTURE had to be nested.

Rules for structures with *BY-SYSCMD

The DECLARE-VARIABLE and BEGIN-STRUCTURE commands must be in this order, one after the other.

BEGIN-STRUCTURE must not contain a layout name or a scope.

The structure elements cannot be accessed until the declaration of all elements is terminated. (Exception: the SHOW-VARIABLE command can be used to display the contents of the structure elements at any time.)

Declaring structures with named structure layouts

A structure layout can be regarded as a template with which similar structures are easily created. If a structure layout exists, it can be used in any number of DECLARE-VARIABLE commands; these then generated any number of structures with identical elements.

A “structure layout” is a sequence of SDF-P commands that begins with the BEGIN-STRUCTURE command, followed by DECLARE-ELEMENT commands, and ends with the END-STRUCTURE command.

The individual elements of structures are explicitly declared in these structure layouts. Structure layouts are identified by a unique name. This name must be repeated in the DECLARE-VARIABLE command when the structure is declared.

The individual steps required are thus as follows:

1. Initiate structure layout (= declaration block for the structure elements)

   The structure layout is initiated by the BEGIN-STRUCTURE command.
   A name must be specified in the NAME operand (NAME = layout name). This name is used later on to establish a link between the declaration of the variable “variable name” with the type structure in the DECLARE-VARIABLE command and the declaration of the elements of this structure layout.
   The structure layout must exist when the DECLARE-VARIABLE command is called.

2. Declare the structure elements

   Each layout element is declared individually by a separate call of the DECLARE-ELEMENT command.
   The elements can be declared as simple or complex variables of the type LIST, ARRAY or STRUCTURE.
   All variable attributes that are not explicitly declared in the DECLARE-ELEMENT command are transferred from the superordinate structure.
   Layout elements may not be initialized.

3. Terminate structure layout/declaration block

   The END-STRUCTURE command terminates the structure layout.

4. Declare structure explicitly

   In the operand NAME = variablename (TYPE = *STRUCTURE(...)) of the DECLARE-VARIABLE command, a layout name is specified in the parentheses (DEFINITION = layoutname) to establish the link to the previously declared structure layout.

Example

First, structure layout A is declared:

```
/BEGIN-STRUCTURE A
/   DECLARE-ELEMENT B
/   DECLARE-ELEMENT C
/END-STRUCTURE
```

A structure does not yet exist; value assignments are not possible. Afterwards, structures are declared to which structure layout A is to apply:

```
/DECLARE-VARIABLE KK (TYPE = *STRUCTURE (DEFINITION = A))
/...
/DECLARE-VARIABLE CC (TYPE = *STRUCTURE (DEFINITION = A))
```

Now there are two static structures, KK and CC, with the elements KK.B and KK.C, and CC.B and CC.C respectively. These elements can be assigned values at any time.

Rules for structure layouts

When a structure layout is declared, a name must be specified in the initiating BEGIN-STRUCTURE command.

Structure layouts must be declared before the structures.

Each structure layout is available for declaring any number of static structures.

Structure layouts have a separate name space, i.e. variables and structure layouts can have the same name.

Values cannot be assigned to a structure layout. The structure layout is simply a template for subsequent declarations.

Unlike other structure declarations, structure layouts may not be nested; i.e. each declaration block must be terminated with END-STRUCTURE before the next declaration block can be opened with BEGIN-STRUCTURE.

Note

Examples of the declarations of structures, structure layouts and structure elements will also be found under the description of the DECLARE-ELEMENT command on page 594.

6.2.2 Variable names

This section begins by describing the rules of syntax for variable names in S procedures. It then lists the words and variable names that are reserved in SDF-P, which users cannot use for their own variable names.

6.2.2.1 Variable name syntax

The following metasyntax applies:

Symbol	Meaning
:=	Definition
< >	SDF data type
m..n	Value range
...	Repetition
/	Alternative entry
[ ]	Optional entry

The SDF data types <composed-name> and <structured-name> are used for variable names and partial variable names in SDF-P in the command syntax. However, the syntax checked by SDF-P is restricted in comparison with these data types and is therefore described here in more detail.

A variable name (composed-variable-name) in SDF-P consists of one or more structure names (structured-variable-name) separated from each other by periods. The period in each case marks the name of a substructure. Blanks are not permitted within a variable name.

```
composed-variable-name:=structured-variable-name[.structured-variable-name]
```

Length	at least 1 character, not more than 255 characters

Example

```
ABCDEFGHIJKLMNO#1
A#1
A123#1
BCD.XYZ
```

A structure name consists of a partial variable name (pvn) which, if it refers to a list element or an array element, may be followed by a # and an index name.

```
structured-variable-name := pvn [# [indexname]]
```

Length	at least 1 character, not more than 20 characters
Partial variable	name (pvn):
Character set	all letters (A, ... Z) all digits (0, ... 9) @ and hyphen (-)
First character	a letter
Conventions	The string SYS at the beginning of a variable name is reserved for system variables, and should not be used. A hyphen may not be directly followed by another hyphen. The hyphen must not be the last character of the partial variable name.

Example

```
AREA-CODE
CLIENT#NUMBER
```

If the structure name refers to a list element or an array element, it is followed by a # and an index name.

```
indexname = <integer -2147483648..2147483647>  / pvn
```

Example

```
TELEPHONE#223366
TELEPHONE#INDEX
```

Element names

In the case of element names, a distinction must be made between array elements, list elements and structure elements.
The syntax that is described under <composed-variable-name> also applies to element names.

List element names

List element names consist, at the user interface, of the following components:

- list name (<composed-variable-name 1..253>)
- #
- optional: element number (<integer 1..2147483647>)

Array element names

Array element names are made up of the following components:

- array name (<composed-variable-name 1..253>)
- # (identifier for array elements when followed by additional characters)
- array index (<integer -2147483648..2147483647>)

Both the list index and the array index can also be identified by a simple variable that contains an integer value within the range of valid values.

SDF-P analyzes the string that follows the aggregate symbol # in the element name: If the first character is a digit or hyphen (= minus sign), the string is interpreted as an integer value, i.e. as a direct entry of the index.

If the first character is a letter, the string is interpreted as a variable name. SDF-P then searches for a variable with the specified name. This name must refer to a simple variable, which must be initialized with a valid integer value. If this is not the case, an error message is issued. If the variable contains a valid integer value, this value is used in the array index or list index, and the aggregate element thus identified is accessed.

Example 1

ACCOUNT is a simple array. The array index can be specified in the array element name directly:

```
ACCOUNT#123
```

However, the array index can also be supplied by means of the variable INDEX:

```
/INDEX = 236
/SHOW-VAR ACCOUNT#INDEX
```

The string INDEX is recognized as a variable name and the variable contents are evaluated. Consequently, the SHOW-VARIABLE command outputs the contents of the array element ACCOUNT#236. Note, however, that the array element ACCOUNT#236 must have been declared and initialized previously.

Example 2

The array index entry is always evaluated as an integer value. Thus, the following entries designate the same element of the array ACCOUNT:

```
ACCOUNT#123
ACCOUNT#0123
ACCOUNT#00123
ACCOUNT#NUMBER
```

A prerequisite for the ACCOUNT#NUMBER entry is that a variable NUMBER has been declared and initialized with the value 123.

Structure element names

Structure element names are made up of the following components:

– structure name (<composed-variable-name 1..253>)
– “.” (identifier for structures)
– element subname (<structured-variable-name 1..20>)

Example

The following variables are elements of the structure ADDRESS:

```
ADDRESS.SURNAME
ADDRESS.FORENAME
ADDRESS.TITLE
ADDRESS.STREET
```

6.2.2.2 Reserved words

“Reserved words” are keywords that are used as operators in expressions or Boolean constants. They must not be declared as variable names.

If they are used in a variable declaration, a variable is not created and an error message is issued.

The following keywords are reserved words:

AND	LE	OFF
DIV	LT	ON
EQ	MOD	OR
FALSE	NE	TRUE
GE	NO	YES
GT	NOT	

6.2.2.3 Reserved variable names

Variable names which begin with the prefix SYS are reserved for the transfer of data from and to system components. Of them, the variable name SYSSWITCH represents a special format.

SYSSWITCH

The variable name SYSSWITCH identifies a complex variable of type array, which can be used to address the job switch.

The SYSSWITCH array is defined as follows:

Data type	BOOLEAN
Scope	TASK (in a dialog and procedure environment)
Number of elements	32
Array index	0,..., 31
Values	TRUE = switch set to ON, FALSE = switch set to OFF

Example

Set job switch:

```
/MODIFY-JOB-SWITCHES ON = 1
/SET-VARIABLE SYSSWITCH#1 = TRUE
```

These two commands have the same effect.

Example

```
/DECLARE-VARIABLE SAVED-SYSSWITCH,MULTIPLE-ELEMENTS=*ARRAY
/SET-VARIABLE SAVED-SYSSWITCH=SYSSWITCH
/MODIFY-JOB-SWITCHES ON =( 1,4,5)
/...
/SET-VARIABLE SYSSWITCH=SAVED-SYSSWITCH
```

The array elements can be addressed by the partial names SYSSWITCH#0 to SYSSWITCH#31. Both the read mode and the write mode are permitted.

Neither the SYSSWITCH array nor the array elements can be deleted; i.e. they cannot be specified in a FREE-VARIABLE- or DELETE-VARIABLE command.

The SYSSWITCH array is always implicitly declared in every procedure.

SYSPARAM

The variable name SYSPARAM designates a variable of type string that can be used to access the program parameters passed with the START-/LOAD-EXECUTABLE-PROGRAM commands. In C programs the program parameters are accessed with the `getopt` function, and assembler programs must read in the SYSPARAM variable and evaluate it themselves using the GETVAR macro call (see page 327).

Example

```
/LOAD-EXE FROM-FILE=*LIBRARY-ELEMENT(LIBRARY=ASS.PLAMLIB,
                                     ELEMENT-OR-SYMBOL=NEUWORT4),
           PROGRAM-PARAMETERS='INPUT=DATEI-1,OUTPUT=OUT.ERGEBNIS'
%  BLS0517 MODULE 'NEUWORT4' LOADED
/SHOW-VARIABLE SYSPARAM,INF=*PAR(VALUE=*C-LIT)
SYSPARAM = 'INPUT=DATEI-1,OUTPUT=OUT.ERGEBNIS'
```

6.2.3 Data type

SDF-P distinguishes between three data types: INTEGER, BOOLEAN and STRING.

INTEGER is a numeric data type and refers to integers in the value range from -2^{31} to $2^{31}-1$.

BOOLEAN is a Boolean data type. The value range comprises the values 0 and 1. These values can be accessed by means of the reserved words TRUE, ON and YES (for 1) and FALSE, OFF and NO (for 0).

STRING is an alphanumeric data type and refers to strings that can be up to 4096 characters long.

The data type is declared in the DECLARE-VARIABLE command using the TYPE operand. In addition to the three data type designations mentioned above, the operand can be assigned the values ANY and STRUCTURE.

ANY is the default value; the data type of a variable is determined by assignment, i.e. it can change with each new assignment (dynamic data type).

STRUCTURE determines that a variable of the type “structure” is created.

The SDF structure introduced by STRUCTURE defines the layout of the complex variable “structure” and specifies whether it is (*DYNAMIC) or static and whether the element declarations follow the structure declaration (*BY-SYSCMD) or are defined by means of a structure layout.

6.2.4 Initial value

Simple variables can be initialized when they are declared by assigning them an initial value by means of the INITIAL-VALUE operand.

If the variable is declared with a fixed data type, i.e. with INTEGER, BOOLEAN or STRING, the initial value must have this same data type; otherwise, an error will occur.

If the ANY data type is specified, the value that is specified for INITIAL-VALUE determines the variable’s current data type.

Unless otherwise specified, the variables are not assigned an initial value; the user must assign this value explicitly.

If the user explicitly assigns an initial value using INITIAL-VALUE, a distinction must be made as to whether a variable is being declared for the first time or an existing variable is being redeclared. Only variables that are being declared for the first time can be initialized. If an existing variable is redeclared, the old value is not overwritten and the new initial value entry is ignored.

Variables that are not initialized (i.e. do not contain a valid value) can be “write” accessed only. Write-accessing variables means that the variable names are assigned a value either explicitly or implicitly using the SET-VARIABLE or READ-VARIABLE command.

Attempts to read-access uninitialized variables result in an error. “Read-accessing variables” means that the contents of the variable are evaluated, e.g. by means of a function, in an expression, etc.

Constant (initial) value

The DECLARE-CONSTANT command declares a write-protected variable with a constant initial value.

The variable declaration must contain a value specification for the VALUE suboperand; the specified value cannot be overwritten by subsequent assignments. This means that it is neither possible to modify the value using SET-VARIABLE nor to delete the value using FREE-VARIABLE.

6.2.5 Scope of variables

The “scope” of a variable determines how long and in which context a variable definition remains valid. It thus determines the lifetime and the visibility of the variable.

The “lifetime” of variables can be linked to the execution of either the procedure or the current task. In the first case, the variable definition becomes invalid at the end of the related procedure; it is also no longer available if the same procedure is called again later.

Specifying the scope CURRENT restricts the lifetime of the variable to the current procedure in which the variable was declared; the scope PROCEDURE links the lifetime of the variable to the last procedure called with CALL-PROCEDURE in the current call nesting hierarchy.

Variables with the scope TASK remain available until the end of the task and are thus independent of the procedure calls in this task.

The “visibility” of a variable determines whether this variable can be accessed from within a procedure. All variables declared in the current procedure are visible, regardless of their specified scope. Any other variable is visible only if

– the current procedure was called with the INCLUDE-PROCEDURE command,
– the variable was visible in the calling procedure and
– no other variable with the same name has been declared in the current procedure.

This means that variables are visible in the called procedure through (several nested) INCLUDE-PROCEDURE calls and can thus be used in the called procedure. A procedure called with CALL-PROCEDURE, in contrast, cannot “see” any variables which are not declared within this procedure. (An exception to this rule is the SYSSWITCH variable, see page 154).

6.2.5.1 Scope TASK

Task variables can be addressed from within a procedure only if this is explicitly permitted. This can be done in three possible ways:

a) Declaration of a task variable

The /DECLARE-VARIABLE ...,SCOPE=*TASK command can be used to create a task variable; after this, the variable can be addressed with the specified name in the procedure which declared it.

If a task variable with the specified name already exists, the DECLARE-VARIABLE command makes it accessible; note, however, that the other operands in the command may not conflict with the attributes of the existing variable.

b) Importing a task variable

If a procedure is to access a task variable which is known to exist, it is sufficient to include an IMPORT-VARIABLE command for this variable in the procedure.

Importing a variable is equivalent to declaring it with a DECLARE-VARIABLE command with the specification SCOPE=*TASK, but it is not possible to create a new variable. On the other hand, it saves the effort of ensuring that the attributes TYPE and MULTIPLE-ELEMENTS (including any existing substructures) specified in DECLARE-VARIABLE do not conflict with the existing values.

Example

A task variable WORKLIBRARY is to be created and initialized only if it does not yet exist; otherwise, the current procedure is to use the existing definition:

```
/IF IS-DECLARED('WORKLIBRARY',SCOPE=*TASK)
/ IMPORT-VARIABLE WORKLIBRARY “The variable may have any data type”
/ELSE
/ DECLARE-VARIABLE WORKLIBRARY(INITIAL-VALUE='#WORKLIB',TYPE=*STRING)
END-IF
```

c) Using a task variable as a container

The two possibilities described above permit a task variable to be used only under its original name from within a procedure. If this is not desired because, for example, a local variable with the same name exists in the procedure or the name of the task variable is too long and too complex (due to the requirement for unique names within a task), a task variable can be used as a “container” for a variable created within the procedure.

Example

```
/DECLARE-VARIABLE I(INIT-VALUE=0,TYPE=*INTEGER), -
/ CONTAINER=*VAR(ZAEHLER-FUER-PROZ-A,SCOPE=*TASK)
```

After this declaration, commands such as

```
/I = I + 1
```

can be used to access the task variable, whose actual name (COUNTER-FOR-PROC-A) avoids conflicts with the names of other task variables.

The task variable used as a container must already exist. For this reason, the attributes of the variable which refers to the variable container may not conflict with the declaration of the variable container (TYPE, DEFINITION, MULTIPLE-ELEMENTS).

Procedure interruption

When a procedure is interrupted, only those variables which are visible within the currently interrupted procedure remain visible.

6.2.5.2 Scopes PROCEDURE and CURRENT

In addition to task variables, there are “procedure-local” variables with the scopes PROCEDURE and CURRENT. For a procedure called with CALL-PROCEDURE, these are identical. In contrast to this, an “include procedure”, which can access the variables of the calling procedure without declaring them again or importing them, can select whether a declaration is to be effective within the scope of the caller (SCOPE=*PROCEDURE) or is to be “private”, i.e to be valid only within the include procedure itself (SCOPE=*CURRENT).

If a variable is declared implicitly, it always receives the scope *CURRENT.

Importing a procedure-local variable

A called procedure can import a procedure-local variable provided the variable has been declared in the calling procedure by means of the following command:

```
/DECLARE-VARIABLE VAR-NAME=...,SCOPE=*CURRENT/*PROCEDURE(IMPORT-ALLOWED=*YES)
```

and the called procedure uses the following command to import the variable:

```
/IMPORT-VARIABLE NAME=..., FROM=*SCOPE(SCOPE==*CALLING-PROCEDURES)
```

Example

Include procedures which use the caller's variables can be kept very simple, since it is often unnecessary to declare parameters and local variables. The following sample procedure calls an include procedure several times in order to calculate the VAT for various amounts of money and to send this VAT value to SYSOUT.

Calling procedure:

```
/ “Calculation of VAT”
/
/    “Defaults”
/ DECIMAL-CHAR = '.'; VAT-RATE = 16 “%”
/
/    “1st calculation”
/ AMOUNT = 5730
/ INCLUDE-PROCEDURE I.VAT
/
/    “2nd calculation”
/ AMOUNT = 9820 * 3
/ INCLUDE-PROCEDURE I.VAT
```

Called procedure I.VAT:

```
/    “Calculate the tax”
/ VATAX = (AMOUNT * VAT-RATE + 50) / 100
/
/    “Output the tax”
/ PENCE-OUT  = STRING(VATAX)
/ PENCE-LGTH = LENGTH(PENCE-OUT)
/ POUND-OUT  = SUBSTR(PENCE-OUT,1,PENCE-LGTH - 2) -
/           // DECIMAL-CHAR -
/           // SUBSTR(PENCE-OUT,PENCE-LGTH - 1)
/ WRITE-TEXT 'The tax is &POUND-OUT Pounds'
```

Placing the above commands in an include procedure saves calculating the tax at several different positions in the first procedure. The include procedure can access the variables DECIMAL-CHAR, AMOUNT and VAT-RATE without having to pass these as parameters.

If variable are declared within an include procedure, this procedure can decide whether these variables are to be visible in the calling procedure. Such visibility is particularly useful if the main purpose of the include call is to execute declarations.

Example

The procedure:

```
/DECLARE-VARIABLE NAME(TYPE=*STRING),SCOPE=*PROCEDURE
/DECLARE-VARIABLE COUNTRY(TYPE=*STRING,INIT='D'),SCOPE=*PROCEDURE
/DECLARE-VARIABLE ZIP(TYPE=*INTEGER),SCOPE=*PROCEDURE
/DECLARE-VARIABLE CITY(TYPE=*STRING),SCOPE=*PROCEDURE
```

can be called with /INCLUDE-PROCEDURE from several procedures in order to ensure compatible declarations for the variables NAME, COUNTRY, ZIP and CITY. If these declarations need to be modified later, it is sufficient to modify them centrally in the include procedure.

In this case, SCOPE=*PROCEDURE is specified to make the declarations effective in the procedure scope of the calling procedure and to ensure that they remain active after termination of the include procedure.

Declarations with SCOPE=*CURRENT, in contrast, are local to the include procedure. This can be used to avoid conflicts with variable definitions in the calling procedure.

Example

If the following declaration is included at the beginning of the sample procedure I.VAT on page 160:

```
/DECL-VARIABLE (VATAX, PENCE-OUT, PENCE-LGTH, POUND-OUT)
```

then the calling procedure could use a variable with one of these names without this variable being changed by the include call (SCOPE=*CURRENT is the default and does not, therefore, need to be specified in the DECLARE-VARIABLE command).

Procedure interruption

If a procedure is interrupted with the HOLD-PROCEDURE command or with the K2 key, all procedure-local variables are accessible in interactive mode, i.e. the interruption dialog belongs to the visibility scope of the interrupted procedure.

Variables which are declared during the interruption dialog are subsequently regarded as procedure-local variables of the interrupted procedure, regardless of whether this is a call procedure or an include procedure.

6.2.6 Variable containers

When variables are declared, they can be linked to a “container”. The contents of the variables are then stored in this container. Linking is done with the CONTAINER operand of the DECLARE-VARIABLE command.

The advantage of the “container mechanism” is that different variables can be linked to the same container, thus ensuring that all these variables have the same contents.

Variable containers can be either S variables, job variables or containers which are saved in PLAM libraries.

Note

A container must always be declared before the variable which refers to it.

6.2.6.1 S variables as containers

One S variable can be a container variable for another S variable. However, both of them (variable container and variable) must be declared with the same variable type and data type, and the container’s lifetime must be the same as or longer than the variable’s (the life span is determined by the scope).

There are two possible ways of defining S variables as variable containers, depending on how long the variables are to be saved for. Thus, variables can be stored temporarily or permanently by means of variable containers.

Note

Variable names are often preset in command modules or system output routines. These reserved variable names can be inserted in the CONTAINER operand as the name of the container variable. This saves the user the trouble of storing the information returned by the system in other variables that are valid in the procedure. Reserved words and variable names are listed in section 6.2.2, “Variable names”.

Variable containers for temporary variables

If a variable container is to be used for temporary storage of a variable (i.e. it is available until the task end, at the latest), the variable container need only be a variable declared in the standard manner. In this case, the contents of the variables to be stored are held by the variable container in class 5 memory.

The container mechanism can in the same way be used to make task-global variables accessible under another name, thus making it easier to avoid possible naming conflicts.

Variable containers for permanent variables

Permanent variable containers are saved in PLAM libraries.

These containers are opened (or, if they do not yet exist, they are also created) by an OPEN-VARIABLE-CONTAINER command. They are closed either explicitly by a CLOSE-VARIABLE-CONTAINER command, or implicitly by the procedure or task end (EXIT-PROCEDURE or EXIT-JOB respectively). These variable containers are saved into PLAM library elements by the SAVE-VARIABLE-CONTAINER command. In addition, SHOW-VAR-CONTAINER-ATTRIBUTES can be used to route any open variable containers into structured outputs.

When a variable container is open, the S variables which it contains can be accessed.

One way of using permanent variables is to initialize procedure parameters. This is because OPEN-VARIABLE-CONTAINER can be specified in the procedure head between BEGIN-PARAMETER-DECLARATION and the first DECLARE-PARAMETER; i.e. a variable container can be opened before the first declaration of a parameter, and hence the variables declared in the container can be used to initialize any required parameters. (The preset value “*ALL” for the AUTOMATIC-DECLARE operand in OPEN-VARIABLE-CONTAINER also ensures that by default all the variables contained in the variable container are already (pre-)declared. The scope of the variables is then that of the variable container.)

If any parameter has the same name as a variable declared by the opening of the variable container, the parameter will be rejected and the procedure call aborted.

Notes

- At the time of the OPEN-VARIABLE-CONTAINER, the library element is either locked after being read (LOCK-ELEMENT = *NO) or not (LOCK-ELEMENT= *YES); i.e. library elements can correspondingly be used either for several tasks or for one only (exclusive access).
- If a variable container is opened with LOCK-ELEMENT = *YES, all subsequent attempts to open it, in the same task or others, will be rejected.
- All variable attributes except for the scope are saved in the variable container. Specifically, these are: the type, name of the structure, array and list limits.
- The variable container is automatically closed at procedure end, with no need to specify CLOSE-VARIABLE-CONTAINER, if SCOPE=*CURRENT was set, or if SCOPE = *PROCEDURE is specified for a subordinate call. The variable container is also closed automatically at task end if SCOPE=*TASK was specified.
- However, in every case except the last the variable container is not saved when it is closed; and in the last case it will only be saved if SAVE-AT-TERMINATION=*YES was set.

PLAM library elements which are used for storing variable containers are handled as follows (see also the manual “LMS” [11] (LMS = Library Maintenance System)):

– Variable containers are held in LMS as PLAM library elements with TYPE=SYSVCONT. They can thus be used by SDF-P and FHS.
– If the specified library element does not exist at the time it is opened, then it is created at opening time if LOCK-ELEMENT=*YES is set, and when it is saved if the setting is LOCK-ELEMENT=*NO.
– If the element is locked when it is opened, it cannot be saved with SAVE-VARIABLE-CONTAINER CONTAINER-NAME=<composed-name 1..64>(ELEMENT-VERSION= *INCREMENT).
– If the element version is not specified when it is opened, and if the element does not yet exist, it is created in the highest possible version, X'FF'.
– If the element version is not specified when it is opened, but it does already exist, its highest version is opened.
– If an element version other than *UPPER-LIMIT (in LMS) is set when the element is opened, and if it does not yet exist, then after its opening and creation it can be saved with SAVE-VARIABLE-CONTAINER CONTAINER-NAME=<composed-name 1..64> (ELEMENT-VERSION=*INCREMENT).
– If an element is saved with SAVE-VARIABLE-CONTAINER CONTAINER-NAME= <composed-name 1..64>(ELEMENT-VERSION=*SAME), then if the opening setting is LOCK-ELEMENT = *YES the element will be written back into the same version, if the opening setting is LOCK-ELEMENT = *NO into the highest version.
– If an element is saved with SAVE-VARIABLE-CONTAINER CONTAINER-NAME= <composed-name 1..64>(ELEMENT-VERSION=*INCREMENT), the increment will be appended when the element is saved, and not the next time that it is opened.

Example

Input

```
/OPEN-VARIABLE-CONTAINER CONT, *LIBRARY-ELEMENT(#MY-CONT-LIB)
/SHOW-VAR-CONTAINER-ATTRIBUTES CONT
```

Output

```
CONTAINER-NAME = CONT
   FROM-FILE = *LIBRARY-ELEMENT
      LIBRARY = :1OSC:$QM123.S.152.OCG6.MY-CONT-LIB
      ELEMENT = CONT
      VERSION = *HIGHEST-EXISTING
   LOCK      = *NO
   SCOPE     = *PROCEDURE
```

Input

```
SAVE-VARIABLE-CONTAINER CONT
CLOSE-VARIABLE-CONTAINER CONT
```

For further notes on “permanent variables” see chapter “SDF-P commands”, particularly the description of the command OPEN-VARIABLE-CONTAINER on page 700.

6.2.6.2 Job variables as containers

If a variable is linked to a job variable, the contents of the declared variable are stored in the job variable. In this case, the scope with which the variable was declared makes no difference.

Linking is accomplished by means of the operand CONTAINER = *JV(jvname) with jvname as the name of the job variable.

Job variables are cataloged like files. When job variables are accessed, the same conditions apply as when files are accessed, including password protection. For information on job variables, see the “Job Variables” manual [5].

Password protection for job variables is retained in its entirety. Password-protected job variables can then be accessed by means of their linked S variables only if the relevant password was already entered in the task’s password table using the ADD-PASSWORD command.

The definition of a job variable as a variable container has no effect on variable lifetime. However, the job variable to which it is linked is not automatically deleted. It must be deleted explicitly, using the FREE-VARIABLE command with the DESTROY-CONTAINER operand.

The following restrictions apply when variables that are linked to job variables are declared:

– the variable must have the data type STRING
– the variable contents must be no more than 256 bytes long

Reason: a data type attribute is not maintained in job variables and a maximum of 256 bytes are provided for the job variable value.

The variable is created just as it is declared in the DECLARE-VARIABLE command. However, the variable contents are not stored in class 5 memory; instead, they are stored in the job variable. Consequently, when the variable is accessed, the job variable is also accessed.

In the case of job variables, it is not possible to distinguish between uninitialized job variables (= “no value assigned”) and job variables with the contents “null string” (= string with the length 0, C”). For this reason, the “uninitialized” state is currently defined as follows: a job variable is not initialized if it contains the string 256 X’EE’. For this reason, variables that are linked to job variables must not be assigned this string as a normal text string.

6.2.7 Multiple declaration

“Multiple declaration” means that a variable that was already declared in another location in the procedure is again declared. For example, the redeclaration of task-global variables in subordinate procedures would be a multiple declaration.

Multiple declarations are accepted only if the variable is declared with exactly the same attributes (except for CONTAINER and INIT) as the preexisting variable. In this case, the declaration is ignored. No new variable is created.

If the variable already exists with a container, the specification CONTAINER = *STD is sufficient for a multiple declaration. This has the same effect as the correct specification of the container.

If the variable already exists with a value, the entry INIT = *NONE is sufficient for a multiple declaration. The variables are not devalued; they maintain their values. This has the same effect as entering the initial value directly.

If the declarations do not match, error handling is activated.

This also applies if implicitly declared variables are subsequently redeclared explicitly. They must then be declared with the default values.

In this way, for example, procedure parameters can be declared as normal S variables; however, the attributes of the procedure parameters must be retained.

6.3 Variable processing

This section describes how values are assigned to variables, how variables are deleted and how variables and structure layouts are output.

6.3.1 Assignment of values to variables

“Value assignment” is the assignment of contents (i.e. a value) to a simple or complex variable.

A distinction can be made between direct value assignment with the SET-VARIABLE command and value assignment using TERMINAL with the READ-VARIABLE command. This section describes direct value assignment using SET-VARIABLE only. Value assignment with READ-VARIABLE is described in section “Input to variables” on page 175.

The SET-VARIABLE command is used to assign values to variables. The assigned value can be the contents of other variables or the results of functions or complex expressions. SET-VARIABLE can be applied to both simple and complex variables. The relevant rules are described below.

The command name SET-VARIABLE need not be stated in assignments; it can be omitted. The string to the left of the equals sign is interpreted as a variable name.

If implicit declaration is allowed for a procedure, the variable does not have to be declared (implicit declaration is only allowed for simple variables, and if IMPLICIT-DECLARATION = *YES; see the SET-PROCEDURE-OPTIONS command on page 734 or MODIFY-PROCEDURE-OPTIONS command on page 692).

If implicit declaration is not allowed, the variable must be declared before the first assignment. Assigning a value to a variable that is not declared results in an error.

In an assignment, the data type of the assigned value must always correspond to the data type of the variable to the left of the equals sign. The variable to the left of the equals sign can be assigned an arbitrary value only if it is declared with the data type ANY.

Variables that have been initialized using the DECLARE-CONSTANT command cannot be assigned a different value by means of SET-VARIABLE.

6.3.1.1 Simple variables

When a value is assigned to a simple variable, the previous value is always overwritten. The variable contents are deleted implicitly with FREE-VARIABLE; the variable is then assigned the value yielded by the expression to the right of the equals sign. Unless otherwise specified, this assignment mode is set in the SET-VARIABLE command (WRITE-MODE = *REPLACE).

6.3.1.2 Complex variables

It is possible to assign a simple value to an element of any type in a complex variable, exactly as for a simple variable.

Complex variables can also be used in assignments. The complex variables must then have the same data type on both sides of the equals sign.

When values are assigned to complex variables, in addition to the “overwrite” assignment mode (= *REPLACE), other assignment modes can also be used. These are merge (= *MERGE) and extend (= *EXTEND or *PREFIX). However, you must note which modes can be used to link complex variables in the assignment and how these modes affect the contents of the variable elements and the structure of the complex variables.

Arrays

When values are assigned to an array as a whole, an array of the same type must also be specified on the right side of the equals sign:

```
/SET-VARIABLE array1 = array2
```

For arrays, the modes “overwrite” (WRITE-MODE = *REPLACE) and “merge” (WRITE-MODE = *MERGE) apply.

*Overwrite (WRITE-MODE = *REPLACE)*

First all elements of $array_1$ are deleted. The elements of $array_2$ are then created under the name $array_1$ with the values from $array_2$.

Example

```
/DECLARE-VARIABLE array1, MULTIPLE-ELEMENTS=*ARRAY
/DECLARE-VARIABLE array2, MULTIPLE-ELEMENTS=*ARRAY
/array1#1=1
/array2#2=2
/SET-VARIABLE array1 = array2, WRITE-MODE=*REPLACE
/SHOW-VARIABLE array1
array1#2 = 2
```

*Merge (MODE = *MERGE)*

If $array_1$ and $array_2$ have identical element names, “merging” and “overwriting” are also identical, i.e. $array_1$#$index_i$ = $array_2$#$index_i$ (WRITE-MODE = *MERGE has the same effect as WRITE-MODE = *REPLACE).

If $array_1$ and $array_2$ are not identical, “merging” has an effect only if the elements of $array_1$ and $array_2$ do not have the same array index.

Elements of $array_1$ for which there is no counterpart with the same array index in $array_2$ are not affected. They are not deleted implicitly.

If $array_2$ contains elements for which there is no counterpart in $array_1$, $array_1$ is extended by these elements, i.e. the elements are placed in $array_1$ in positions which correspond to their index.

Example

```
/DECLARE-VARIABLE array1, MULTIPLE-ELEMENTS=*ARRAY
/DECLARE-VARIABLE array2, MULTIPLE-ELEMENTS=*ARRAY
/array1#1 = 1
/array1#3 = 4
/array2#2 = 2
/array2#4 = 8
/SET-VARIABLE array1 = array2, WRITE-MODE=*MERGE
/SHOW-VARIABLE array1
array1#1 = 1
array1#2 = 2
array1#3 = 4
array1#4 = 8
```

Lists

When values are assigned to a list, the assignment mode determines what must be entered to the right of the equals sign.

For lists, the modes “overwrite” (WRITE-MODE = *REPLACE) and “extend” (WRITE-MODE = *EXTEND, WRITE-MODE = *PREFIX) apply.

*Overwrite (WRITE-MODE = *REPLACE)*

A list must be specified on the right side of the assignment:

```
/SET-VARIABLE list1 = list2, WRITE-MODE = *REPLACE
```

All elements of $list_1$ are deleted. The elements of $list_2$ are then created under the name $list_1$ with the values of $list_2$.

*Extend (WRITE-MODE = *EXTEND)*

If one or more elements are to be appended to a list, the operand WRITE-MODE = *EXTEND must be included in the SET-VARIABLE command. With WRITE-MODE = *EXTEND, an expression or another list can be assigned to a list:

```
/SET-VARIABLE list1 = expression, WRITE-MODE = *EXTEND
/SET-VARIABLE list1 = list2, WRITE-MODE = *EXTEND
```

If /SET-VARIABLE $list_1$ = expression, WRITE-MODE = *EXTEND: $list_1$ is extended by one element, i.e. an element is appended to $list_1$. The results of “expression” are then assigned to this element.

If /SET-VARIABLE $list_1$ = $list_2$, WRITE-MODE = *EXTEND: $list_2$ is appended to $list_1$, i.e. the same number of elements as is contained in $list_2$ is appended to $list_1$. One after the other, these new elements are assigned the contents of the elements in $list_2$.

*Extend (WRITE-MODE = *PREFIX)*

If one or more elements are to be added at the beginning of a list, the operand WRITE-MODE = *PREFIX must be included in the SET-VARIABLE command. With WRITE-MODE = *PREFIX, an expression or another list can be assigned to a list:

```
/SET-VARIABLE list1 = expression, WRITE-MODE = *PREFIX
/SET-VARIABLE list1 = list2, WRITE-MODE = *PREFIX
```

If /SET-VARIABLE $list_1$ = expression, WRITE-MODE = *PREFIX: $list_1$ is extended by one element, i.e. a new element is inserted in $list_1$ ahead of the element that is currently first (list1 is given a new list header). This element is then assigned the value of “expression”.

If /SET-VARIABLE $list_1$ = $list_2$, WRITE-MODE = *PREFIX: $list_2$ is inserted before the element of $list_1$ that is currently the first (the list header), i.e. the same number of list elements as is contained in $list_2$ is inserted in $list_1$ before the list header. One after the other, these new elements are assigned the contents of the elements in $list_2$.

Structures

When values are assigned to a structure as a whole, a structure must also be specified to the right of the equals sign:

```
/SET-VARIABLE structure1 = structure2
```

For structures, the modes “overwrite” (WRITE-MODE = *REPLACE) and “merge” (WRITE-MODE = *MERGE) apply.

*Overwrite (WRITE-MODE = *REPLACE)*

A distinction must be made as to whether $structure_1$ is a static or dynamic structure:

If $structure_1$ is a static structure, the assignment applies only to those elements of $structure_1$ for which there are elements of the same name in $structure_2$. The contents of the elements of $structure_1$ are first deleted implicitly (implicit FREE-VARIABLE). The contents of the corresponding elements in $structure_2$ are then assigned to the elements in $structure_1$. Elements that do not have a counterpart in the other structure are ignored.

Example

```
/DECLARE-VARIABLE structur1(TYPE=*STRUCTURE(*BY-SYSCMD))
/BEGIN-STRUCTURE
/   DECLARE-ELEMENT A(INITIAL-VALUE='A')
/   DECLARE-ELEMENT B(INITIAL-VALUE='B')
/END-STRUCTURE
/DECLARE-VARIABLE structur2(TYPE=*STRUCTURE)
/STRUCTUR2.A='C'
/STRUCTUR2.B='D'
/STRUCTUR2.C='E'
/SET-VARIABLE structur1=structur2,WRITE-MODE=*REPLACE
/SHOW-VARIABLE structur1
STRUCTUR1.A = C
STRUCTUR1.B = D
```

If $structure_1$ is a dynamic structure then, just as for arrays, an implicit FREE-VARIABLE is executed on $structure_1$ and an element is then declared implicitly in $structure_1$ for each element in $structure_2$. The contents of the elements in $structure_2$ are then assigned to these new elements in $structure_1$.

*Merge (MODE = *MERGE)*

If $structure_1$ and $structure_2$ are identical, “merging” and “overwriting” are also identical (WRITE-MODE = *MERGE has the same effect as WRITE-MODE = *REPLACE).

If $structure_1$ and $structure_2$ are not identical, a distinction must again be made as to whether $structure_1$ is a static or dynamic structure.

If $structure_1$ is a static structure, the assignment first affects all elements of $structure_1$ that have the same names as elements in $structure_2$, just as it did in overwriting. The contents of these elements in $structure_2$ are then assigned to the elements in $structure_1$. However, if $structure_2$ contains further elements, $structure_1$ is not extended.

If structure$_1$ is a dynamic structure, an element is declared implicitly in structure$_1$ for each element in structure$_2$. The contents of the elements in structure$_2$ are then assigned to these new elements in structure$_1$. (WRITE-MODE = *MERGE does not have the same effect as WRITE-MODE = *REPLACE.). Elements of structure$_1$ that do not have a counterpart in structure$_2$ are not affected. The contents of the elements in structure$_2$ are assigned to the elements of structure$_1$ which have counterparts in structure$_2$.

Example

```
/DECLARE-VARIABLE STRUCTUR1(TYPE=*STRUCTURE(*DYNAMIC))
/STRUCTUR1.A='A'
/STRUCTUR1.B='B'
/STRUCTUR1.D='X'
/DECLARE-VARIABLE STRUCTUR2(TYPE=*STRUCTURE)
/STRUCTUR2.A='C'
/STRUCTUR2.B='D'
/STRUCTUR2.C='E'
/SET-VARIABLE STRUCTUR1=STRUCTUR2,WRITE-MODE=*MERGE
/SHOW-VARIABLE STRUCTUR1
STRUCTUR1.A = C
STRUCTUR1.B = D
STRUCTUR1.D = X
STRUCTUR1.C = E
/SHOW-VARIABLE STRUCTUR2
STRUCTUR2.A = C
STRUCTUR2.B = D
STRUCTUR2.C = E
```

6.3.2 Deleting and removing variables and variable declarations

The first point which should be noted here is that there is a difference between deletion and removal, just as there is between deleting the contents of a variable and deleting the declaration of a variable.

That is to say, the contents of variables are deleted, and so are variable declarations. On the other hand, elements of complex variables are removed from within the corresponding declaration, including their contents. In doing this, the contents of variables can be deleted either implicitly or explicitly: explicitly with the FREE-VARIABLE command, implicitly each time a new value is assigned. (Because before a new value is assigned to a simple variable, its old contents are deleted.)

Variable declarations can be deleted with the DELETE-VARIABLE command.

On the other hand, elements of complex variables are removed, together with their contents, by means of the FREE-VARIABLE command. It is then no longer possible to access their contents, nor can they be redeclared.

Note

In this context, it must also be noted that variable declarations and the contents of variables also depend on their lifetimes (see also section “Scope of variables” on page 157).

6.3.2.1 Deleting the contents of variables and removing elements

For complex variables, “overwrite” or “extend” can be declared in assignments. As with simple variables, “overwriting” then means that the old contents of the relevant elements are deleted before they are assigned new contents. “Extend” has no effect on the contents of existing elements; they are not deleted.

When variable contents are deleted explicitly with the FREE-VARIABLE command, note the effects on complex variables and variable containers.

Basically, it must be remembered that a FREE-VARIABLE applied to an array, a list or a dynamic structure deletes and removes all elements. In contrast, a FREE-VARIABLE applied to an element of an array, of a list or of a dynamic structure only deletes and removes this one element.

Variables that have been initialized using the DECLARE-CONSTANT command cannot be released by means of FREE-VARIABLE.

Lists

If FREE-VARIABLE is applied to a list, all list elements are deleted and removed. The declaration is retained.

If the FREE-VARIABLE refers to a list element or a section of list elements, then the list is renumbered after the deletion and removal of the list elements because the list may not contain gaps in the numbering.

Arrays

If FREE-VARIABLE is applied to an array, all array elements are deleted and removed; the declaration is retained.

If FREE-VARIABLE is applied to an array element, this element is deleted and removed.

If the array elements are themselves structures, the elements of these structures are deleted but the structure declaration is retained. In the case of dynamic structures, this means that the declaration TYPE = *STRUCTURE(*DYNAMIC) is retained; for static structures, it means that the element declarations are retained (the same applies to lists).

Structures

If FREE-VARIABLE is applied to a static structure, the elements are deleted; the structure declaration, i.e. the declaration of its elements, is retained. Correspondingly, if FREE-VARIABLE is applied to an element of a static structure, the element’s contents are deleted.

If FREE-VARIABLE is applied to a dynamic structure, the contents of the elements in this structure are deleted and all elements are removed. If FREE-VARIABLE is applied to an element in a dynamic structure, this element is removed. If the structure element itself is a complex variable, all elements of this variable (and all other subelements that may be dependent on it) are also removed.

Variable containers

Variables can be linked with other variables or job variables as variable containers. If, in such a link, the variable container is a variable, the contents of the container variable are deleted as specified in the FREE-VARIABLE command.

If the variable container is a job variable, the programmer can determine whether the job variable is deleted together with the variable when the FREE-VARIABLE command is issued. In this case, “deleting a job variable” means that the job variable’s catalog entry is deleted.

6.3.2.2 Deleting variable declarations

DELETE-VARIABLE can be used to delete a variable declaration from its current scope; i.e. including the declarations of imported task variables. The name of the variable can then no longer be used, and in addition its contents are no longer available.

The following variable declarations cannot be deleted by DELETE-VARIABLE:

– procedure parameters
– elements of complex variables
– system variables (e.g. SYSWITCH)
– container JVs
– variable containers for temporary variables
– structure layouts

6.3.3 Input to variables

Input to variables is effected by the READ-VARIABLE command. Unless otherwise specified, this command reads data from *TERMINAL and then assigns it to a variable.

6.3.3.1 Input destination

The command's VARIABLE-NAME operand refers to the variable to which new contents are to be assigned. This variable can be a simple or a complex variable. Unless otherwise specified, for each assignment the old variable contents are first deleted and then the new contents are assigned.

Specifying variable names directly

The input destination can be specified in the VARIABLE-NAME operand directly as the variable name of a simple variable or the variable name of a structure.

If NAME designates a simple variable, this variable must be overwritable. If the specified variable is not a variable element and has not yet been declared, it is declared implicitly with SCOPE = *CURRENT and the data type ANY, provided that the implicit declaration of variables is allowed.

If NAME designates a variable element that does not yet exist, it is again declared implicitly, provided that the superordinate complex variable exists and can be extended.

A number of variable names (up to 2000) can be specified in the form of a list: they should then be enclosed in parentheses: (varname$_1$, varname$_2$, ...). The rules described above apply to each of these variable names.

When variables are specified in this type of list, it must be possible to assign a value to each of these variables. If the input is shorter than this list (i.e. if values cannot be assigned to all the variables), error handling is activated.

Creating variables implicitly

Variables can be created implicitly in the VARIABLE-NAME operand of the READ-VARIABLE command, depending on the input (VARIABLE-NAME = *BY-INPUT(..)). This entry requires that variables be read in the format that is generated by the SHOW-VARIABLE command with its operands when variables are output. This means that a variable name and variable value are read in.

If simple variables that already exist are read in, they are assigned a value.

If the simple variables are variable elements, the following distinctions must be made:

– If a list element is read in, the command is rejected.
– If a structure element is read in whose superordinate structure has not yet been declared, the structure is created implicitly with SCOPE = *CURRENT.
  If a static structure is read in that does not yet exist in the procedure, it is recreated, not as a static structure, but as a dynamic structure.

The following applies to reading in both simple and complex variables: if a variable is read in that is not yet initialized (contents *NO-INIT) (i.e. that has not yet been assigned a value), the corresponding variable in the procedure is deleted implicitly (implicit FREE-VARIABLE).

Reading into a list variable

The name of a list variable can be specified in the VARIABLE-NAME operand of the READ-VARIABLE command. If a list with this name does not yet exist in the procedure, a list is created implicitly (if allowed) with SCOPE = *CURRENT and the data type TYPE = *STRING.

The list can be overwritten (WRITE-MODE = *REPLACE) or extended (WRITE-MODE = *EXTEND).

“Overwrite” means that the old list contents are first deleted. The values read in are then assigned to the list elements one after the other, beginning with the list header. “Extend” means that the old list contents are retained. With each assignment, the list is extended by one element, i.e. an element is appended to the end of the list and is then assigned the value read in.

6.3.3.2 Input source

The input source is defined in the INPUT operand. By default, the data display terminal *TERMINAL is set as the input source. However, this input source can also be defined as:

– the terminal
– a cataloged file
– a variable
– a library element
– the SYDTA system file

The way in which values that have been read in are assigned to a variable depends on the entries in the VARIABLE-NAME operand (see page 705).

Terminal

The inputs are read from the terminal (= *TERMINAL). Each input that is terminated with the transmit key is then regarded as a variable value.

The suboperand SECRET-INPUT provides the additional possibility of secret input from a terminal into a protected (blanked) field.

User file / library element

User files and library elements differ only in the way in which they are stored and, thus, in the way in which their names are entered. If a user file or library element is specified as an input source, each data record is interpreted as a value that is assigned to a variable.

In the case of ISAM files, it is possible to specify whether the ISAM key is to be retained or removed at the time of input. Unless otherwise specified, the ISAM key is removed and only the remainder of the data record is assigned to a variable.

List variable

If a list variable is specified as the input source, this list must be declared with the *STRING data type and have valid contents. (The list can also be declared with the data type *ANY if it only contains string values.)

Each element of this list is then valid as a value that can be assigned to a variable.

SYSDTA system file

Data is read from SYSDTA exactly as it is from a file. However, there is one exception: If the read operation is terminated before SYSDTA EOF, the next SYSDTA record is accessed the next time data is read in from SYSDTA. This allows reading of specific data from SYSDTA. This is useful, for example, if only a single simple variable is to be read.

Input ends at the end of the SYSDTA system file, i.e. either at the end of the file if a cataloged file is assigned to SYSDTA, or with the next command if SYSDTA is assigned to SYSCMD (default value for S procedures).

It is not possible to interrupt the input of data read from SYSDTA by means of HOLD-PROGRAM, K2 or BEGIN-BLOCK PROGRAM-INPUT=*MIXED-WITH-CMD (in order to switch to command mode). These commands and actions only result in the termination of input, i.e. SYSDTA EOF (end-of-file) being reported to the appropriate command. Only the SEND-DATA command does not terminate the READ-VARIABLE command.

6.3.4 Output from variables

Variable contents can be output with the SHOW-VARIABLE command. SHOW-VARIABLE can applied to several variables simultaneously; unless otherwise specified, output is routed to SYSOUT.

6.3.4.1 Output source

Two operands serve to determine the variable(s) to which the command call is to apply: the variable name in the VARIABLE-NAME operand and, as an additional “search” criterion, the scope in the SELECT operand. This means that, in addition to querying the variables that are declared in a particular procedure, the programmer can also obtain information on which variables are visible during a job.

The nature of the output is controlled by a single operand: INFORMATION. This determines: first, the data type of the output; and secondly, whether the output is simply the variable values, or whether the variable names are to be output in addition. If a variable has been declared but not yet initialized, the value output for it is the string ’*NO-INIT’ (however, this only applies to INFORMATION = *PAR(VALUE = *C-LITERAL or VALUE = *X-LITERAL or VALUE = *WITHOUT-QUOTES)).

6.3.4.2 Output destination

The output destination is defined by the OUTPUT operand. It can be:

- the SYSOUT (default setting) or SYSLST system file
- a user file or library element
- a list variable

When variables are output, each variable value corresponds to an output line.

User file or library element

If output is rerouted to a user file or library element, each set of variable contents becomes a data record.

For files, the user can determine whether they are to be overwritten or extended. “Overwrite” means that the contents of the file are deleted. The first set of variable contents that is output becomes the first data record in the file. “Extend” means that the new data records are appended to the end of the file.

List variable

The output of variable contents can be rerouted to a complex variable of the list type. This list variable must not be both the source and destination of output.

If the list variable does not yet exist, it is declared implicitly with the data type *STRING and SCOPE = *CURRENT.

If the list variable exists, it is either overwritten or extended, depending on the supplementary operand WRITE-MODE.

“Overwrite” means that the old list element contents are deleted and the new contents are assigned to the list elements one after the other. One of the variable values that is output is assigned to each list element.

“Extend” means that the list variable is dynamically extended. For each variable value that is output, a new list element is appended to the list.

6.3.4.3 Structure layout output

The SHOW-STRUCTURE-LAYOUT command has a structure similar to that of the SHOW-VARIABLE command. However, it does not output the contents of variables; instead, it displays the structure of structure layouts, i.e. it relates to the structure of explicitly declared, static structure layouts.

Structure layouts can be displayed only with SHOW-STRUCTURE-LAYOUT. Since values cannot be assigned to the elements of structure layouts, SHOW-STRUCTURE-LAYOUT displays only the element names declared in the specified layout.

In the case of the SCOPE operand, note that task-global structure layouts are always visible.

Otherwise, the same information applies to SHOW-STRUCTURE-LAYOUT as described for SHOW-VARIABLE.

6.3.5 Converting SDF command strings to S variables and vice versa

SDF command strings can be converted to structure or list type S variables according to very specific rules. Conversely, the conversion of structure or list type S variables into SDF strings is equally possible.

To convert an SDF command string into an S variable the operand *STRING-TO-VARIABLE(...) should be set in the SET-VARIABLE command (see page 740), to effect the reverse conversion use the predefined function VARIABLE-TO-STRING( ) (see page 532).

Note

It is not possible to specify positional operands for the conversion in SET-VARIABLE. Doing so results in an error message.

Conversion rules

Converting SDF command strings to S variables

1. Operands are converted to structure elements.
2. Simple operand values are converted to simple structure element values.
3. SDF lists are converted into list variables.
4. The lists are converted regardless of the SDF data types.
   The following applies by default (VALUE-TYPE = *STD): If an integer value is found in the input string, then “integer” is stored as the data type for the variable. If the value TRUE or FALSE is found (in capital or small letters), then “boolean” is stored as the data type for the variable. In all other cases, “string” is stored as the data type for the variable. When VALUE-TYPE = *STRING is specified, “string” stored as the data type for the variable regardless of its value.
5. Command/statement names are converted to element values with the reserved name SYSOPER.
6. Values which initiate a structure are converted to element values with the reserved name SYSSTRUC.
7. Operands in SDF structures are converted to second-level structure element names. These elements are appended to the element name which results from the conversion of the operand, and which in turn identifies the structure.

Converting S variables to SDF command strings

The conversion of S variables of type string to character strings is similar to & replacement. Integer values are automatically converted to strings, Boolean values are converted to the string value ‘TRUE‘ or ‘FALSE‘ as applicable.

Thus, an S variable must be initialized as follows if its value after conversion from an S variable to an SDF string is to be a value of type C-string:

```
/SET-VARIABLE DATA.OPER = 'C''mychain'''
```

This has the effect that the string value 'C''mychain'''is saved in the variable DATA.OPER, and on conversion is transformed into the following SDF syntax:

```
OPER = C'mychain'
```

This results in the value “C'mychain'” being specified with the SDF data type <C-string> in accordance with the OPER operand.

Summary:

SDF syntax string	Complex S variables (name DATA)
OPER = value	DATA.OPER = 'value'

operation oper1 = val1	DATA.SYSOPER = 'operation'
	DATA.OPER1 = 'val1'

oper = struc (oper1 = val1)	DATA.OPER.SYSSTRUC = 'struc'
	DATA.OPER.OPER1 = 'val1'

oper = (val1,val2,val3)	DATA.OPER#1 = 'val1' DATA.OPER#2 = 'val2' DATA.OPER#3 = 'val3'

Exceptions

Only the external form of SDF strings will be parsed according to these rules. The basic data items processed are the characters strings. Neither the semantic information nor the internal SDF syntax descriptors are converted and stored as S variables: e.g. if OPER = A (OP1=VAL1) is stored in an S variable, this gives no indication whether A is a keyword, a name or a filename.

Restrictions on the string input

The SDF string FROM=(file,*LIB(LIB=lib,EL=elem)) cannot be converted to an S variable, because SDF-P does not support any form of heterogeneous list.

Examples (showing the restrictions)

Example 1

FROM=(file,*LIB(LIB=lib,EL=elem)) is converted to:

```
A.FROM#1.SYSSTRUC = 'file'
A.FROM#2.SYSSTRUC = '*LIB'
A.FROM#2.LIB = 'lib'
A.FROM#2.EL = 'elem'
```

Such complex variables are possible for dynamic structure lists. However, the first element of the list must be converted as a SYSSTRUC element, which conflicts with the actual structure of the SDF string. A.FROM#1 = 'file' is impossible.

Example 2

OP = (a,b,c) is converted to:

```
A.OP#1 = 'a'
A.OP#2 = 'b'
A.OP#3 = 'c'
```

On the other hand, OP = (a,b,c(OPR=d)) is converted to:

```
A.OP#1.SYSSTRUC = 'a'
A.OP#2.SYSSTRUC = 'b'
A.OP#3.SYSSTRUC = 'c'
A.OP#3.OPR = 'd'
```

Note

The conversion of an individual value depends on the structure of the input string, and may therefore be rejected (except for VALUE-TYPE= *STRING).

Example 3

The SDF strings FCB-TYPE=ISAM and FCB-TYPE=ISAM(KEY-POS=5,KEY-LEN=8) result in two different structures in SDF-P:

```
DATA.FCB-TYPE = 'ISAM'
```

and

```
DATA.FCB-TYPE.SYSSTRUC = 'ISAM'
DATA.FCB-TYPE.KEY-LEN = 8
DATA.FCB-TYPE.KEY-POS = 5
```

Hence, in S procedures the two variable structures can be sorted by means of the predefined function VARIABLE-ATTRIBUTE(..., ATTRIBUTE=*TYPE). For example, for VARIABLE-ATTRIBUTE('DATA.FCB-TYPE', ATTRIBUTE=*TYPE) this gives the results:

– '*STRUCTURE', if the operand value is a structure.
– in any other case, '*ANY' or '*STRING'

Example 4

The strings OPER=A(OP1=X,OP2=Y) and OPER=B(OP1=X,OP2=Y) create the same structure elements, but OP1 means something fundamentally different in the two cases:

```
DATA.OPER.SYSSTRUC = 'A'
DATA.OPER.OP1 = 'X'
DATA.OPER.OP2 = 'Y'

DATA.OPER.SYSSTRUC = 'B'
DATA.OPER.OP1 = 'X'
DATA.OPER.OP2 = 'Y'
```

6.3.6 S variables and procedure parameters

Procedure parameters in BS2000 are parameters which are passed from the caller to the procedure when the procedure is called. They serve to pass information from one procedure to another

Both S and non-S procedures can use procedure parameters.

In S procedures, procedure parameters are created and processed as S variables. However, the attributes of procedure parameters are not quite the same as for “normal” S variables; they differ from the latter in their function, location and the commands used to declare them, and also in that not all variable attributes apply to procedure parameters.

The following table shows the differences and common features.

	S variable	**Procedure parameter**
Location of declaration	Procedure body	Procedure head
Command used for the declaration	DECLARE-VARIABLE DECLARE-ELEMENT	DECLARE-PARAMETER
Variable type	Simple variable Complex variable	Simple variable
Permissible data types	ANY STRING INTEGER BOOLEAN	ANY STRING INTEGER BOOLEAN
Scope	Explicitly definable: current procedure task	Implicitly definable: Current
Container	As variable container: S variable job variable	No container

Whereas, in non-S procedures, procedure parameters can only be used at procedure call to pass values to the called procedure, in S procedures procedure parameters can be used in addition to access the contents of variables in superordinate procedures. For this to be possible, either the procedure must be called by an INCLUDE-PROCEDURE or the variables concerned must be task-global variables. (For further details, see section “Scope of variables” on page 157.)

The link between variables and procedure parameters is effected by declaring the procedure parameters in a DECLARE-PARAMETER command, with TRANSFER-TYPE = *BY-REFERENCE. The string that is transferred in the procedure parameter is then interpreted as a variable name. In the called procedure, the variable from the calling procedure is then accessed. (For further details see section “Passing procedure parameters” on page 106)

6.3.7 Job variables and S variables

Job variables are a component of the chargeable software product JV (Job Variables). Only if JV is loaded is it possible to access the job variables. (The Job Variables system and the job variables themselves are described fully in the “Job Variables” manual [5]).

Job variables are utilized in both non-S and S procedures.

The use of job variables is the same in both S and non-S procedures; that is, to synchronize batch jobs, i.e. procedures which are started in the background

The links between variables and job variables are defined in the variable declarations (CONTAINER operand). The S variables must be of data type STRING, and must not be longer than 256 bytes.

Job variables which have not yet been explicitly initialized contain the value 256 X’EE’, so that they can be distinguished from empty strings (strings of length 0).

Job variables are managed like files, by means of a catalog entry. Like files, they can also be protected by a password. To allow such password protected job variables to be accessed, for example when a job variable is defined as a variable container, the password must be entered in the password table for the job by means of the command ADD-PASSWORD. Only then is it possible to call the SDF-P command which accesses the job variable.

Job variables also play an important role in expression replacement. This is described in detail in section “Expression replacement” on page 55.

The table on the next page gives a comparison between S variables and job variables.

Note

It is possible to query whether S variables have been initialized using a predefined function: IS-INITIALIZED( )

Attribute	**S variable**	**Job variable (JV)**
Name SDF data type Length Character set	 <composed-name> 1..255 A...Z, 0...9, #, @, -, .	 <filename> 1..54 A...Z, 0...9, $, #, @, -, .
Permissible data types (variable contents)	STRING BOOLEAN INTEGER	STRING

Attribute	**S variable**	**Job variable (JV)**
Maximum size (= field length)	4096 bytes (STRING)	256 bytes
Variable formats	Simple variable Complex variable	User JV Special JV Monitoring JV
Scope	Task Procedure Include (permanent container in library)	Task (temporary JV); JV system (permanent JV)
Declaration	Implicitly from assignment Explicitly by SDF-P command	Explicitly by command (user JV)
Initialization	As part of declaration	-
Value assignment	As part of declaration Explicitly by assignment	Explicitly by assignment (user JV)
Deletion	Automatic at procedure or task end Explicitly by command	Automatic at task end (only temporary JVs) Explicitly by command

7 S variable streams

S variable streams make it possible - in addition to providing screen-oriented SYSOUT outputs - to transfer highly structured information which is then routed to S variables or passed to an output server for further processing (e.g. to FHS for output in FHS masks).

7.1 The concept of S variable streams

The concept of S variable streams is based on the interaction between client and server. As a variation on the classical client/server model, there are the following three connections in the SDF-P context:

– On the client side, the user sends S variables (created either by the system, or by a DECLARE-VARIABLE or SET-VARIABLE command) to the server, and/or receives variables sent back by the server.

– The server - FHS or SDF-P itself - processes these S variables as necessary for the client’s requirements. Thus, for example, FHS can be requested to output the S variables received from the client to screen masks, and to update them. If, in addition, the client expects an answer, FHS will later send these or other variables back to the client.

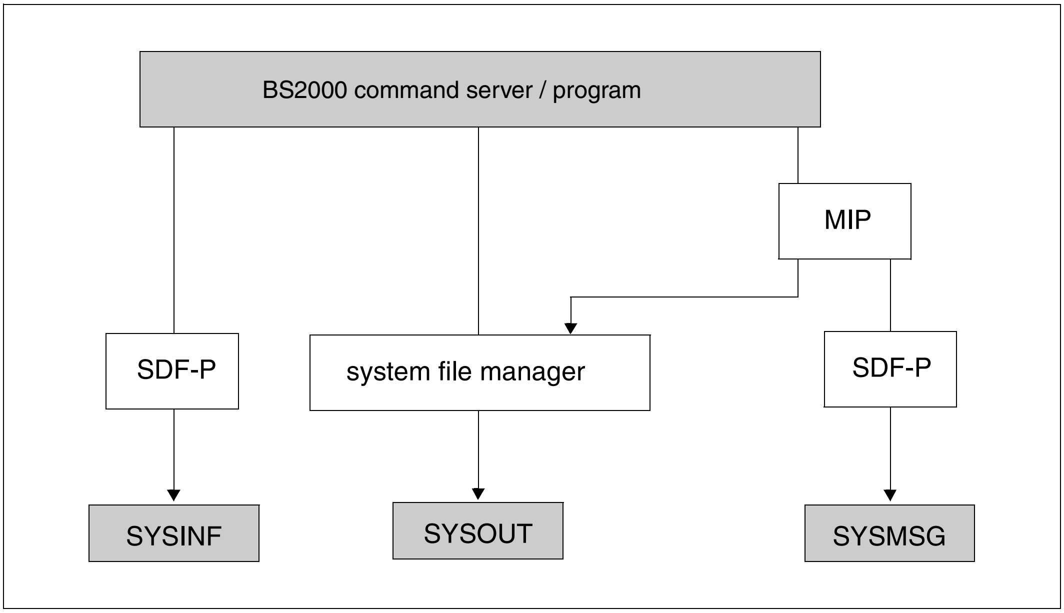
– SDF-P has - in addition to the optional task of server - the role of controller or router. Thus, for example, SDF-P is responsible for establishing the server’s link to the client at runtime, because this link is not a fixed one; instead it can be chosen dynamically, to ensure that the client code has the maximum possible independence from the server.

7.1.1 Functional scope

- Output from BS2000 SHOW commands to S variables:
  Output from SHOW commands is rerouted to S variables. These serve as filters between BS2000 command and output servers. Thus, for example, the variable values can be inserted by the FHS output server in interactive screen masks for convenience of use. (For further details, see also the “FHS” manual [19].)
- Re-use of output data as input data for later commands:
  Data output by commands and stored in S variables can be used as input data for later commands. Not only can simple variables replace the values of individual operands in this way, but in the same way structure-type variables can replace operand structures, or even a complete command or statement.
- Selecting the output server at runtime:
  The use of a dynamic selection mechanism for applications and S procedures means that the output server (e.g. SDF-P or FHS) does not need to be specified until runtime.
- Alternative usage of FHS
  In the context of SDF-P, FHS is available as a server at both program and command level. Here, S procedures can be used to create applications for FHS.
- Redefinition of EXECUTE-CMD command jobs:
  While ASSIGN-STREAM is used to create basic assignments, the EXECUTE-CMD command enables alternative assignments to be made for the default variable stream SYSVAR.
- Complementary or alternative outputs to SYSOUT and to S variables:
  S variables, or S variable streams, can be used as alternatives to the output to SYSOUT, or to complement it. The default setting for ASSIGN-STREAM is complementary output.

7.1.2 S variable streams SYSINF, SYSMSG and SYSVAR

The diagram below shows both output to SYSOUT by the system file manager and output rerouted to the S variable streams SYSINF and SYSMSG. The combination of the SYSINF and SYSMSG streams is referred to as SYSVAR.



Command and program output to S variable streams SYSINF and SYSMSG or to SYSOUT

SYSINF

The S variable stream SYSINF contains the structured output of (SHOW) commands and programs. See the manual “Commands, Volume 6” [4] for more details on the output formats.

SYSMSG

The S variable stream SYSMSG contains the structured output of guaranteed messages. See the manual “MSGMAKER” [21] for more details on message output in S variables.

SYSOUT

The output stream SYSOUT typically contains all outputs from command servers and utilities. The output is edited for screen display.

7.1.3 Assigning S variable streams

Normally, an S variable stream will be assigned to an output destination by means of an ASSIGN-STREAM command. The output destination may be a server or an S variable.

The EXECUTE-CMD command implicitly assigns the S variable stream SYSVAR to an S variable. This assignment is, however, only temporary and only applies to the specified command.

Assignment using ASSIGN-STREAM

This command has two main operands: STREAM-NAME and TO.

The STREAM-NAME operand

The STREAM-NAME operand is used to specify the name of an S variable stream which is to be assigned. For this purpose, the system provides three predefined S variable streams, each of which begins with the prefix SYS: SYSINF, SYSMSG, SYSVAR. However, a user-specific S variable stream, or one with a user-specific name, may also be specified. The specification of user-defined S variable streams is useful, for instance, if FHS applications are to be used.

The TO operand

The TO operand is used to specify the output destination or the server which is to be linked to the S variable stream. It is possible to make the following specifications:

– TO = *STD
  This is the default value for the TO operand. The following table shows the operand values to which *STD corresponds, depending on the specification for the STREAM-NAME operand:

STREAM-NAME=	TO=*STD	Information transferred
SYSINF	SYSVAR	Structured outputs from commands and programs
SYSMSG	SYSVAR	Structured guaranteed messages
SYSVAR	*DUMMY	Structured outputs from commands and programs, or structured messages
<structured-name 1..20>	*DUMMY	User variable stream

  Unless otherwise specified, the variable stream for system output and guaranteed messages is SYSVAR, to which in turn *DUMMY is assigned by default. Thus, SYSOUT and SYSVAR outputs coexist. (If output to SYSOUT is to be suppressed, the assignment ASSIGN-SYSOUT TO-FILE=*DUMMY must be made.)

- TO = <structured-name 1..20>
  TO = Name of a (user-specific) server.
- *DUMMY
  Specifying *DUMMY means: no assignment.
- TO = *SAME-AS-CALLING-PROC
  Specifying *SAME-AS-CALLING-PROC means: the assignment specified in the calling procedure remains valid.
- TO = *VARIABLE(...)
  Defines the specified S variable as the output destination. This indirectly assigns SDF-P as the server. It is possible to specify control variables for data exchange with the server. All variables specified must be declared as structure-type list variables. The data items will be processed in the following order:

TRANSMIT-BY-STREAM	**Direction**	**ASSIGN-STREAM**
(1) VARIABLE-NAME	⟶	VARIABLE-NAME
(2) CONTROL-VAR-NAME	⟶	CONTROL-VAR-NAME
(3) RETURN-VARIABLE-NAME	⟵	RETURN-VARIABLE-NAME
(4) RET-CONTROL-VAR-NAME	⟵	RET-CONTROL-VAR-NAME

  If an error arises during one of these operations, then this operation - and all subsequent ones - will be terminated. The transmission will then be aborted, with the time of error as the status.
- TO = *SERVER(...)
  Name of a server, e.g. FHS. Additional information such as name of the FHS format library can be passed to the server.

Notes

- Even if the assignment is SYSVAR, the different information for SYSINF and SYSMSG can still be subject to separate subsequent processing.
- Any specification of a system file (SYSDTA, SYSCMD, SYSOUT, SYSLST, SYSOPT, SYSIPT) for STREAM-NAME will be rejected.
- If the S variables specified in *VARIABLE(...) are incompletely declared at the time of the ASSIGN-STREAM assignment, the assignment will be rejected.
- If the variables are incompletely assigned at a point after the ASSIGN-STREAM assignment, e.g. because they have in the meantime been the subject of a DELETE-VARIABLE command, the next transmission will be rejected with a warning, and SDF-P will set the variable stream to *DUMMY.

- If the same S variable is assigned to two different variable streams, the data items from the two variable streams will be processed in the order that they are transmitted.
- S variables can be changed between two transmissions. The next transmission will take account of this change.

Example

```
/DECLARE-VARIABLE OPS-VAR(TYPE=*STRUCTURE),MULTIPLE-ELEMENTS=*LIST
/ASSIGN-STREAM SYSINF,TO=*VARIABLE(OPS-VAR)
/ASSIGN-SYSOUT TO=#ERROR-SYSOUT
```

7.1.4 Using S variable streams to transmit S variables

In S procedures, the transmission of S variables is controlled by means of the command TRANSMIT-BY-STREAM. This command is used at the client side to control the transfer of variables to and/or from the specified server via a selected S variable stream.

The STREAM-NAME operand is used to specify the name of the selected variable stream for this activity.

The remaining operands, i.e. VARIABLE-NAME, RETURN-VARIABLE-NAME, CONTROL-VAR-NAME and RET-CONTROL-VAR-NAME, are used to specify which variables are to be transmitted to or from the server, and correspond to the settings for ASSIGN-STREAM ...,TO=*VARIABLE( ). That is, as for ASSIGN-STREAM, the variables specified here must be declared as structures. Likewise, as far as the processing sequence of the data items is concerned, the variables specified for VARIABLE-NAME and CONTROL-VAR-NAME take precedence over the others.

For further notes (particularly about the default header) see the TRANSMIT-BY-STREAM command on page 783.

Example

```
/DECLARE-VARIABLE A1(TYPE=*STRUCTURE),MULTIPLE-ELEMENTS=*LIST
/DECLARE-VARIABLE A2(TYPE=*STRUCTURE)
/ASSIGN-STREAM SYSINF,TO=*VARIABLE(A1)
/TRANSMIT-BY-STREAM SYSINF, VARIABLE=A2, RETURN-VARIABLE=*NONE
```

7.1.5 Showing S variable stream assignments

The SHOW-STREAM-ASSIGNMENT command can be used to request the display of the current assignment of the specified S variable stream(s) (for further details refer to page 750).

The STREAM-NAME operand is used to specify which variable stream(s) should be displayed. If the value specified is *STD-STREAMS, then all the standard variable streams will be output; i.e. all the S variable streams with the prefix “SYS”.

The INFORMATION operand is used to output details of the assigned server. The default setting of *CURRENT-ASSIGNMENT outputs what is specified in the TO operand of the ASSIGN-STREAM command. If *FINAL-DESTINATION is specified, the current server name is output.

The OUTPUT operand is used to specify where the output is to be sent to. Unless otherwise specified it is sent to SYSOUT, and in parallel to the variable specified in the ASSIGN-STREAM command. If *SYSLST is specified, the output will be sent only to SYSLST, and not to any variable.

Example

Input

```
/DECLARE-VARIABLE VAR1(TYPE=*STRUCTURE),MULTIPLE-ELEMENTS=*LIST
/ASSIGN-STREAM SYSINF,TO=*VARIABLE(VAR1)
/SHOW-STREAM-ASSIGNMENT SYSINF
```

Output

```
STREAM-NAME = SYSINF
 ASSIGN-LEVEL = 0
 DESTINATION = *VARIABLE
    VARIABLE-NAME = VAR1
        VAR-MODE = *EXTEND
    RETURN-VARIABLE-NAME = *NONE
    CONTROL-VAR-NAME = *NONE
    RET-CONTROL-VAR-NAME = *NONE
```

7.1.6 Deleting S variable streams

The DELETE-STREAM command is used to delete S variable streams. The variable streams to be deleted can be specified in the STREAM-NAME operand explicitly in a list or using a search pattern.

The effect of deletion is that the variable stream is set as *DUMMY, and can no longer be used as either an assignment destination (ASSIGN-STREAM TO=...) or a transmission destination (TRANSMIT-BY-STREAM).

The only S variable streams which can be deleted are those on the highest level (external to procedures), or in procedures called from that level if SYSTEM-FILE-CONTEXT= *SAME-AS-CALLER is set.

It is never possible to delete variable streams with reserved names.

Variable streams in procedures with unreserved names are implicitly deleted on exiting from the procedure unless SYSTEM-FILE-CONTEXT=*SAME-AS-CALLER is set in the SET-PROCEDURE-OPTIONS command.

7.2 Structured output in S variables

SHOW commands can output their information in complex S variables of the type ’structure’. This allows the user to access specific information directly. Every SHOW command with this functionality predefines the layout of the structure:

– A structure is defined for an object specified in the SHOW command (e.g. a file or device). If more than one object is specified (e.g. as a wildcard search pattern), a list of structures is created.

– For each specific item of information on this object, an S variable is defined as an element of this structure and the specific information is assigned to it as the content.

– If information on an object can be organized in a hierarchical structure, a complex S variable is defined for each hierarchy as an element of the superordinate structure. A hierarchically lower-ranking S variable can therefore be a simple S variable, a structure or a list of simple S variables and/or structures.

– The names of the elements are command-specific and predefined for each SHOW command. As far as possible, they match the corresponding operand names or a unique abbreviation. Elements containing the same information are given the same name across all SHOW commands. The names of the S variables are preset for each SHOW command and are guaranteed for future versions.

– As far as possible, the contents of the S variables (the specific information) match corresponding operand values or unique abbreviations.

– The S variables have a defined type: string, integer or Boolean.

Information about the layout of the various output structures is contained in the command descriptions in the relevant product manual. The output structures for the BS2000/OSD-BC SHOW commands can be found in the “Commands, Volume 6” manual [4].

7.2.1 Method

1. **Declare S variable**

   The user declares a list variable of the type ’structure’. The structure should be dynamically expandable (default).

   ```
   /DECLARE-VARIABLE NAME=USERVAR(TYPE=*STRUCTURE),MULTIPLE-ELEMENTS=*LIST
   ```

   If, on the other hand, the structure is created statically, the user can receive only specific information for which he/she has explicitly declared structure elements with the defined names.

2. **Create structured output**

   EXECUTE-CMD command

   For structured output of a *single* command, the user calls a SHOW command via the EXECUTE-CMD command, specifying that the structured system output is to be routed to the declared S variable.

   ```
   /EXECUTE-CMD CMD =( SHOW-SYNTAX-VERSIONS SOFTWARE-UNIT-NAME=(JV, LMS) ), -
                /STRUCTURED-OUTPUT = USERVAR ,TEXT-OUTPUT = *NONE
   ```

   Specification of TEXT-OUTPUT=*NONE suppresses output to SYSOUT.

   If output is directed to a variable with a static structure, only already existing elements of the structure are assigned a value. If there is no value for an element, it is assigned a default value that depends on its type:

   ```
   INTEGER: 0
   STRING:  '' (null string)
   BOOLEAN: FALSE
   ```

   If none of the existing element names match any of the defined names, no values are assigned; no error message or warning is returned. In this case, the variable is empty after execution of EXECUTE-CMD if WRITE-MODE=*REPLACE was specified in the DECLARE-VARIABLE command (because of an implicit FREE-VARIABLE prior to command execution).

   The MSG-STRUCTURE-OUTPUT can be used to specify that guaranteed messages for the specified commands are to be output to an S variable.

   Since the EXECUTE-CMD command temporarily overwrites the assignments for SYSINF and SYSMSG made in the ASSIGN-STREAM command, there is no parallel output to the server specified in ASSIGN-STREAM.

ASSIGN-STREAM command

ASSIGN-STREAM serves to assign the S variable streams SYSINF, SYSMSG or SYSVAR to the declared variable. Structured output from SHOW commands is directed to SYSINF, guaranteed messages are directed to SYSMSG. While the assignment is in effect, the S variable is extended dynamically for each command that is issued and supports structured output. The assignment remains in effect until it is explicitly cancelled or until the procedure terminates.

3. **Output contents of the S variable**

```
/SHOW-VARIABLE  USERVAR
```

```
/show-var uservar,list-index-number=*yes
USERVAR#1.F-NAME = :1OSH:$TSOS.SYSSDF.JV.140
USERVAR#1.TYPE = *SYS
USERVAR#1.SW-UNIT#1.NAME = JV
USERVAR#1.SW-UNIT#1.VERSION = 14.0C100                          (1)
USERVAR#1.SW-UNIT#1.COMPONENT#1.NAME = JVS
USERVAR#1.SW-UNIT#1.COMPONENT#1.VERSION = 42C   (4)   (3)
USERVAR#1.SW-UNIT#1.COMPONENT#2.NAME = CJC
USERVAR#1.SW-UNIT#1.COMPONENT#2.VERSION = 20C   (5)
USERVAR#2.F-NAME = :1OSH:$TSOS.SYSSDF.LMS.033
USERVAR#2.TYPE = *SYS
USERVAR#2.SW-UNIT#1.NAME = LMS
USERVAR#2.SW-UNIT#1.VERSION = 03.3B300                          (2)
USERVAR#2.SW-UNIT#1.COMPONENT#1.NAME = LMS
USERVAR#2.SW-UNIT#1.COMPONENT#1.VERSION = 033
/
```

Explanation of the output:

The user-defined S variable (`USERVAR`) contains the entire output. The string “#i” shows the number of the corresponding list element (LIST-INDEX-NUMBER=*YES was specified in the SHOW-VARIABLE command). The S variable `USERVAR` contains 2 elements:
One is the structure for information concerning the software unit JV (marked with (1) in the output), the other is the structure for information concerning the software unit LMS (marked with (2) in the output).
The information concerning a software unit consists of the elements F-NAME, TYPE and SW-UNIT, where SW-UNIT can again contain a list. In the case of software unit JV, SW-UNIT contains a list element, namely the structure built by the elements NAME, VERSION and COMPONENT (marked with (3) in the output).

COMPONENT can again be a list: since software unit JV consists of 2 components, COMPONENT contains 2 list elements. Each list element is a structure with the elements NAME and VERSION. The components CJC and JVS are marked with (4) and (5) in the output.

4. **Access specific information**

   Specific information can be accessed via the names of the S variables. The names must be specified in the following format:

uservar#i.element where:

uservar	Name of the structure declared by the user
#i	i-th element in the list For i=1, “i” can be omitted, i.e. only “#” is specified.
period	Delimiter in names of complex S variables.
element	Predefined name of the structure element. element can again be a complex variable: e.g. uservar#.SW-UNIT#.NAME

The contents can be displayed, e.g. with “SHOW-VARIABLE uservar#.element”, or used for further processing via variable replacement:

```
/show-var (uservar#.sw-unit#.component#1.name,
            uservar#.sw-unit#.component#2.name),list-index-number=*yes
USERVAR#1.SW-UNIT#1.COMPONENT#1.NAME = JVS
USERVAR#1.SW-UNIT#1.COMPONENT#2.NAME = CJC

/write-text '*** SW-UNIT &(uservar#2.sw-unit#.name) -
/mit der Version &(uservar#2.sw-unit#.version) ***'
*** SW-UNIT LMS mit der Version 03.3B300 ***

/show-var uservar#2,list-index-number=*yes
USERVAR#2.F-NAME = :1OSH:$TSOS.SYSSDF.LMS.033
USERVAR#2.TYPE = *SYS
USERVAR#2.SW-UNIT#1.NAME = LMS
USERVAR#2.SW-UNIT#1.VERSION = 03.3B300
USERVAR#2.SW-UNIT#1.COMPONENT#1.NAME = LMS
USERVAR#2.SW-UNIT#1.COMPONENT#1.VERSION = 033
```

7.2.2 Example

```
/declare-var var-name=out-1(type=*structure),multiple-elements=*list
/execute-cmd cmd=(show-file-attr file-name=job*,inf=*par(alloc=*yes)),-
/                 text-output=*none,structure-output=out-1
```

```
/show-var out-1,inf=*par(value=*c-lit,list-index-number=*yes)
OUT-1#1.F-NAME = ':2OSG:$USER1.JOBA'
OUT-1#1.CAT-ID = '2OSG'
OUT-1#1.USER-ID = 'USER1'
OUT-1#1.SHORT-F-NAME = 'JOBA'
OUT-1#1.F-SIZE = 3
OUT-1#1.SUP = '*PUB'
OUT-1#1.HIGHEST-USED-PAGES = 1
OUT-1#1.SEC-ALLOC = 24
OUT-1#1.BLOCK-COUNT = 0
OUT-1#1.EXT#1.VOL = 'GVS2.2'
OUT-1#1.EXT#1.DEV = 'D3435'
OUT-1#1.EXT#1.NUM-OF-EXT = 1
*END-OF-VAR
OUT-1#1.NUM-OF-EXT = 1
*END-OF-VAR
OUT-1#2.F-NAME = ':2OSG:$USER1.JOBB'
OUT-1#2.CAT-ID = '2OSG'
OUT-1#2.USER-ID = 'USER1'
OUT-1#2.SHORT-F-NAME = 'JOBB'
OUT-1#2.F-SIZE = 3
OUT-1#2.SUP = '*PUB'
OUT-1#2.HIGHEST-USED-PAGES = 1
OUT-1#2.SEC-ALLOC = 24
OUT-1#2.BLOCK-COUNT = 0
OUT-1#2.EXT#1.VOL = 'GVS2.3'
OUT-1#2.EXT#1.DEV = 'D3435'
OUT-1#2.EXT#1.NUM-OF-EXT = 1
*END-OF-VAR
OUT-1#2.NUM-OF-EXT = 1
*END-OF-VAR
OUT-1#3.F-NAME = ':2OSG:$USER1.JOBC'
OUT-1#3.CAT-ID = '2OSG'
OUT-1#3.USER-ID = 'USER1'
OUT-1#3.SHORT-F-NAME = 'JOBC'
OUT-1#3.F-SIZE = 3
OUT-1#3.SUP = '*PUB'
OUT-1#3.HIGHEST-USED-PAGES = 1
OUT-1#3.SEC-ALLOC = 24
OUT-1#3.BLOCK-COUNT = 0
OUT-1#3.EXT#1.VOL = 'GVS2.0'
OUT-1#3.EXT#1.DEV = 'D3435'
```

```
OUT-1#3.EXT#1.NUM-OF-EXT = 1
*END-OF-VAR
OUT-1#3.NUM-OF-EXT = 1
*END-OF-VAR
```

Explanation:

The user-defined list variable VAR contains three elements.

The specification inf=*par(alloc=*yes) causes all file attributes of the selected files to be output that are relevant for memory allocation. The file attributes F-NAME, CAT-ID, USER-ID,....,EXT form the structure elements. The element EXT is again a list consisting of the elements VOL, DEV and NUM-OF-EXT.

If the information about an object can be organized in a hierarchical structure as for element EXT, a complex S variable is defined as an element of the superordinate structure. The subordinate S variable can be a simple S variable (as for VOL, DEV, NUM-OF-EXT), a structure of a list of simple S variables and/or structures.

The names of the list elements (e.g. F-NAME, F-SIZE) are command-specific and predefined for each SHOW command. They are appended to the names of the S variables as declared by the user. For each further hierarchical level of information, another name is appended, separated by a period.

7.3 FHS as output server

From FHS V8.1 on, FHS can act as output server for S variable streams. In addition, FHS applications can be used and controlled with the aid of S procedures. (For details see the “FHS” manual [19].)

Both these functions are described in the sections which follow.

7.3.1 Using FHS as the output server

In the context of SDF-P, FHS is available as the output server at both program and command level. This means:

- on the one hand, FHS-PRIV can be called by a TU program, using a TRANSVV SVC
- on the other hand, FHS-PRIV can be called at command level by a TRANSMIT-BY-STREAM command.

Before either of these options can be used, the variable stream used by the program must be assigned to FHS, using the SERVER operand of the ASSIGN-STREAM command.

When FHS receives data as a result of a TRANSMIT-BY-STREAM command, it provides display services similar to those it offers in TU mode through the interfaces DISPLAY, ADDPOP and REMPOP (for further details see the “FHS” manual [19]).

Structure layout and initialization

If FHS is used as an output server, the structure layout for FHS must be defined and initialized with particular values. To permit this, an S procedure is supplied as element SYSFHS-CONTROL of the library $TSOS.SYSPRC.FHS.<version> (included in the delivery). The S procedure is as follows for FHS V8.3:

```
/set-procedure-options caller=include
/begin-parameter-declaration
/   declare-parameter -
/   "------------ std param --------------------------------*"-
/               (PREFIX       (type=string,init='SYSFHS-') -
/               ,INCLUDE-FORM (type=string,init='LAYOUT') "/initialize" -
/               ,VARIABLE-NAME(type=string,init='') -
/   "------------ include specific param -------------------*"-
/  " action variables " -
/               ,SERVICE      (type=string,init='*DISPLAY') -
/               ,DIAGINFO     (type=string,init='*NO') -
/               ,POP-LOCATION (type=string,init='*NONE') -
/               ,POP-LOC-IND  (type=integer,init=0) -
/               ,ROW          (type=integer,init=2) -
/               ,COLUMN       (type=integer,init=2) -
/  " resource variables " -
/                ,RESOURCE      (type=string,init='*SAME') -
/                ,MESSAGE-ID    (type=string,init='*NONE') -
/                ,MESSAGE-FIELD (type=string,init='*NONE') -
/                ,MSG-FIELD-IND (type=integer,init=0) -
/  " panel variables " -
/                ,CURSOR-OUTPUT-INDEX (type=integer,init=0) -
/                ,CURSOR-OUTPUT       (type=string,init='*NONE')  -
/                ,CURSOR-OUTPUT-POS   (type=integer,init=0) -
/                ,LOCK                (type=string,init='*NO') -
/                ,ALARM               (type=string,init='*NO') -
/                ,HARDCOPY            (type=string,init='*NO') -
/                ,AUTOTAB             (type=string,init='*YES')  -
/                ,MANDATORY           (type=string,init='*NO')  -
/                ,REFRESH             (type=string,init='*NO')  -
/                ,ACK                 (type=string,init='*NO')  -
/                ,KEYLOCK             (type=string,init='*NONE')  -
/  " field attributes " -
/                ,ATTR-LIST    (type=integer,init=0) -
/                 " number of list elements to reset       "-
/                ,FIELD     (type=string,init='*CURSOR') -
/                ,FIELD-IND (type=string,init='0') -
/                ,TYPE      (type=string,init='*UNCHANGED') -
/                ,HILITE    (type=string,init='*UNCHANGED') -
/                ,INTENSITY (type=string,init='*UNCHANGED') -
/                ,COLOR     (type=string,init='*UNCHANGED') -
/                ,OUTPUT    (type=string,init='*UNCHANGED') -
/   " input information " -
/                ,COMMAND               (type=string,init='') -
/                ,FHS-VERSION           (type=string,init='') -
/                ,CURSOR-INPUT          (type=string,init='') -
/                ,CURSOR-INPUT-INDEX    (type=integer,init=0) -
/                ,CURSOR-INPUT-POS      (type=integer,init=0) -
/                )
/end-parameter-declaration
/
/if (upper-case(INCLUDE-FORM) == 'LAYOUT')
/
/begin-structure ATTR,scope=proc
/   declare-element -
/                    (FIELD     (type=string) -
/                    ,FIELD-IND (type=string) -
/                    ,TYPE      (type=string) -
/                    ,HILITE    (type=string) -
/                    ,INTENSITY (type=string) -
/                    ,COLOR     (type=string) -
/                    ,OUTPUT    (type=string) -
/                    )
```

```
/end-structure
/
/begin-structure name=&PREFIX.LAYOUT,scope=proc
/  declare-element STD-HEADER(type=structure(&PREFIX.FHDR))
/  declare-element -
/  " action variables " -
/                  (SERVICE      (type=string) -
/                  ,DIAGINFO     (type=string) -
/                  ,POP-LOCATION (type=string) -
/                  ,POP-LOC-IND  (type=integer) -
/                  ,ROW          (type=integer) -
/                  ,COLUMN       (type=integer) -
/  " resource variables " -
/                  ,RESOURCE         (type=string) -
/                  ,MESSAGE-ID       (type=string) -
/                  ,MESSAGE-FIELD    (type=string) -
/                  ,MSG-FIELD-IND    (type=integer) -
/  " panel variables " -
/                  ,CURSOR-OUTPUT-INDEX (type=integer) -
/                  ,CURSOR-OUTPUT       (type=string)  -
/                  ,CURSOR-OUTPUT-POS   (type=integer) -
/                  ,LOCK                (type=string)  -
/                  ,ALARM               (type=string)  -
/                  ,HARDCOPY            (type=string)  -
/                  ,AUTOTAB             (type=string)  -
/                  ,MANDATORY           (type=string)  -
/                  ,REFRESH             (type=string)  -
/                  ,ACK                 (type=string)  -
/                  ,KEYLOCK             (type=string)  -
/                  )
/  " field attributes "
/  declare-element ATTR            (type=struc(attr))-
/                 ,mult-elem=list
/
/   " input information " -
/   declare-element -
/                  (COMMAND               (type=string) -
/                  ,FHS-VERSION           (type=string) -
/                  ,CURSOR-INPUT          (type=string) -
/                  ,CURSOR-INPUT-INDEX    (type=integer) -
/                  ,CURSOR-INPUT-POS      (type=integer) -
/                  )
/end-structure
/
/else-if (upper-case(INCLUDE-FORM) == 'INITIALIZE')
/
/   if (VARIABLE-NAME == '')
/      write-text '%  mandatory parameter variable-name missing.'
/      raise-error
/   end-if
/  declare-variable PARAM(type=string)
/  SYSPRC-NAME = '$.SYSPRC.FHS.083'
/  IF ( SDF-P-VERSION >= 'V02.0A00' )
/     SYSPRC-NAME  = INSTALLATION-PATH -
/                    ( LOGICAL-ID        = 'SYSPRC' -
/                     ,INSTALLATION-UNIT = 'FHS' -
/                     ,VERSION           = 'V08.3' -
/                     ,DEFAULT-PATH-NAME = '&SYSPRC-NAME')
/  END-IF
/  include-procedure *lib-elem(lib=&SYSPRC-NAME,el=FHDR) -
/   "------------ std param -------------------------------*"-
/                ,(INCLUDE-FORM='INITIALIZE' -
/                 ,VARIABLE-NAME='&VARIABLE-NAME..STD-HEADER' -
/   "------------ include specific param ------------------*"-
/                 ,UNIT ='FHS',  "fhs unit name" -
/                 ,FUNCTION ='DISPLAY', "fhs fc for display?" -
/                 ,VERSION = 2 "control variable layout version"-
/                 )
/
/   for PARAM = -
/                   ('SERVICE'              -
/                   ,'DIAGINFO'             -
/                   ,'POP-LOCATION'         -
/                   ,'POP-LOC-IND'          -
/                   ,'ROW'                  -
/                   ,'COLUMN'               -
/                   ,'RESOURCE'             -
/                   ,'MESSAGE-ID'           -
/                   ,'MESSAGE-FIELD'        -
/                   ,'MSG-FIELD-IND'        -
/                   ,'CURSOR-OUTPUT-INDEX'  -
/                   ,'CURSOR-OUTPUT'        -
/                   ,'CURSOR-OUTPUT-POS'    -
/                   ,'LOCK'                 -
/                   ,'ALARM'                -
/                   ,'HARDCOPY'             -
/                   ,'AUTOTAB'              -
/                   ,'MANDATORY'            -
/                   ,'REFRESH'              -
/                   ,'ACK'                  -
/                   ,'KEYLOCK'              -
/                   ,'COMMAND'              -
/                   ,'FHS-VERSION'          -
/                   ,'CURSOR-INPUT'         -
/                   ,'CURSOR-INPUT-INDEX'   -
/                   ,'CURSOR-INPUT-POS'     -
/                   )
/         &VARIABLE-NAME..&PARAM = &PARAM
/    end-for
/
/    if (ATTR-LIST > 0)
/       I = 1
/       while ( I <= ATTR-LIST )
/          for PARAM = -
/                     ('FIELD'     -
/                     ,'FIELD-IND' -
/                     ,'TYPE'      -
/                     ,'HILITE'    -
/                     ,'INTENSITY' -
/                     ,'COLOR'     -
/                     ,'OUTPUT'    -
/                     )
/             &VARIABLE-NAME..ATTR#I.&PARAM = &PARAM
/          end-for
/          I=I+1
/       end-while
/    end-if
/
/else
/   write-text '%  form=&INCLUDE-FORM not supported; include aborts'
/   raise-error
/end-if
/EXIT-PROCEDURE
```

Note

This procedure contains an Include which links in the standard header. This is an S procedure, supplied with SDF-P, which is responsible for function identification and return code data (for further details see under TRANSMIT-BY-STREAM on page 541).

Example

The following sample procedure illustrates how the variable MYVAR is declared and initialized with the help of SYSFHS-CONTROL and the standard header:

```
/INCLUDE-PROC *LIB-ELEM(LIB=$TSOS.SYSPRC.SDF-P-BASYS.024,EL=FHDR),-
/(PREFIX='SYSFHS-')
/INCLUDE-PROC *LIB-ELEM(LIB=$TSOS.SYSPRC.FHS.083,EL=SYSFHS-CONTROL)
/DECLARE-VAR MYVAR(TYPE=*STRUCT(SYSFHS-LAYOUT))
/INCLUDE-PROC *LIB-ELEM(LIB=$TSOS.SYSPRC.FHS.083,EL=SYSFHS-CONTROL),-
/(INCLUDE-FORM=INITIALIZE,VARIABLE-NAME='MYVAR', ATTR-LIST = 2)
```

```
/SHOW-VARIABLE MYVAR
```

The variable is then generated and initialized in the following form:

```
MYVAR.STD-HEADER.INTERFACE-ID.UNIT = FHS
MYVAR.STD-HEADER.INTERFACE-ID.FUNCTION = DISPLAY
MYVAR.STD-HEADER.INTERFACE-ID.VERSION = 2
MYVAR.STD-HEADER.RETURNCODE.SUBCODE2 = 0
MYVAR.STD-HEADER.RETURNCODE.SUBCODE1 = 0
MYVAR.STD-HEADER.RETURNCODE.MAINCODE = CMD0001
MYVAR.SERVICE = *DISPLAY
MYVAR.DIAGINFO = *NO
MYVAR.POP-LOCATION = *NONE
MYVAR.POP-LOC-IND = 0
MYVAR.ROW = 2
MYVAR.COLUMN = 2
MYVAR.RESOURCE = *SAME
MYVAR.MESSAGE-ID = *NONE
MYVAR.MESSAGE-FIELD = *NONE
MYVAR.MSG-FIELD-IND = 0
MYVAR.CURSOR-OUTPUT-INDEX = 0
MYVAR.CURSOR-OUTPUT = *NONE
MYVAR.CURSOR-OUTPUT-POS = 0
MYVAR.LOCK = *NO
MYVAR.ALARM = *NO
MYVAR.HARDCOPY = *NO
MYVAR.AUTOTAB = *YES
MYVAR.MANDATORY = *NO
MYVAR.REFRESH = *NO
MYVAR.ACK = *NO
MYVAR.KEYLOCK = *NONE
MYVAR.ATTR(*LIST).FIELD = *CURSOR
MYVAR.ATTR(*LIST).FIELD-IND = 0
MYVAR.ATTR(*LIST).TYPE = *UNCHANGED
MYVAR.ATTR(*LIST).HILITE = *UNCHANGED
MYVAR.ATTR(*LIST).INTENSITY = *UNCHANGED
MYVAR.ATTR(*LIST).COLOR = *UNCHANGED
MYVAR.ATTR(*LIST).OUTPUT = *UNCHANGED
MYVAR.ATTR(*LIST).FIELD = *CURSOR
MYVAR.ATTR(*LIST).FIELD-IND = 0
MYVAR.ATTR(*LIST).TYPE = *UNCHANGED
MYVAR.ATTR(*LIST).HILITE = *UNCHANGED
MYVAR.ATTR(*LIST).INTENSITY = *UNCHANGED
MYVAR.ATTR(*LIST).COLOR = *UNCHANGED
MYVAR.ATTR(*LIST).OUTPUT = *UNCHANGED
MYVAR.COMMAND =
MYVAR.FHS-VERSION =
MYVAR.CURSOR-INPUT =
MYVAR.CURSOR-INPUT-INDEX = 0
MYVAR.CURSOR-INPUT-POS = 0
```

FHS return codes

SDF-P supplies the values of the TRANSMIT-BY-STREAM command return codes. FHS supplies the values of the return code variable for the structure which is passed back.

Name	Data type	Return values
SUBCODE2	integer	<integer>
SUBCODE1	integer	<integer>
MAINCODE	string	<name 1..7>

7.3.2 Use and control of FHS applications by S procedures

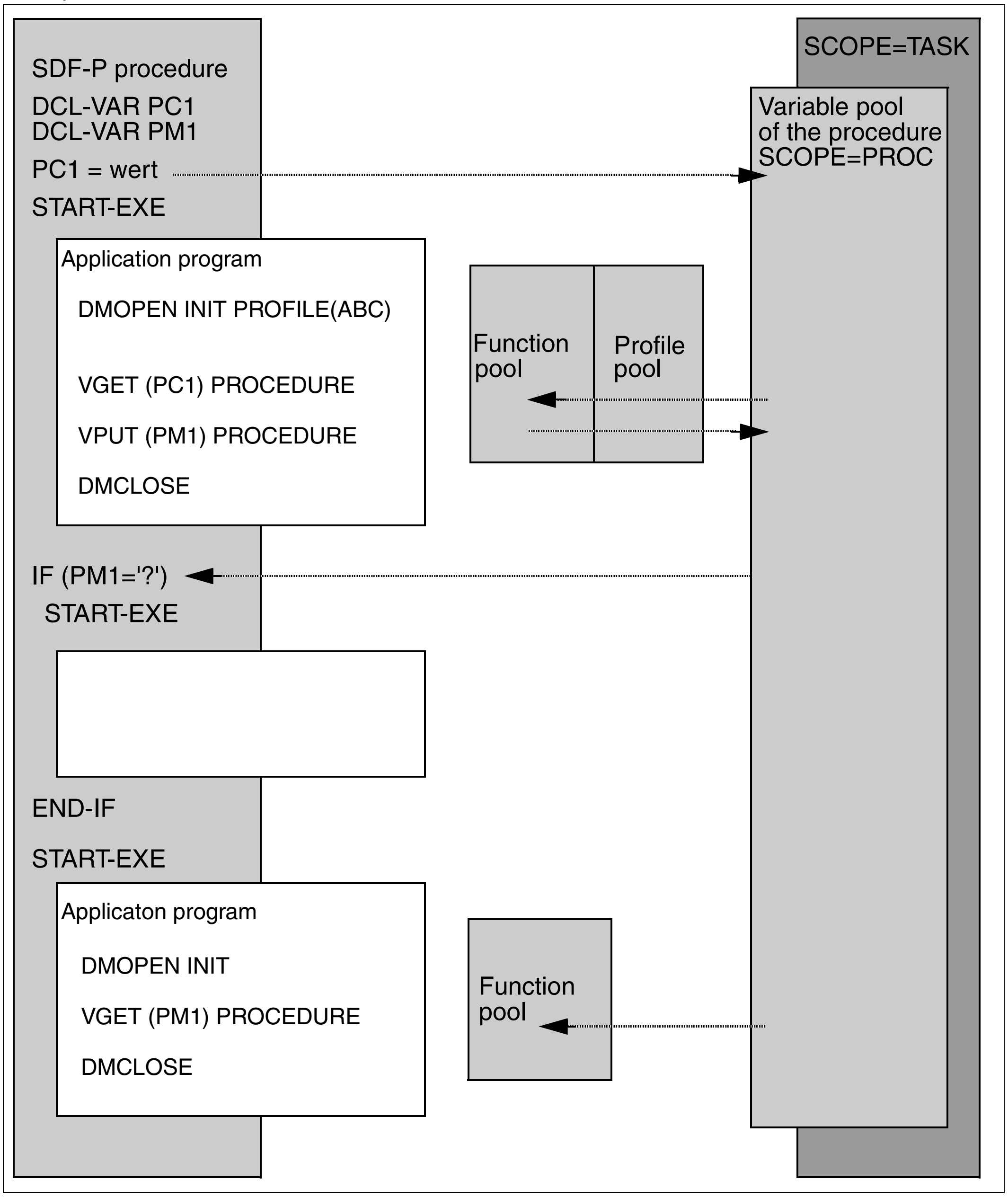
7.3.2.1 Using FHS to output S variables

S procedures can make use of the ASSIGN-STREAM and TRANSMIT-BY-STREAM commands to initiate a dialog with the end user, using interactive FHS control panels into which FHS dialog variables are inserted.

7.3.2.2 Outputting and creating S variables in FHS-TIAM programs

The FHS macros VGET and VPUT enable an application program to read and write simple S variables. This permits an S procedure and an application program called by this procedure to communicate.

The diagram below shows an exchange of variables between an application program and an S procedure.



7.3.2.3 Controlling FHS applications from nested S procedures

The assignments of S variable streams are batched in exactly the same way in nested S procedures as for system files (SYSDTA, SYSOUT,..). The operand value SYSTEM-FILE-CONTEXT=*STD or *OWN should therefore be set in the SET-PROCEDURE-OPTIONS command if the user wishes the stream assignments also to be processed in batches.

In TPR mode, FHS saves its display environment to agree with the S variable assignments in the TRANSMIT-BY-STREAM command. At the time the assignment is made, FHS initializes a context specific to the variable stream, which is automatically used for every FHS operation on that variable stream. This continues until the assignment ends, e.g. when *DUMMY or another server is assigned to that variable stream - either explicitly or implicitly when the procedure ends (in accordance with the SYSTEM-FILE-CONTEXT assignment).

This mechanism includes the possibility of nested procedures being coded independently, with no overlaying of display commands in different FHS contexts.

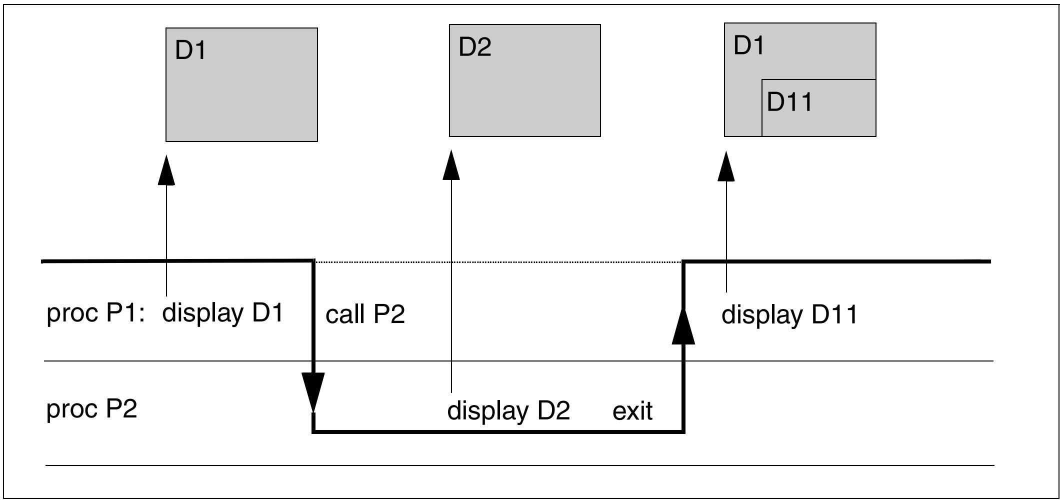
In order to ensure independence from the calling procedure, the variable SYSFHS-CONTROL.REFRESH must be assigned the value *YES, so that an operating panel is output when the procedure has been called.

Example

In the procedure P1, the variable stream S1 is assigned to FHS; the operating panel D1 is displayed in procedure P1. P1 calls procedure P2; the variable stream S2 in P2 is also assigned to FHS; P2 displays operating panel D2. D2 completely overwrites D1.

```
Proc P1 :  assign   S1 -----> FHS
           displays D1
           calls    P2
           pop-up   D11

Proc P2 :  assign   S2 -----> FHS
           displays D2
           exit
```

After P2 has ended, FHS returns to the display environment for S1. To effect this, the “pop-up menu” on the current screen (with or without branching, e.g. operating panel D11) is then implicitly reconstructed on the current operating panel for P1 (D1).



Operating panel D2 has been deleted and, the next time that P1 uses a display, operating panel D1 will be reconstructed.

Note

Variables which are to be displayed in an operating panel must be visible in the current S procedure (see section “Scope of variables” on page 157).

7.3.2.4 Application example

The example of a possible application which follows in full shows how the interaction of S procedures, S variable streams and FHS can be used to create a graphical library manager.

The individual S procedures which carry out this task are stored as J-type elements in the (user) library LIBRARY-MANAGER.PL, under the names RUN, SCREEN01 and SCREEN02. RUN is the controlling S procedure, which is called by a CALL-PROCEDURE. On the other hand, SCREEN01 and SCREEN02 are dependent on RUN and are responsible for the two possible standard screen displays.

A listing of these three S procedures appears below. Following this, a few applications are indicated to demonstrate how this FHS-supported library manager can be used.

Procedure: RUN

```
/SET-PROCEDURE-OPTIONS CALLER=CALL
/
/"-------------------------------------------------------------"
/"First get library name from which this procedure is called"
/"-------------------------------------------------------------"
/
/DECLARE-VARIABLE SYSOUT(TYPE=*STRUCTURE),MULTIPLE-ELEMENTS=*LIST
/EXECUTE-CMD (SHOW-SYSTEM-FILE-ASSIGNMENT SYSTEM-FILE=*SYSCMD),-
             /STRUCTURE-OUTPUT=SYSOUT,TEXT-OUTPUT=*NONE
/LIBRARY-NAME = SYSOUT#1.SYSCMD.LIB
/
/"-------------------------------------------------------------"
/"Get FHS library names via IMON-GPN                          "
/"-------------------------------------------------------------"
/
/FHSLNK = '$TSOS.SYSFHS.FHS.082.FHS-DM.D'
/FHSLNK = INSTALLATION-PATH(LOGICAL-ID        = 'SYSFHS.FHS-DM.D' , -
                           /INSTALLATION-UNIT = 'FHS' , -
                           /VERSION           = *STD , -
                           /DEFAULT-PATH-NAME = FHSLNK)
/
/FHSPRC = '$TSOS.SYSPRC.FHS.082'
/FHSPRC = INSTALLATION-PATH(LOGICAL-ID        = 'SYSPRC' , -
                           /INSTALLATION-UNIT = 'FHS' , -
                           /VERSION           = *STD , -
                           /DEFAULT-PATH-NAME = FHSPRC)
/
/"-------------------------------------------------------------"
/"Get SDF-P-BASYS library name via IMON-GPN                   "
/"-------------------------------------------------------------"
/
```

```
/SDPPRC = '$TSOS.SYSPRC.SDF-P-BASYS.022'
/SDPPRC = INSTALLATION-PATH(LOGICAL-ID        = 'SYSPRC' , -
                           /INSTALLATION-UNIT = 'SDF-P-BASYS' , -
                           /VERSION           = *STD , -
                           /DEFAULT-PATH-NAME = SDPPRC)
/
/"---------------------------------------------------------------"
/"Initialize FHS control variables                                "
/"---------------------------------------------------------------"
/WRITE-TEXT 'LIBRARY MANAGER V1.0 - LOADING'
/SHOW-VAR *ALL
/INCLUDE-PROCEDURE *LIBRARY-ELEMENT(LIBRARY = &(SDPPRC) -
/                                  ,ELEMENT = FHDR) -
/                  ,PROCEDURE-PARAMETERS = (PREFIX = 'SYSFHS-')
/SHOW-VAR *ALL
/INCLUDE-PROCEDURE *LIBRARY-ELEMENT(LIBRARY = &(FHSPRC) -
/                                  ,ELEMENT = SYSFHS-CONTROL)
/DECLARE-VARIABLE  SYSPINFO (TYPE = *STRUCTURE(SYSFHS-LAYOUT))
/DECLARE-VARIABLE  SYSPINFO-SAVE (TYPE = *STRUCTURE(SYSFHS-LAYOUT))
/INCLUDE-PROCEDURE *LIBRARY-ELEMENT(LIBRARY = &(FHSPRC) -
/                                  ,ELEMENT = SYSFHS-CONTROL) -
/                  ,PROCEDURE-PARAMETERS = (INCLUDE-FORM='INITIALIZE' -
/                                          ,VARIABLE-NAME='SYSPINFO')
/
/"---------------------------------------------------------------"
/"Assign the stream to FHS                                        "
/"---------------------------------------------------------------"
/ADD-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME=&(FHSLNK)
/ASSIGN-STREAM STREAM-NAME = PRESENTATION -
/              ,TO          = *SERVER(FHS -
/              ,SERVER-INFO = 'FHS-LIB = &(LIBRARY-NAME)')
/
/"---------------------------------------------------------------"
/"Start LMS                                                       "
/"---------------------------------------------------------------"
/ASSIGN-SYSOUT TO=*DUMMY
/START-LMS
/HOLD-PROGRAM
/ASSIGN-SYSOUT TO=*PRIMARY
/
/"------------------------------------------------------------------"
/"Set timeout to 0 when switching from line mode to full screen    "
/"------------------------------------------------------------------"
/MODIFY-TERMINAL-OPTIONS OVERFLOW-CONTROL = *TIME(TIMEOUT = 0)
/
/"---------------------------------------------------------------"
/"Call main procedure (screen01)                                  "
/"---------------------------------------------------------------"
```

```
/INCLUDE-PROCEDURE *LIBRARY-ELEMENT(LIBRARY = &(LIBRARY-NAME) -
/                                  ,ELEMENT = SCREEN01)
/
/"-----------------------------------------------------------"
/"Stop  LMS                                                  "
/"-----------------------------------------------------------"
/RESUME-PROGRAM
//END
```

Prozedur: SCREEN01

```
/declare-variable screen01(type=*structure(*by-syscmd))
/begin-structure
/  declare-element name = filelist (type = *structure(*by-syscmd)) -
/                 ,multiple-elements = *list
/  begin-structure
/    declare-element choice
/    declare-element f-size
/    declare-element cat-id
/    declare-element user-id
/    declare-element short-f-name
/  end-structure
/  declare-element name = sdfplist-modindex (type = integer) -
/                 ,multiple-elements = *list
/  declare-element sdfplist-topindex(initial-value = 1)
/  declare-element sdfplist-botindex
/  declare-element sdfplist-numrow
/  declare-element file-menu
/  declare-element file-choice
/end-structure
/
/declare-variable i(type = *integer)
/
/while (true)
/
/     "initialize modindex list for 50 elements "
/     " (fhs requirement)                        "
/     for i = *counter(1,50)
/        screen01.sdfplist-modindex#&(i) = 0
/     end-for
/
/     "get library names"
/     exec-cmd cmd=(show-file-attributes -
/                     select=*by-attributes(type-of-files = *plam-library) -
/                    ,info=*name-and-space -
/                  ) -
/             ,structure-output=screen01.filelist -
/              ,text-output=*none -
/              ,returncode=*variable(subcode2=sub2 -
/                                   ,subcode1=sub1 -
/                                   ,maincode=main)
/
/      if (sub1 ne 0)
/         write-text 'Error &sub2 &sub1 &main returned by EXEC-CMD'
/         write-text 'LIBRARY MANAGER V1.0 abnormally terminated'
/         exit-procedure
/      end-if
/
/      syspinfo.resource = 'screen01'
/      syspinfo.service = '*display'
/      syspinfo.refresh = '*yes'
/      syspinfo.command = ''
/      screen01.sdfplist-numrow = size('screen01.filelist')
/      screen01.file-menu=0
/      screen01.file-choice=0
/      transmit-by-stream variable-name = screen01 -
/                         ,stream-name = presentation -
/                         ,control-var-name = syspinfo
/
/      if (   (syspinfo.std-header.returncode.maincode == 'IDH0004') -
/          or (syspinfo.std-header.returncode.maincode == 'IDH0008'))
/         write-text 'LIBRARY MANAGER V1.0 normally terminated'
/         exit-procedure
/      end-if
/
/      if (syspinfo.std-header.returncode.maincode ne 'IDH0000')
/         sub2 = syspinfo.std-header.returncode.subcode2
/         sub1 = syspinfo.std-header.returncode.subcode1
/         main = syspinfo.std-header.returncode.maincode
/         write-text 'Error &sub2 &sub1 &main returned by FHS server'
/         write-text 'LIBRARY MANAGER V1.0 abnormally terminated'
/         exit-procedure
/      end-if
/
/      if screen01.file-menu ne 0
/         if screen01.file-choice == 9
/            write-text 'LIBRARY MANAGER V1.0 normally terminated'
/            exit-procedure
/         else-if screen01.file-choice == 1
/          for i = *counter(1,size('screen01.sdfplist-modindex')), -
               /cond=(screen01.sdfplist-modindex#i ne 0)
/              screen01-curr-index = screen01.sdfplist-modindex#i
/              if screen01.filelist#screen01-curr-index.choice == '/'
/                 syspinfo-save = syspinfo
/                 include-procedure -
/                      name=*library-element(&library-name. -
/                                            ,screen02) -
/                     ,procedure-parameters=(&(screen01.filelist#screen01-curr-
index.short-f-name))
/                   if-cmd-error
/                      write-text 'LIBRARY MANAGER V1.0 abnormally terminated'
/                      exit-procedure
/                   else
/                      save-returncode
/                      if (maincode() = 'STOPOOK')
/                         write-text 'LIBRARY MANAGER V1.0 normally terminated'
/                         exit-procedure
/                      end-if
/                   end-if
/                   syspinfo = syspinfo-save
/                end-if
/             end-for
/          end-if
/       end-if
/
/       if (syspinfo.command ne '')
/          exec-cmd cmd=(&(syspinfo.command)) -
/                  ,text-output=*none -
/                  ,returncode=*variable(subcode2=sub2 -
/                                       ,subcode1=sub1 -
/                                       ,maincode=main)
/
/          if (sub1 ne 0)
/             write-text 'Error &sub2 &sub1 &main returned by command server'
/             write-text 'LIBRARY MANAGER V1.0 abnormally terminated'
/             exit-procedure
/          end-if
/       end-if
/end-while
```

Procedure: SCREEN02

```
/begin-parameter-declaration
/  declare-parameter library
/end-parameter-declaration
/
/declare-variable screen02(type=*structure(*by-syscmd))
/begin-structure
/  declare-element name = elemlist(type = *structure(*dynamic)) -
/                 ,multiple-element = *list
/  declare-element name = sdfplist-modindex(type = *integer) -
/                 ,multiple-element = *list
/  declare-element sdfplist-topindex(initial-value = 1)
/  declare-element sdfplist-botindex
/  declare-element sdfplist-numrow
/  declare-element file-menu
/  declare-element file-choice
/end-structure
/
/declare-variable sysout(type = *string), multiple-elements = *list
/declare-variable error-on-print( type = *boolean, initial-value = false )
/declare-variable i(type = *integer)
/
/resume-program
//open-library library = &library.,mode = *update
/hold-program
/
/while (true)
/
/   "initialize modindex list for 50 elements "
/   " (fhs requirement)                       "
/   for i = *counter(1,50)
/      screen02.sdfplist-modindex#&(i) = 0
/   end-for
/
/   assign-sysout to = *variable(sysout)
/   resume-program
//  show-element-attributes -
//       element = *library-element(library = *std -
//                                 ,element = *all ( version = *all ) -
//                                 ,type = *all) -
//      ,information = *maximum -
//      ,sort = *by-name -
//      ,structure-output = screen02.elemlist
/   hold-program
/   assign-sysout to = *primary
/
/   if ( stmt-spinoff() == 'YES' )
/      show-variable sysout, information = *parameters( name = *none )
/      maincode = 'LMS0ERR'
/      goto end
/   end-if
/
/   "Following loop is only necessary to rep a problem between"
/   "FHS and VAS. Correction in VAS V02.0A85, FHS V08.1A75"
/   for i = *counter(1,size('screen02.elemlist'))
/       screen02.elemlist#i.choice = ' '
/   end-for
/
/   syspinfo.resource = 'screen02'
/   syspinfo.service = '*display'
/   syspinfo.refresh = '*yes'
/   syspinfo.command = ''
/   screen02.sdfplist-numrow = size('screen02.elemlist')
/   screen02.file-menu=0
/   screen02.file-choice=0
/   transmit-by-stream variable-name = screen02 -
/                     ,stream-name = presentation -
/                     ,control-var-name = syspinfo
/
/   if (    (syspinfo.std-header.returncode.maincode == 'IDH0004') -
/       or (syspinfo.std-header.returncode.maincode == 'IDH0008'))
/      maincode = 'FHSEXIT'
/      goto end
/   end-if
/
/   if (syspinfo.std-header.returncode.maincode ne 'IDH0000')
/      sub2 = syspinfo.std-header.returncode.subcode2
/      sub1 = syspinfo.std-header.returncode.subcode1
/      main = syspinfo.std-header.returncode.maincode
/      write-text 'Error &sub2 &sub1 &main returned by FHS server'
/      maincode = 'FHS0ERR'
/      goto end
/   end-if
/
/   if screen02.file-menu ne 0
/      if screen02.file-choice == 9
/         maincode = 'FHS0RET'
/         goto end
/      else
/         for i = *counter(1,size('screen02.sdfplist-modindex')), -
            /cond=(screen02.sdfplist-modindex#i ne 0)
/            screen02-curr-index = screen02.sdfplist-modindex#i
/            if screen02.elemlist#screen02-curr-index.choice == '/'
/               element = screen02.elemlist#screen02-curr-index.elem
/               version = screen02.elemlist#screen02-curr-index.version
/               type    = screen02.elemlist#screen02-curr-index.type
/               if screen02.file-choice == 1 "delete element"
/                  assign-sysout to = *variable(sysout)
/                  resume-program
//                 delete-element element = *library-element -
//                                            ( library = *std -
//                                            , element = &element.-
//                                                ( version = &version. ) -
//                                            , type = &type. )
/                  hold-program
/                  assign-sysout *primary
/               else-if screen02.file-choice == 2 "edit element"
/                  assign-sysout to = *variable(sysout)
/                  resume-program
//                 edit-element element = *library-element -
//                                            ( library = *std -
//                                            , element = &element. -
//                                                ( version = &version. ) -
//                                            , type = &type. )
/                  hold-program
/                  assign-sysout *primary
/               else-if screen02.file-choice == 3 "copy element"
/                  write-text 'Function not implemented'
/               else-if screen02.file-choice == 4 "print element"
/                  assign-sysout to = *variable(sysout)
/                  print-file *library-element -
/                                ( library = &library. -
/                                , element = &element. -
/                                    ( version = &version. ) -
/                                , type = &type. )
/                  if-cmd-error
/                     error-on-print = true
/                  end-if
/                  assign-sysout *primary
/               else-if screen02.file-choice == 5 "select element"
/                  assign-sysout to = *variable(sysout)
/                  resume-program
//                 extract-element element = *library-element -
//                                            ( library = *std -
//                                            , element = &element. -
//                                                ( version = &version. ) -
//                                            , type = &type. ) -
//                              ,to-file = *std
/                  hold-program
/                  assign-sysout *primary
/               else-if screen02.file-choice == 6 "add element"
/                  write-text 'Function not implemented'
/               end-if
```

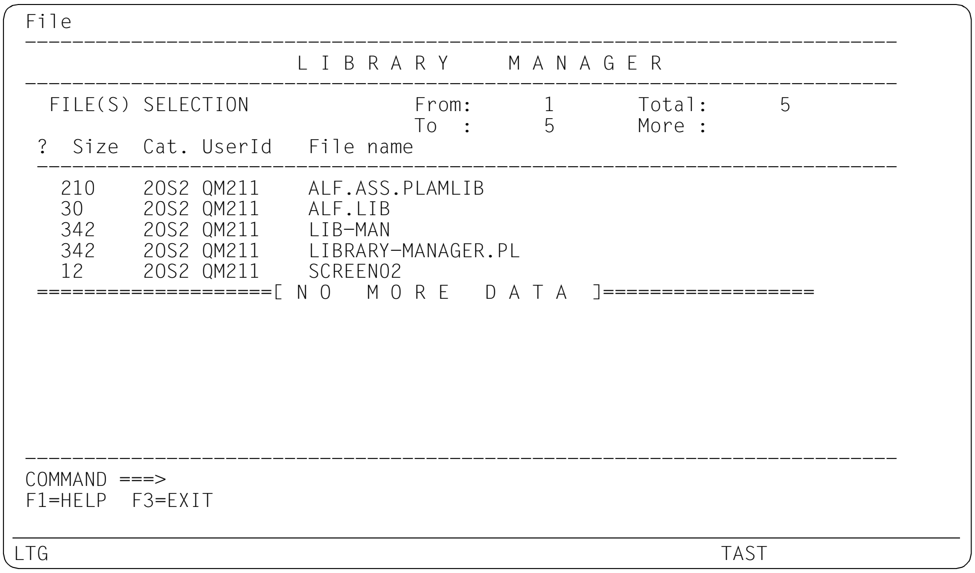
```
/
/               if ( error-on-print )
/                  show-variable sysout, information = *parameters( name =
*none )
/                  maincode = 'PRT0ERR'
/                  goto end
/               end-if
/
/               if ( stmt-spinoff() == 'YES' )
/                  show-variable sysout, information = *parameters( name =
*none )
/                  maincode = 'LMS0ERR'
/                  goto end
/               end-if
/            end-if
/         end-for
/      end-if
/   end-if
/
/   if (syspinfo.command ne '')
/      exec-cmd cmd=(&(syspinfo.command)) -
/              ,text-output=*none -
/              ,returncode=*variable(subcode2=sub2 -
/                                   ,subcode1=sub1 -
/                                   ,maincode=main)
/
/      if (sub1 ne 0)
/         write-text 'Error &sub2 &sub1 &main returned by command server'
/         maincode = 'CMD0ERR'
/         goto end
/      end-if
/   end-if
/end-while
/
/
/end:
/if (    (maincode = 'CMD0ERR') -
/     or (maincode = 'PRT0ERR') -
/     or (maincode = 'LMS0ERR') -
/     or (maincode = 'FHS0ERR') -
/   )
/   exit-procedure error = *yes(subcode2 = 0 -
/                              ,subcode1 = 64 -
/                              ,maincode = STOPERR)
/else-if (maincode = 'FHSEXIT')
/   exit-procedure error = *yes(subcode2 = 0 -
/                              ,subcode1 = 0  -
/                              ,maincode = STOP0OK)
```

```
/else-if (maincode = 'FHSORET')
/   exit-procedure
/else
/   write-text 'Error &(sc2()) &(sc1()) &(mc()) reported'
/   exit-procedure error = *yes(subcode2 = 0 -
/                              ,subcode1 = 64 -
/                              ,maincode = STOPERR)
/end-if
```

Assuming that FHS-PRIV is loaded, the library manager can be called as follows:

```
/CALL-PROCEDURE FROM-FILE=*LIBRARY-ELEMENT(LIBRARY-MANAGER.PL, RUN)
```

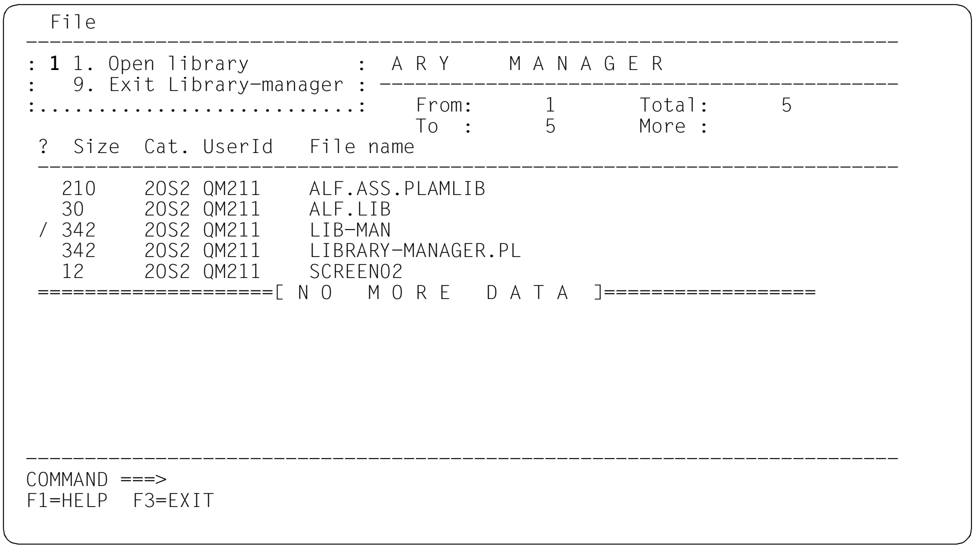
A screen display should then appear (making use of SCREEN01), listing the names of the libraries which are held under the ID, e.g.:



```
File
--------------------------------------------------------------------------
                      L I B R A R Y     M A N A G E R
--------------------------------------------------------------------------
  FILE(S) SELECTION             From:     1      Total:     5
                                To  :     5      More :
 ?  Size  Cat. UserId   File name
 -------------------------------------------------------------------------
   210    20S2 QM211    ALF.ASS.PLAMLIB
   30     20S2 QM211    ALF.LIB
   342    20S2 QM211    LIB-MAN
   342    20S2 QM211    LIBRARY-MANAGER.PL
   12     20S2 QM211    SCREEN02
 ====================[ N O   M O R E   D A T A ]==================




--------------------------------------------------------------------------
COMMAND ===>
F1=HELP  F3=EXIT

LTG                                                        TAST
```

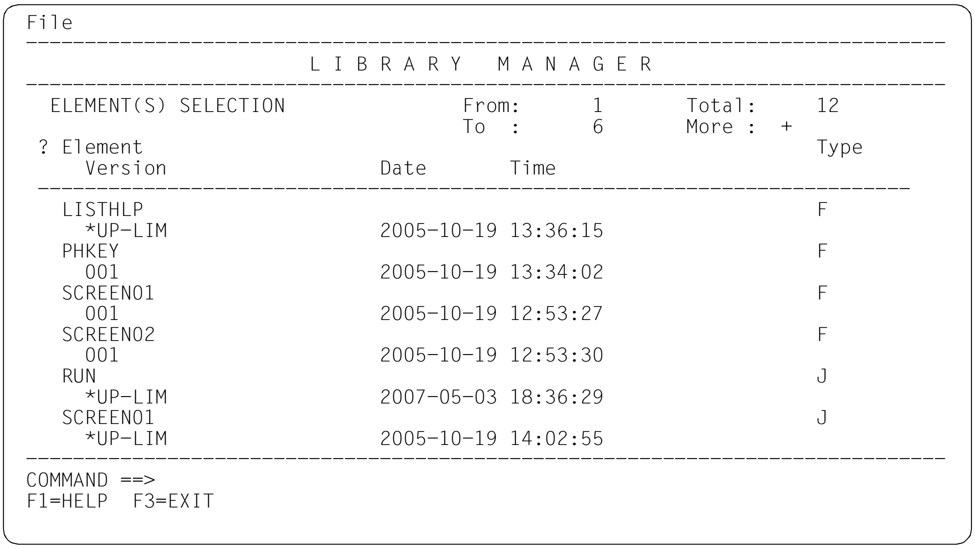
It is now possible to open the listed libraries, and request a display of the appropriate elements. To do so, identify the library to be opened by marking the start of the corresponding line with a “/”; press the tab key to move the cursor into the “FILE” field (top left); press the DUE key, and enter “1” into the pull-down menu which then appears.

The screen output below should clarify this:



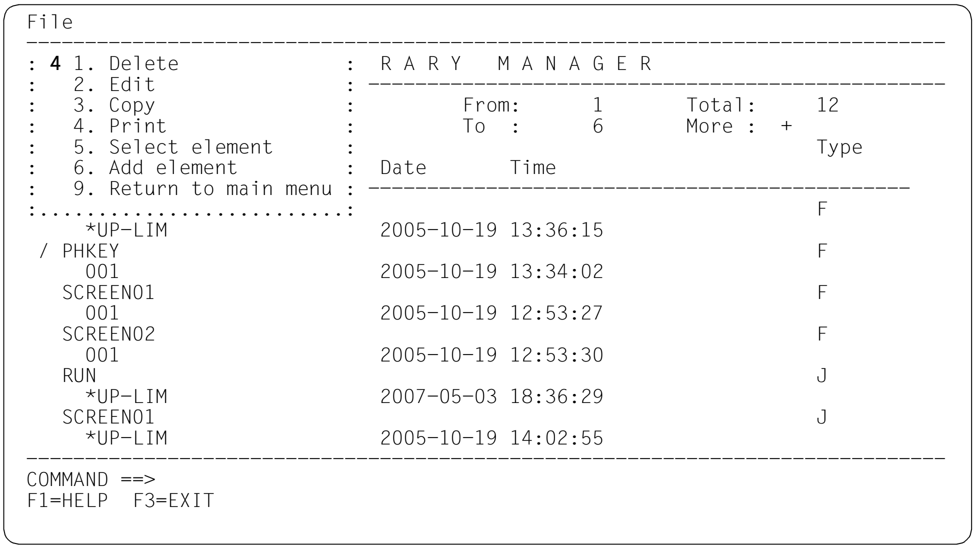
```
  File
---------------------------------------------------------------------------
: 1 1. Open library        :  A R Y     M A N A G E R
:   9. Exit Library-manager : ------------------------------------------
:.........................:    From:      1       Total:      5
                               To  :      5       More :
 ?  Size  Cat. UserId   File name
 --------------------------------------------------------------------------
   210    2OS2 QM211    ALF.ASS.PLAMLIB
   30     2OS2 QM211    ALF.LIB
 / 342    2OS2 QM211    LIB-MAN
   342    2OS2 QM211    LIBRARY-MANAGER.PL
   12     2OS2 QM211    SCREEN02
 ====================[ N O   M O R E   D A T A  ]==================







---------------------------------------------------------------------------
COMMAND ===>
F1=HELP  F3=EXIT
```

If the DUE key is now pressed again, then another screen appears (making use of SCREEN02) in which is output the appropriate names of the elements in this library (together with the dates and times of their creation, their types, etc.).



```
File
-------------------------------------------------------------------------------
                        L I B R A R Y   M A N A G E R
-------------------------------------------------------------------------------
  ELEMENT(S) SELECTION                 From:      1       Total:     12
                                       To  :      6       More :  +
 ? Element                                                          Type
     Version                     Date       Time
 ---------------------------------------------------------------------------
   LISTHLP                                                          F
     *UP-LIM                     2005-10-19 13:36:15
   PHKEY                                                            F
     001                         2005-10-19 13:34:02
   SCREEN01                                                         F
     001                         2005-10-19 12:53:27
   SCREEN02                                                         F
     001                         2005-10-19 12:53:30
   RUN                                                              J
     *UP-LIM                     2007-05-03 18:36:29
   SCREEN01                                                         J
     *UP-LIM                     2005-10-19 14:02:55
-------------------------------------------------------------------------------
COMMAND ==>
F1=HELP  F3=EXIT
```

A further option at this point is to process any of the listed elements. To do so, identify the element to be processed (e.g. PHKEY) by marking the start of the corresponding line with a “/”; press the tab key to move the cursor into the “FILE” field (top left); press the DUE key, and enter into the pull-down menu which now appears the number which corresponds to the required processing for the element (e.g. “4” to print it out).


```
File
------------------------------------------------------------------------------
: 4 1. Delete              :  R A R Y   M A N A G E R
:   2. Edit                : -------------------------------------------------
:   3. Copy                :         From:      1       Total:     12
:   4. Print               :         To  :      6       More :  +
:   5. Select element      :                                       Type
:   6. Add element         :  Date       Time
:   9. Return to main menu : ----------------------------------------------
:..........................:                                       F
     *UP-LIM                  2005-10-19 13:36:15
 / PHKEY                                                           F
     001                      2005-10-19 13:34:02
   SCREEN01                                                        F
     001                      2005-10-19 12:53:27
   SCREEN02                                                        F
     001                      2005-10-19 12:53:30
   RUN                                                             J
     *UP-LIM                  2007-05-03 18:36:29
   SCREEN01                                                        J
     *UP-LIM                  2005-10-19 14:02:55
------------------------------------------------------------------------------
COMMAND ==>
F1=HELP  F3=EXIT
```

If the DUE key is now pressed, the required action will be performed.

So much for the possible applications. Pressing the F3 key exits from the library manager or returns to the initial menu.

Incidentally, it is possible to scroll the display by entering “+” or “-” in the “COMMAND” line to the right of the arrow. There is a Help menu which can be called up by pressing the F1 key.

8 Functions

Functions in SDF-P have the following characteristics:

– Functions are called by means of a function name.
– The result of a function is obtained from the input parameters.
– Exactly one result value is returned.
– The result value is inserted at the position of the function call.

The SDF-P scope of supply includes “predefined functions” (also called “built-in” functions) with which the user can process variables and strings or request information on the current system environment. For this purpose, SDF-P also supports functions which the system administrator creates via a special Assembler interface.

This chapter describes, first, general aspects of the use of functions, i.e. how they are called, etc.; and secondly, the factors which must be taken into consideration when the system administrator writes his/her own functions.

8.1 Function call

Functions are called by means of their function name.

SDF-P recognizes a function by the parentheses that follow the function name, e.g. USER-IDENTIFICATION( ).

These parentheses can be omitted if the function call does not contain any input parameters. In this case, however, SDF-P first interprets the (function) name as a variable name. SDF-P interprets the name as a function name only if no variable with this name exists.

If a variable of the same name exists, SDF-P accesses the variable and inserts the variable value, if any.

Note

The parentheses should always be included in the function call in order to avoid any possibility of confusion with variables.

Functions are components of expressions. This means they can be called wherever expressions are allowed.

In assignments, all functions can be called that appear on the right-hand side of the assignment (i.e. to the right of the equals sign). In this way, for example, the result value of the function is assigned to a variable.

Example

```
/USER = USER-IDENTIFICATION( )
```

The user ID is assigned as a value to the USER variable.

If functions are called in expressions, when the expression is evaluated SDF-P inserts the result of the function at the location in the expression from which the function was called.

Example

```
/DECLARE-VARIABLE VAR-A
/NAME = 'ELEM1' // FIRST-VARIABLE-NAME('VAR-A')
/SHOW-VARIABLE NAME
NAME = ELEM1*END
```

The NAME variable is assigned a string made up of the string ELEM1 and the string returned as the result of the function (in this case *END).

In command calls, functions can be inserted in order to build up a command. For example, functions can be called when operand values are assigned. They can also be used like variables in & replacement.

Example

```
/DECLARE-VARIABLE ST(TYPE=*STRUCTURE(*DYNAMIC))
/SET-VARIABLE ST.A1 = 'ANNA'
/SHOW-VARIABLE ST.A1
ST.A1=ANNA
/SET-VARIABLE &(FIRST-VARIABLE-NAME('ST')) = 'MARIA'
/SHOW-VARIABLE ST.A1
ST.A1 = MARIA
```

The first element of structure ST, which can be determined by & replacement of the function FIRST-VARIABLE-NAME( ), is assigned the string ’MARIA’ as its contents (thus overwriting the old contents).

8.1.1 Input parameters in function calls

When functions are called, a distinction must be made between functions without input parameters and functions with input parameters.

Note

The syntax rules for function calls, including abbreviation options, are described in the sections below. However, the abbreviation options should be used only if evaluation remains straightforward. In accordance with the rules of structured programming and, above all, for the sake of ease of maintenance, function calls should be entered in as complete a form as possible.

The syntax of function calls without input parameters and the syntax of function calls with input parameters are dealt with separately in the next two sections in order to keep the syntax representations clear.

8.1.2 Functions without input parameters

Syntax

```
function[( )]
```

function
Name of the function.

( )
Identifies a function call.

The parentheses ( ) indicate that this is a function and not a variable. If unique function and variable names are assigned, the parentheses can be omitted.

If function and variable names are not unique (i.e. if there are functions and variables with the same name), parentheses must be used to identify this as a function. If the parentheses are omitted, SDF-P interprets the name as a variable and inserts the value of the variable in accordance with the rules of variable handling.

Examples

If there is no variable with the name USER-IDENTIFICATION, the entries USER-IDENTIFICATION and USER-IDENTIFICATION( ) are equivalent; they are both interpreted as function calls.

If a variable named USER-IDENTIFICATION is defined, only the entry USER-IDENTIFICATION( ) is interpreted as a function call; in this case, the entry USER-IDENTIFICATION is handled as a variable name.

8.1.3 Functions with input parameters

Syntax

```
function([[[parameter =] value], ...])
```

function
Name of the function.

parameter
Name of an input parameter; can be omitted.

value
Value of the input parameter.
Keyword or expression that corresponds to a data type that is valid in SDF-P (STRING, INTEGER, BOOLEAN; for information on expressions, see chapter “Expressions” on page 249).
A keyword must always begin with an asterisk (to distinguish keywords from variable names).

Examples

Keyword as parameter value:

```
DATE(FORMAT=*ISO)
```

Expression as parameter value:

```
/ADDRESS = 'hello'
/U = UPPER-CASE(STRING=ADDRESS)
```

or:

```
/A = 3
/INT = INTEGER(190+A)
```

Functions can have several input parameters that must all comply with this syntax. In accordance with the rules for structured programming and to facilitate program maintenance, parameter names should be specified wherever possible, especially in function calls with several input parameters.

Keyword/positional parameters

In the case of functions with several input parameters, a distinction must be made between keyword parameters and positional parameters.

When keyword parameters are used, the parameter name is identified in the assignment as a keyword. The PARAMETER = VALUE assignments can then be in any order.

When positional parameters are used, the parameter name and equals sign are omitted from the assignment; only the parameter value is specified. The assignment is then identifiable by its position within the assignment sequence only.

If you do not specify parameter names in the function call, the order of input parameters must comply with the order given in the functional description in chapter “Predefined functions” on page 347. If no parameter names are specified, SDF-P evaluates the parameter values in the function call in this exact order.

The rules for positional and keyword parameters in SDF-P functions correspond to the general rules for positional and keyword parameters in SDF commands. Important: positional parameters are only allowed before keyword parameters.

Example

In the syntax representation of the FILL function, the input parameters are listed in the following order: STRING, LENGTH, SIDE, FILL-BYTE. The value of the SIDE input parameter is specified by means of a keyword (*RIGHT or *LEFT); the values of the other input parameters are expressions that can be specified by means of variables or directly as a single character (for FILL-BYTE), a string (for STRING) or a number (for LENGTH). Consequently, the following function calls are equivalent:

```
/ADDRESS = 'ABCDE'
/B = FILL(STRING=ADDRESS, LENGTH=18, SIDE=*LEFT, FILL-BYTE=C' ')
/B = FILL('ABCDE', 18, *LEFT,C' ')
/B = FILL(LENGTH=18, FILL-BYTE=C' ',STRING=ADDRESS, SIDE=*LEFT)
```

Input parameters can also be specified more than once in a function call. The last entry is always valid.

Example

```
DATE(FORMAT=*ISO, FORMAT=*GERMAN)
```

The date is returned in the *GERMAN format.

8.1.4 Transferring default values

If a default value is defined for an input parameter, the parameter does not have to be specified in the function call if the default value is to be transferred (the value of default settings is always underlined in the syntax representations of functions).

For functions that have only one input parameter, such as DATE( ), this means that the parentheses identifying the function call are sufficient. They can even be omitted if no variables have been defined with the same name.

Example

The following function calls for the DATE( ) function are equivalent, provided that there is no variable with the name DATE:

```
/DATE(FORMAT=*ISO)
/DATE(*ISO)
/DATE( )
/DATE

/SHOW-VARIABLE D
D = 1996-05-20141
```

For functions with several input parameters, the following rules apply:

- If all input parameters are specified as keyword parameters, the input parameters for which the default value is to be transferred can be omitted without replacement.
- If all input parameters are specified as positional parameters, the order of the input parameters must be correct. Input parameters for which the default value is to be transferred must be separated by commas, unless they are located at the end of the parameter list.
- If positional and keyword parameters are mixed, the positional parameters must be specified first, followed by the keyword parameters. Input parameters for which the default value is to be transferred must be separated in the list of positional parameters by commas.

Examples

The example of the FILL( ) function illustrates the rules for transferring default values. The following syntax applies for the input parameters of FILL( ):

FILL
STRING = string_expression
,LENGTH = number
,SIDE = *RIGHT / *LEFT
,FILL-BYTE = C' ' / character

In this example, default settings are defined for the input parameters SIDE and FILL-BYTE:

– *RIGHT means that filling is to the right
– C’␣’ means that spaces are used as fill characters (not to be confused with a null string (C”)!)

If you now wish to transfer both default values and insert the contents of the ADDRESS variable for STRING and the number 18 for LENGTH, the following function calls are equivalent:

```
FILL(STRING=ADDRESS, LENGTH=18, SIDE=*RIGHT, FILL-BYTE=C' ')
FILL(STRING=ADDRESS, LENGTH=18,,)
FILL(STRING=ADDRESS, LENGTH=18)
FILL(ADDRESS,18)
```

If you specify the parameter names, you can also change the order of input parameters:

```
FILL(LENGTH=18, STRING=ADDRESS)
```

If only the default value for SIDE is to be transferred and different fill characters are to be defined (e.g. dots), the following function calls are equivalent:

```
FILL(STRING=ADDRESS, LENGTH=18, SIDE=*RIGHT, FILL-BYTE=C'.')
FILL(STRING=ADDRESS, LENGTH=18, FILL-BYTE=C'.')
FILL(ADDRESS, 18, SIDE=*RIGHT, FILL-BYTE=C'.')
FILL(ADDRESS, 18, ,C'.')
```

Whenever you use abbreviation options, be careful that you do not make familiarization with the procedures unnecessarily difficult for your “successors”. You should abbreviate function calls only to the extent that they remain straightforward and easily understood.

8.1.5 Guidelines on specifying values for input parameters

Depending on whether or not quotation marks and the escape character are specified for those predefined functions which accept file names or variables, different actions will be performed. In the normal situation, what happens will be the same as with the other predefined functions. However, it should be noted that the behavior is not comparable with BS2000 commands (e.g. DMS commands), and consequently the results may at first sight appear surprising.

Example of a function which accepts file names

```
/FILE = 'ABC'

/A=IS-CATALOGED-FILE(FILE)       "TESTS, WHETHER 'ABC' IS CATALOGED"
/A=IS-CATALOGED-FILE('FILE')     "TESTS, WHETHER 'FILE' IS CATALOGED"
/A=IS-CATALOGED-FILE('&FILE')    "TESTS, WHETHER 'ABC' IS CATALOGED"
```

Example of a function which accepts variable names

```
/A = 'B'
/B = 0

/CURRENT-TYPE('A')  "RETURNS *STRING"
/CURRENT-TYPE('B')  "RETURNS *INTEGER"
```

8.1.6 Abbreviations of names and keywords

Abbreviations of function names must follow certain rules. For some functions, a fixed abbreviation is defined as an alias; it is specified in the syntax representation as a shorter name.

Example

```
SUBSTRING( )
SUBSTR( )
```

The function can be called either with the name SUBSTRING or SUBSTR.

Parameter names can be abbreviated as long as they remain unique.

An abbreviation in the format “parameter=” is not possible. The entry is rejected.

Keywords can be abbreviated to the point of uniqueness in accordance with the SDF abbreviation rules. The following points must be noted:

– The abbreviation may no longer be unique in later SDF-P versions, if there are new keywords with the same abbreviation options. For this reason, you should try to abbreviate less than what is allowed.
– Procedures should be kept straightforward and easy to read, i.e. it is not always a good idea to abbreviate names to their shortest possible form.

8.2 Function result

The predefined functions included in the SDF-P scope of supply always return exactly one return value from the input parameters of the function call and the current environment data. This value is the function result.

Example

The function USER-IDENTIFICATION( ) is a function without input parameters; it returns the user ID of the current task as a return value.

Call:	Return value:
USER-IDENTIFICATION( )	'US123456'

Example

The function DATE( ) has exactly one input parameter; it returns the current date as its return value. The FORMAT input parameter determines the format in which the date is transferred, i.e. the way in which the day, month and year are arranged.

Call:	Return value:
DATE(FORMAT=*ISO)	'1996-06-24176'
DATE(FORMAT=*GERMAN)	'24.06.1996'
DATE(FORMAT=*AMERICAN)	'06/24/96176'

Example

The FILL function has four input parameters; it also returns exactly one return value, a string that is filled with fill characters to the specified length and in the specified direction.

Call:	Return value:
FILL('ABCDE', 8, SIDE=*RIGHT, FILL-BYTE=C'.')	'ABCDE...'
FILL('ABCDE', 8, SIDE=*LEFT)	' ABCDE'

8.3 Function groups with predefined functions

The predefined functions included in the SDF-P scope of supply can be divided into groups.

8.3.1 String functions

This section presents all functions that process or analyze strings.

String processing

The following functions process one or more strings and return a new string as a result:

Uppercase/ lowercase	`UPPER-CASE( )`	Changes lowercase letters to uppercase
	`LOWER-CASE( )`	Changes uppercase letters to lowercase
String length	`FILL( )`	Lengthens a string with fill characters
	`TRIM( )`	Removes repeated characters at the beginning or end of a string
Substring	`SUBSTRING( )`	Returns a specific substring
	`REPLACE( )`	Replaces a specific substring
SDF list	`EXTEND-SDF-LIST( )`	Appends a new element to an SDF list
New name	`RENAME( )`	Gives a new name to the specified string; wildcards may be used
Field extraction	`EXTRACT-FIELD( )`	Extracts a field from a string

String analysis

The following functions analyze a specified string:

Initial position	`INDEX( )`	Searches for the position of a substring within a full string
String length	`LENGTH( )`	Determines the length of a string
Character search	`VERIFY( )`	Checks whether particular characters are contained in the specified string
Pattern	`WILDCARD( )`	Checks whether a string contains a particular pattern
C-Literal	`IS-C-LITERAL( )`	Checks whether a string is a C literal
X-Literal	`IS-X-LITERAL( )`	Checks whether a string is an X literal
Number	`IS-INTEGER( )`	Checks whether a string represents an integer (i.e. can be converted into an INTEGER value)

SDF structure	SDF-STRUCTURE-VALUE( )	Returns the value of an SDF structure as a string
	IS-SDF-STRUCTURE()	Checks whether a string is an SDF structure
SDF list	IS-SDF-LIST( )	Checks whether a string is an SDF list (an SDF list is a string that is interpreted in accordance with the rules for operand lists in commands)
	SUBLIST( )	Returns an element in an SDF list
	SUBLIST-NUMBER( )	Returns the number of elements in an SDF list
SDF data type	CHECK-DATA-TYPE( )	Checks whether a string satisfies the SDF data type requirements
List variable	SEARCH-LIST-INDEX( )	Searches a list variable for a string (including regular POSIX expression)

Name checking

The following function checks whether the specified string complies with the necessary naming conventions:

Variable name	IS-VARIABLE-NAME()	

Functions for accessing variables

The functions presented in this section return as a return value a variable name, the contents of a variable or entries for building up a variable. They make it possible to process complex variables.

Complex variable	`FIRST-VARIABLE-NAME( )`	Returns the variable name of the first variable element
	`NEXT-VARIABLE-NAME( )`	Returns the variable name of the next variable element
Number of variable elements	`SIZE( )`	Returns the number of elements in a complex variable
Upper limit for list elements	`LIMIT( )`	Outputs the maximum number of elements that the specified list variable can contain
Value of an array index	`ARRAY-INDEX( )`	Returns the value of an array index, i.e. the array index that complies with the specified conditions

Variable attributes

Variable type	`CURRENT-TYPE( )`	Returns the current data type of a simple variable
Attribute occurrence	`VARIABLE-ATTRIBUTE( )`	Returns the value of the specified attribute
Variable declaration	`IS-DECLARED( )`	Checks whether a variable is declared
Initialization	`IS-INITIALIZED( )`	Checks whether a variable contains a valid value
Scope	`LAYOUT-SCOPE( )`	Returns the scope of a structure layout

8.3.2 Environment information

The functions presented in this section are used to obtain information on the calling task, the job, the procedure, user switches, job variables, system options, etc. Most of these functions are called without input parameters, since they are uniquely allocated to the value to be queried.

Information on a job/task/files

Job name	`JOB-NAME( )`	Returns the job name of the current task (the name specified in SET-LOGON-PARAMETERS)
Operating mode	`TASK-MODE( )`	Outputs the current operating mode of the current task
TSN	`TSN( )`	Returns the job number (TSN) of the current task
Priority	`RUN-PRIORITY( )`	Returns the priority level of the current task
Account number	`ACCOUNT( )`	Returns the account number of the task
User ID	`USER-IDENTIFICATION( )`	Returns the user ID for the current task
Default catalog ID	`STD-CAT-ID( )`	Returns the catalog ID of the default pubset for the user ID of the current task
Home pubset	`HOME-CAT-ID( )`	Returns the catalog ID (CATID) of the home pubset for the user ID of the current task
Catalog entry	`IS-CATALOGED-FILE( )`	Checks whether the specified file is cataloged
	`IS-CATALOGED-JV( )`	Checks whether the specified job variable is cataloged
	`IS-LIBRARY( )`	Checks whether the specified file is a library
	`IS-LIBRARY-ELEMENT( )`	Checks whether the specified library element exists
Call counter	`COUNTER( )`	Counts the COUNTER() calls
Program name	`PROG-NAME( )`	Returns the name of the program file currently loaded
Path name	`INSTALLATION-PATH( )`	Specifies the path name of a file in accordance with the product version
File contents	`IS-EMPTY-FILE( )`	Checks whether the file is empty

SYSFILE management: system files

SYSCMD	`SYSCMD( )`	Returns *PRIMARY or the name of the file to which SYSCMD is assigned
SYSDTA	`SYSDTA( )`	Returns *PRIMARY or *SYSCMD or the name of the file to which SYSDTA is assigned
SYSLST	`SYSLST( )`	Returns *PRIMARY or the name of the file to which SYSLST is assigned
SYSOUT	`SYSOUT( )`	Returns *PRIMARY or the name of the file to which SYSOUT is assigned

Date/time

Date	`DATE( )`	Returns the current date in the specified format
	`DATE-VALUE( )`	Returns a particular day's date in the specified format
	`ELAPSED-DAYS( )`	Returns number of days difference between two specified dates
Day of week	`DAY( )`	Returns the name of the current day abbreviated in the specified language
Month	`MONTH( )`	Returns the name of the current month abbreviated in the specified language
Time	`TIME( )`	Returns the current time to the second with any separator between the various units

System data

This section presents the functions that return system data relating to the hardware and software used and to the settings made by the system administrator for the current system. This does not include task-related data or job variable information.

Host name	`HOST( )`	Returns the internal name of the BS2000 computer on which the current task is running
Query: SDF-P on the system	`IS-SDF-P( )`	Returns TRUE if SDF-P is loaded on the system
System parameters	`SYSTEM-INFORMATION( )`	Returns the values of system parameters; input parameters as in the SINF macro
System ID	`SYS-ID( )`	Returns the system ID
Session number	`SESSION-NUMBER( )`	Returns the internal number of the current session
SDF-P version	`SDF-P-VERSION( )`	Returns the current version designations of the loaded subsystems SDF-P and SDF-P-BASYS

TIAM information

Station name	`STATION( )`	Returns the station name of the TIAM station
Device type	`STATION-TYPE( )`	Returns the device type of the TIAM station
Processor name	`PROCESSOR( )`	Returns the processor name of the TIAM station

Procedure information

The functions below check the settings of procedure attributes and return information on the current procedure.

Nesting level	`PROC-LEVEL( )`	Returns the nesting level of the S procedure at the time of the function call
Logging	`LOGGING-MODE( )`	Indicates whether the logging of commands or data is enabled for the current S procedure
Spin-off	`STMT-SPINOFF( )`	Indicates whether statement spin-off is enabled
User switch	`USER-SWITCH( )`	Queries the status of the specified user switch
Call	`EXPLICIT-CALL( )`	Returns the type of the procedure call
	`SYSTEM-CALL( )`	Returns the syntax file hierarchy level for the command calling the procedure

Job variables

The functions below return information on job variables that monitor the job or program. Job variables are part of the (chargeable) product JV (Job Variable System) and were used up until now in BS2000 procedures for procedure monitoring and control.

Job monitoring	`JOB-MONJV( )`	Returns the name of the monitor JV that monitors the current job
Program monitoring	`PROG-MONJV( )`	Returns the name of the monitor JV that supervises the current program
Contents	`JV( )`	Returns the contents of the specified job variable
Class	`JOB-CLASS( )`	Returns the job class of the current task
Catalog entry	`IS-CATALOGED-JV( )`	Checks whether the specified job variable is cataloged

8.3.3 Conversion functions

The functions presented in this section are used for explicit conversion.

String conversion for literals, C literals, X literals

`FROM-C-LITERAL( )`	Converts a C literal to the string it represents (reverse function of TO-C-LITERAL)
`FROM-X-LITERAL( )`	Converts an X literal to the string it represents (reverse function of TO-X-LITERAL)
`TO-C-LITERAL( )`	Converts a string to a C literal
`TO-X-LITERAL( )`	Converts a string to an external hexadecimal representation of the string
`BOOLEAN( )`	Converts an expression to BOOLEAN
`INTEGER( )`	Converts an expression to INTEGER
`INTEGER-TO-X-LITERAL( )`	Converts an integer to a 4-byte long X literal which contains the coding of the integer (inverse function to X-LITERAL-TO-INTEGER)
`STRING( )`	Converts an expression to STRING
`VARIABLE-TO-STRING( )`	Converts an S variable to an SDF string
`HASH-STRING( )`	Codes an expression as a string
`HASH-VALUE( )`	Codes an expression as an integer value
`X-LITERAL-TO-INTEGER( )`	Converts a string which is up to 4 bytes long to an integer (inverse function to INTEGER-TO-X-LITERAL)

Character by character (re)coding

`CHARACTER-TO-INTEGER( )`	Supplies the value in EBCDI code as an integer for the specified character
`INTEGER-TO-CHARACTER( )`	Supplies the character coded with this value in EBCDI code for a specified integer from 1 to 255
`TRANSLATE( )`	Replaces a string with another string defined by the user
`TRANSLATE-BOOLEAN( )`	Allocates another expression defined by the user to the result of a Boolean expression

8.3.4 Command return codes / error messages

The SDF-P and SDF commands return a standardized command return code that is made up of three components. These components indicate how the command was executed or whether an error occurred.

The command return code consists of the following three components: Subcode1 (SC1), designating the error class; Subcode2 (SC2), which returns additional information on the error class; Maincode, which returns a seven-byte error code to which a message text is allocated.

Request subcode1	`SUBCODE1( )`	Returns as a result the value 0 if no error has yet occurred, or the error class of the last error to occur; i.e. Subcode1 for the last command which was not correctly executed
Request subcode2	`SUBCODE2( )`	Returns additional information on the error class. SUBCODE2( ) needs to be called only if SUBCODE1( ) returned a value that was not equal to 0
Request maincode	`MAINCODE( )`	Returns the 7-byte error code that exactly describes the message class and the error; the error message for this code can be requested with the MSG( ) function or the command HELP-MSG
Request error message text	`MSG( )`	Returns the message text that is allocated to the specified message code, in the specified language

8.4 System administration functions

A prerequisite for the creation of user-written or system administration functions is that the system administrator has loaded the SDF-P-BIF subsystem. This contains the tools and macros required for users to be able to write and install their own functions.

There are Assembler interfaces with which the system administrator can create functions. These are described in section “Program interfaces for systems support” on page 305.

Because only the system administrator can develop user-written functions, he/she is also responsible for the correctness and security of these functions.

8.4.1 Naming conventions

The name of any system administration function must not be identical with any existing or future function, to avoid possible incompatibilities. Hence, the names of user-written functions should begin with an “X” in a similar way to SDF-A user commands, such as for example X-MY-BUILT-IN.

It should also be noted that the names of functions must not exceed 20 characters.

8.4.2 Creating programs

Programs which are used to implement user-created functions consist of two separate parts:

1. The syntax specification
   In this part, the syntax of the parameters of the function is specified. The code is generated in the BIFDESC macro.
2. The execution specification
   This part contains the code for the function. A parameter list is passed from the system to the function, containing an array of n records (string_length, string_ptr, value_type); n is here equal to the number of parameters of the function plus two (function_value, returncode) (the number of function parameters may not exceed 254). The BIFMDL macro contains the records with their values and the data structures.

8.4.3 Updating BIFTAB and objects

When a program has been written for a function, the system administrator must update the BIFTAB source, which is contained in SYSSRC.SDF-P-BIF.010. The BIFDEF macro generates a table entry which links the name of the function with the addresses of the executable module entry and the syntax specification entry.

The object generated for this BIFTAB module must be appended to the SYSLNK.SDF-P-BIF.010 file. This deletes any earlier BIFTAB module. This new module will then be loaded by an automatic link the next time that the SDF-P-BIF subsystem is started.

The objects generated for the syntax specification and the function code must also be appended to the SYSLNK.SDF-P-BIF.010 file. They too will be loaded by an automatic link the next time that the SDF-P-BIF subsystem is started, but only if the entries in the BIFTAB module have been defined.

The functions thus appended can be accessed after the next start of the SDF-P-BIF subsystem.

8.4.4 Parameter transfer

Parameter transfer is implemented using registers.

All the input parameters, plus the return values and return code are specified in a structure (address, length, type).

Apart from explicit types (string, integer and Boolean), it is also possible to specify keywords as input parameters.

For further details, see chapter “Program interfaces” on page 305.

Note

The system administration functions can only be called in TPR mode. In all other respects they behave like predefined functions when called.

8.4.5 Examples

The following function is to be implemented in Assembler and C:

```
XSUBSTRING[2/3](STRING = <string_expression>
                ,START = <arithm_expression>
                ,LENGTH = <arithm_expression> / *REST-LENGTH
               )
```

Assembler interface

Title of the function:	`XSUBSTRING2`
Entry:	`XSUBEX2`
Input data:	`STRING, START POSITION, LENGTH`
Output data:	`SUBSTRING`

First step: syntax specification.

```
BIFDESC    NAME='XSUBSTRING2',                -
           ENTRYN=(*CSECT,XSUBDEF2),          -
           PARLIST=                           -
            ((STRING,*STRING),                -
              (START,*INTEGER,1),             -
              (LENGTH,*INTEGER,*REST-LENGTH,  -
               *REST-LENGTH)                  -
              ),                              -
           PARFORM=BY-VALUE,
           VALTYPE=*STRING
END
```

Second step: execution program

```
XSUBEX2     CSECT
XSUBEX2     AMODE     ANY
XSUBEX2     RMODE     ANY
            STM       14,12,12(13)          Save the caller’s registers
            BASR      10,0                  R10: base register
            USING     *,10
            USING     DSMDL1,4
            USING     DSMDL2,5
*
            L         1,0(0,1)              R1 ->a(p1)    => R1=a(p1)
*
            LA        5,48(1)               R5 -> fifth element of the
                                            operand list (e.g. RC)
            L         5,4(5)                R5 -> RC
            MVC       0(9,5),OKRC           Set RC = OK
*
            LA        4,0(1)                operand list starts at R4
            L         6,BIF1VLG             R6: length of the STRING operand
            L         7,BIF1VPT             R7: addr. of the STRING operand
            LTR       6,6                   Is the length of STRING=0?
            BE        MSGNULL               Send a message NULL_STRING
*
            LA        4,12(1)               Second element of the operand
                                            list follows at R4 (e.g. START)
            LA        2,1(0,0)              R2 = 1
            L         8,BIF1VPT             R8: address of the START value
            L         8,0(8)                R8: value of START
            CR        8,2                   If START < 1
            BL        MSGBOUN               Send message OUT_OF_BOUNDS
            CR        8,6                   If START > length of STRING
            BH        MSGBOUN               Send message OUT_OF_BOUNDS
*
            LA        4,24(1)               Third element of the operand
                                            list follows at R4 (e.g. LENGTH)
*                                           R3 = len. of STRING - START + 1
            LA        3,1(0,0)              R3 = 1
            AR        3,6                   R3 = R3 + length of STRING
            SR        3,8                   R3 = R3 - START
            CLI       BIF1VTY,BIF1INT
            BE        LENINT
LENKEYW     DS        0H
*                                           *Look for REST-LENGTH
            LR        9,3
            B         @0001
```

```
LENINT      DS      OH
*                                          LENGTH = *INTEGER
            L       9,BIFTVPT              R9: address of the LENGTH value
            L       9,0(9)                 R9: value of LENGTH
            CR      9,2                    If LENGTH < 1
            BL      MSGBOUN                Send message OUT_OF_BOUNDS
            CR      9,3                    If LENGTH < REST-LENGTH
            BNH     @0001                  Then ok
            LR      9,3                    LENGTH is truncated
            MVC     0(9,5),TRUNCRC         Set RC = TRUNCATED
@0001       DS      OH
                                           Calculate the function value
            LA      4,36(1)                Fourth element of the operand
                                           list (e.g. FUNCTION_VALUE)
                                           follows at R4
            L       6,BIF1VPT              R6 : address of function value
            AR      8,7                    R8 = address of STRING + START
            SR      8,2                    R8 = R8-1
            LR      7,9                    R7 = length of function value
            ST      7,BIF1VLG
            MVCL    6,8                    Copy into function value
*           B       RETURN
MSGNULL
            MSG7X   MF=E,PARAM=NULLPL
            MVC     0(9,5),NULLRC          Set RC = NULL_STRING
*           B       RETURN
MSGBOUN
            MSG7X   MF=E,PARAM=BOUNDPL     Set RC = OUT OF BOUNDS
*           MVC     0(9,5),BOUNDRC
RETURN
            LM      14,12,12(13)           Write back caller's registers
OKRC        BR      14                     Return to the caller
            DC      XL1'00'
            DC      XL1'00'
TRUNCRC     DC      CL7'CMD0001'
            DC      XL1'02'
            DC      XL1'00'
BOUNDRC     DC      CL7'SDP0414'
            DC      XL1'00'
            DC      XL1'01'
NULLRC      DC      CL7'SDP0412'
            DC      XL1'00'
            DC      XL1'01'
            DC      CL7'SDP0411'
            DS      OF
```

```
NULLPL      MSG7X    MF=L,ID=SDP0411
BOUNDPL     MSG7X    MF=L,ID=SDP0412
            DS       OF
DSMDL1      BIFMDL1  MF=D
            DS       OF
DSMDL2      BIFMDL2  MF=D
            END
```

Third step: update BIFTAB

```
BIFTAB     CSECT
           .....
XSUBSTR2   BIFDEF   MF=L,NAME=XSUBSTRING2,SYNTAX=XSUBDEF2,CODE=XSUBEX2
           .....
           END
```

9 Expressions

Expressions determine how a new value is calculated on the basis of specified values. The operands, which are linked together by operators, can be base terms or can themselves be expressions. Base terms are terms within an expression that are not further divisible, i.e. that do not contain operators.

In the simplest case, an expression consists of a single operand; the value of the expression is then the value of the operand. This type of expression and expressions that consist of a single operator whose operands are base terms are called “simple expressions”, as opposed to complex expressions in which at least one of the operands is an expression. This expression can be either a simple or a complex expression.

The following terms are combined under the generic term “base term”:

- numbers
- Boolean constants
- string literals
- variable names
- function calls

The operators are divided into four categories:

- arithmetic operators
- logical operators
- relational operators
- concatenation operators

The data type of an expression without operators is determined by the data type of the base term. Otherwise, the way in which the operators are combined determines the data type of the expression. There are thus three types of expression:

- arithmetic expressions
- logical or Boolean expressions
- string expressions

This chapter begins by describing the base terms, followed by the operators, expression types and, finally, the rules of syntax, interpretation and evaluation for expressions.

9.1 Base terms

Base terms are terms within an expression that are not further divisible, i.e. terms that do not contain another operator.

The table below shows the different base terms with their representation in metasyntax:

Base term	**Representation**
Number	<integer>
String literals	<c-string> / <x-string>
Boolean constant	<boole-const>
Variable name	<composed-name>
Function call	<functioncall>

These base terms are described in detail in the sections below.

9.1.1 Numbers

Only integers are allowed in expressions.

Data type	<integer>
Character set	0 ... 9, +, -
Value range	$-2^{31} \leq \text{number} \leq +2^{31}-1$

Numbers must not be enclosed in single quotes, since they would then be interpreted as strings.

Numbers can be linked together by means of arithmetic operators and relational operators.

Numbers can also be inserted in a new expression as the contents of a variable, as a result of a function call or as a result of an expression.

Example

Correct integer assignments:

```
/A = -12345
/B = 3456
/C = +1287
```

The variables A, B and C are initialized with the specified numeric values.

Incorrect integer assignments:

```
/D = +123.5
/E = '+1234'
```

When variable D is initialized, SDF-P reports a syntax error in the variable name (+123.5 is not interpreted as a value). Variable E is initialized with the string ’+1234’; as soon as this variable is inserted in an arithmetic operation, SDF-P reports an incorrect data type (providing E has the data type ANY or STRING; otherwise, an error occurs during assignment).

9.1.2 Boolean constants

There are two Boolean constants: TRUE and FALSE. Permissible synonyms for TRUE and FALSE are YES/NO and ON/OFF. These names are reserved names and therefore cannot be used as variable names. The table below defines the data type and shows the names that are valid for Boolean constants:

Data type	<boole-const>
Value range	TRUE, FALSE
Character set	TRUE, YES, ON, FALSE, NO, OFF

Boolean constants must not be enclosed in single quotes, since they would then be interpreted as strings.

Boolean constants can be linked to logical expressions by means of relational or logical operators.

Boolean constants are specified directly. The results returned by Boolean variables, expressions and functions are Boolean values that equal either TRUE or FALSE.

Example

```
/DECLARE-VARIABLE SWITCH-1(TYPE=*BOOLEAN)
/DECLARE-VARIABLE SWITCH-2(TYPE=*ANY)
/SWITCH-1 = ON     “Correct assignment”
/SWITCH-2 = 'OFF'  “Incorrect assignment”
```

Both assignments are syntactically correct, but only SWITCH-1 can later be inserted in a logic operation as a Boolean constant. SWITCH-2 was assigned a string, which means that it can only be inserted in relational and string operations; otherwise, SDF-P reports an incorrect data type.

9.1.3 String literals

A string literal is a sequence of any characters that is enclosed in single quotes. In the literature, string literals are also called character strings; in this manual, however, the two terms are not identical: a character string does not need to be enclosed in single quotes. ABC, for example, is a character string, while ’ABC’ is a string literal.

A string literal may be represented in one of two ways, namely as C string or as an X string.

Data type	**<c-string>**	**<x-string>**
Character set	All EBCDIC characters	Hexadecimal digits (0 ... F)
Length	Freely selectable	Freely selectable
Representation	[C]'.......'	X'......'
Null string	[C]''	X''

Note

Internally, C strings and X strings are represented identically, which means that a C string can also be represented as an X string.

The terms C string and X string refer to how the bytes comprising the character string are represented:

– In a C string, each byte is represented by its EBCDIC character (C stands for “character”). Consequently, its character set is the entire EBCDI code; this also means that uppercase and lowercase are retained.
– In an X string, each byte is represented by the resulting EBCDIC character half-bytes; thus, the X string contains a sequence of paired representations of the left and right half-bytes. Consequently, its character set is the digits of the hexadecimal number system, i.e. the digits from 0 to F.
– If an odd number of hexadecimal digits is specified for an X string, the string is filled internally from the left with zeros. (Example: X’123’ becomes X’0123’.)

– The null string constitutes a special case. It contains pairs of single quotes only, which can follow a C or X (C”/X”). The null string must not be confused with a space string (C'␣' / X’40’).

Strings are always enclosed in single quotes; at the same time, any character strings or numbers that are enclosed in single quotes are interpreted as strings.

If the single quote is not preceded by a character, the string is normally considered to be a C string. An X string must be preceded by an X. Other letters are not allowed and result in an error.
Strings can be linked together by means of relational operators or concatenation operators. Strings can also be inserted in a new expression as the contents of a variable, as a result of a function call or as a result of an expression.

Note

The designation “string_expression“, used as a parameter value in predefined functions, stands for any of the following values:

– a string enclosed in quotes (<c-string>)
– the name of a variable containing a string (<composed-name>)
– an expression that returns a string as result

Example

```
/JV-NAME = 'MY-JV'
/MY-VAR = JV('JV-NAME')      value of the job variable JV-NAME
/MY-VAR = JV(JV-NAME)        value of the job variable MY-JV
/MY-VAR = JV('&JV-NAME')     value of the job variable MY-JV
/MY-VAR = JV(JV(JV-NAME))    value of the job variable whose name
                             is stored in job variable MY-JV
```

Example

```
/A = 'ABCD'
/B = C'ABCD'
/C = X'C1C2C3C4'
```

The variables A and B are assigned the same C string, while variable C is assigned an X string that yields the string ABCD when evaluated. All three variables thus have identical contents.

9.1.4 Variable names

A variable name that designates a simple variable can be a component in an expression, provided that this variable is already initialized, i.e. has valid contents.

The rules for the formation of variable names are described in section “Variable names” on page 150.

When an expression is calculated, not the variable name but the contents of the variable are inserted.

The contents of a variable can have the data type INTEGER, BOOLEAN or STRING, i.e. they can be a number, a Boolean value or a string. Consequently, the rules that apply for evaluating variable contents are the same as those described in preceding sections for numbers, Boolean constants and strings.

If the contents of a variable are to be evaluated, the variable name must not be enclosed in single quotes. If it is enclosed in single quotes, the variable contents are not evaluated and the variable name is interpreted as a string.

Example

The following variable declarations and assignments are made:

```
/SET-VARIABLE A = 36
/SET-VARIABLE B = 72
/DECL-VARIABLE C
```

These variables are used in (simple) expressions:

`/D = A + B`	“correct”
`/E = A - C`	“incorrect”

The first assignment is valid, since variables A and B are correctly initialized. The second assignment in which variable E is to be initialized is invalid, since variable C is declared but not initialized.

9.1.5 Function call

Similar rules apply for the use of function calls in expressions as for the use of variable names. Instead of the function name being inserted in the calculation of the expression, the result that is returned by the function thus called is inserted.

For a description of how functions are called and which functions are available in structured procedures, see chapter “Functions” on page 223.

The result of the function can have the data type INTEGER, BOOLEAN or STRING, i.e. it can be a number, a Boolean value or a string. Consequently, the rules that apply to function results are the same as those described in preceding sections for numbers, Boolean constants and strings.

If the result of the function call is to be evaluated, the function call must not be enclosed in single quotes. If it is enclosed in single quotes, the function name is interpreted as a string.

If variables and functions have the same name, the variable is evaluated in accordance with the rules for S procedures. Any confusion can be avoided by specifying the identifying parentheses in the function call.

9.2 Operators

Operators link base terms to simple expressions, which can then be linked by means of operators to complex expressions, and so on.

Since some operators are also allowed as special characters in names (such as variable names), operators should generally be surrounded by spaces. Otherwise, some operators cannot be interpreted correctly. For example, the minus sign (-) can be interpreted as a hyphen.

9.2.1 Arithmetic operators

Arithmetic operators are used for performing arithmetic operations on numbers. Thus, arithmetic operators link numbers. Numbers can be specified as numeric literals (equivalent to a numeric constant) or as variables containing a valid numeric value. In complex expressions, arithmetic operators link expressions whose result is a numeric value.

The result of an arithmetic operation is always a numeric (= arithmetic) value, i.e. a number with the data type INTEGER in the range from -2^{31} to $+2^{31}-1$.

Operation	**Operator**
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulo	MOD

9.2.1.1 Addition

The addition operator is the plus sign (+).

Rules:

- For the plus sign as an operator in an addition operation:
  The result value must be greater than or equal to -2^{31} and less than or equal to $+2^{31}-1$.
- For the plus sign as a positive sign:
  The plus sign must be directly followed by the numeric literal or variable name.

Example

```
/A = +45
/B = 36
/C = 45 + 5
/D = A+ B
```

All these assignments and the simple addition expressions are valid.

9.2.1.2 Subtraction

The subtraction operator is the minus sign (-).

Rules:

For the minus sign as an operator in a subtraction operation:
- The minus sign must be preceded or followed by a space so that it will not be mistaken for a hyphen (see example).
- The result value must be greater than or equal to -2^{31} and less than or equal to $2^{31}-1$.

For the minus sign as a negative sign:
- The minus sign must be preceded by a space.
- The minus sign must be directly followed by the numeric literal or variable name.

Example

```
/A = -45
/B = -45 - 5
/C = A - B
/D = A-B
```

All these assignments are syntactically correct: variables A, B and C are assigned integer values (-45, -50, 5) and variable D is assigned the contents of a variable A-B, assuming this variable is declared and initialized. If there is no variable with the name A-B or if it is not initialized, this assignment results in an error.

9.2.1.3 Multiplication

The multiplication operator is an asterisk (*).

Rules:
The result value must be in the range from -2^{31} to $+2^{31}-1$.

9.2.1.4 Division

The division operator is a slash (/). When using division, it is important to note whether the quotient is an integer, i.e. whether the dividend is a multiple of the divisor. If it is not, the remainder can be determined by means of a modulo operation (see section “Modulo operation” on page 259).

Rules:

- The result is an integer (without remainder) in the range from -2^{31} to $+2^{31}-1$.
- The result of the division operation is calculated as follows:
  - If the quotient is an integer (i.e. there is no remainder), the quotient is inserted as the result.
  - If the quotient is not an integer (i.e. there is a remainder), the result is rounded down to the next number, which is provided with the sign yielded by the division operation.
- Division by zero results in an error.

Example

Assignment	Result
/A = 7	
/B = -4	
/C = A / B	-1
/D = A / 2	3

The division operation 7 : -4 returns the quotient -1.75. Since only an integer can be inserted as the result of a division operation, variable C is not rounded down to the next integer (-2); instead, it is rounded down from the unsigned amount, which is then provided with the sign yielded by the division operation: C = -1
The division operation 7 : 2 returns the quotient 3.5. Variable D is rounded down to the next integer, which is provided with the sign yielded by the division operation: D = 3.

9.2.1.5 Modulo operation

The modulo operation returns the integer remainder of a division operation; the operator is the reserved name MOD.

Rules:

- The operator name MOD must be preceded and followed by a space; otherwise, this string is interpreted as part of a variable name.
- The result is calculated according to the following formula:

```
A MOD B = A - (A / B) * B
```

Example

Assignment	Result
/Y = 9 MOD 4	1
/Y = -9 MOD 4	-1

Table 1:

The variable Y is assigned the value 1, since the expression is calculated as follows:

```
9 MOD 4 = 9 - (9 / 4) * 4
```

First the parentheses are solved: The division operation 9 : 4 yields the number 2.25; consequently, the number 2 is inserted in the equation:

```
9 MOD 4 = 9 - 2 * 4
```

According to the rules of arithmetic, this yields 9 - 8, i.e.:

```
9 MOD 4 = 1
```

Consequently, the value -1 is yielded and assigned to variable Z.

```
-9 MOD 4 = -9 - (-9 / 4) * 4
                  = -9 - (-2) * 4
                  = -9 - (-8)
                  = -1
```

9.2.2 Relational operators

Relational operators are used in simple expressions to compare two base terms of the same type. They are used in complex expressions to compare expressions, the results of which must have the same data type.

Comparison	Operators		
Less than	LT	<	
Less than or equal to	LE	<=	
Equal to	EQ	=	==
Not equal to	NE	<>	
Greater than or equal to	GE	>=	
Greater than	GT	>	

The result of a relational operation is always a Boolean value, i.e. a value that is either FALSE or TRUE.

The same rules apply to all relational operators; therefore, these rules are described only once.

Rules:

– The operands of a relational operator must be of the same type; otherwise, an error message is issued and error handling is activated.
– The result of a relational operation is either TRUE or FALSE.
– If the relational operator is an equals sign (=), the relational expression must be enclosed in parentheses to distinguish a comparison of equality from the assignment of a value to an operand (operand_1 = operand_2). If the equals sign is duplicated, the parentheses can be omitted.

Example

```
/B = A + COUNT
/IF (B = A + COUNT)
```

The first line contains an assignment: variable B is assigned the results yielded by adding the contents of the variables A and COUNT.

The second line contains a relational comparison: if the contents of variable B, which were set at another position in the procedure, correspond to the results yielded by adding A and COUNT, the THEN branch of the IF block is executed. The contents of variable B are not modified, nor is it assigned a new value. In order to make the difference between an assignment and a comparison more clear, the relational operator can optionally be written as “==”:

```
/IF (B == A + COUNT)
```

Numeric comparison

A “numeric comparison” is when both operands of the relational operator are integer expressions. The values of the operands are compared.

Comparison of Boolean values

Both operands of the relational operator must be Boolean expressions.

Rules:

Only the following operators are allowed:

Operation	**Operators**		
Equal to	EQ	=	==
Not equal to	NE	<>	

String comparison

“String comparison” means that both operands of the relational operator are string expressions.

Rules:

- Strings are compared character by character (i.e. byte by byte), from left to right, until the first difference between characters is detected.
- The first difference between characters determines which string is greater or less; the other characters are no longer taken into account for the comparison.
- The terms “greater” and “less” are based on the order of the characters in EBCDI code, from X’00’ to X’FF’.
- If the lengths of the two strings differ but have the same character string up to the last character of the shorter string, the shorter string is considered to be less.
- Strings are equal if they are the same length and have exactly the same characters.

A character-by-character or byte-by-byte comparison means that the EBCDIC equivalents for the characters are examined.

Example

Assignment	**Result**
/A = 'ABC'	
/B = 'ABCDE'	
/C = X'C1C2C3'	
/D = 'B'	
/E = (B > A)	TRUE
/F = (D > A)	TRUE
/G = (C = A)	TRUE
/H = (B = A)	FALSE

Variable E is assigned the Boolean value TRUE, since the first three characters of the strings in variables B and A are identical but the string in variable A (’ABC’) is shorter and therefore less than the string in variable B (’ABCDE’).

Variable F is also assigned the Boolean value TRUE: string ’B’ (in variable D) is shorter that the string ’ABC’ (in variable A) but the first character in the string ’B’ has a higher value in EBCDI code than the first character in the string ’ABC’.

A comparison of the variable contents of C and A returns equality, since the X string with which the C variable was initialized is the half-byte notation for the string ’ABC’; consequently, variable G is assigned the Boolean value TRUE.

Variable H is assigned the Boolean value FALSE, since the strings in variables B and A are not equal.

9.2.3 Logical operators

Logical operators link together two Boolean expressions (exception: NOT. This operator applies only to a single Boolean expression).

The result yielded by linking logical operators is always a Boolean value (TRUE or FALSE) that can be addressed by means of one of the names reserved for Boolean constants.

Operation	**Operator**
Negation	NOT
Or	OR
And	AND
Either Or (= exclusive or)	XOR

Rules:

– The rules for logic operations apply.
– NOT inverts the value of an expression.

Example

```
/A = TRUE
/B = 4
/C = 20
/D = A OR (B > C)
/E = A AND (B > C)
```

Variable D is assigned the value TRUE, since one of the operands has the value TRUE in the OR operation.
Variable E is assigned the value FALSE, since only one of the two operands of the AND operation has the value TRUE.

9.2.4 Concatenation operator

The concatenation operator // concatenates two string expressions.

The result of concatenation is always a string.

Rule:
The strings specified as operands are concatenated contiguously without gaps.

Example

```
/A = 'Date: '
/B = DATE(FORMAT = *AMERICAN)
/C = A // B
```

The C string ’Date: ’ is assigned directly to variable A; variable B is assigned the result of the function DATE as a string. These strings are concatenated to form a new string, which is assigned to variable C. C then has the following contents: ’Date: 06/26/96’.

9.3 Expression types

The type of an expression is always identical to the data type of the result value. There are three types of expression, corresponding to the three data types:

- arithmetic expressions
- logical or Boolean expressions
- string expressions

The type of a simple expression is determined by the operator that links together the base terms. The table below lists simple expressions and their components:

Type	**Operators**	**Base terms**	**Result data type**
Arithmetic expression	Arithmetic	Numbers Variable names Function calls	Integer
Relational expression	Relational	Numbers Boolean constants String literals Variable names Function calls	Boolean
Logical or Boolean expression	Logical	Boolean constants Variable names Function calls	Boolean
String expression	Concatenation	String literals Variable names Function calls	String

For complex expressions, the data type of the result, and thus of the expression, is determined by the last operator to be evaluated. The order of precedence for operators and the way in which expressions are evaluated is described in the next section.

9.4 Evaluation of expressions

Simple expressions contain only one operator; they are evaluated as shown above in the descriptions of the operators.

Complex expressions must first be divided into subexpressions until only simple expressions remain. The order in which the subexpressions are evaluated is determined by the priority of the operators. The user can control evaluation by inserting parentheses.

9.4.1 Operator priority

Operator priority is evaluated in two steps: firstly, by the order of precedence of operator types; secondly, within an operator type by the order of precedence for operators.

Order of precedence for “operator types”:

1. Sign, negation
2. Arithmetic operators
3. Concatenation operator
4. Relational operators
5. Logical operators

Order of precedence for arithmetic operators (“Dot operations before line operations”):

1. Multiplication, division, modulo operation (*, /, MOD)
2. Addition, subtraction (+, -)

Order of precedence for logical operators:

1. AND operation (AND)
2. OR operation (OR, XOR)

Relational operators:
All relational operators have the same order of precedence.

Example

Complex logical expressions are often used in IF blocks in the CONDITION operand of the IF command. If the condition determined by the expression is true, the command that follows the IF command is processed. If the condition is false, the next ELSE-IF or ELSE command is processed (for more information on IF, ELSE-IF, and ELSE commands, see chapter “Creating S procedures” or chapter “SDF-P commands”).

For example, an IF command could contain the following condition:

```
A + B / C > D + C MOD E AND A + D * E < D * C OR F // G > H
```

At the time of the IF query, variables A to H have the following values:

```
/A = 4
/B = 29
/C = 9
/D = 3
/E = 5
/F = 'ABC'
/G = 'DEF'
/H = 'ABCDE'
```

The expression is evaluated in the following steps:

1. Arithmetic operators: Multiplication / division

Operation	Corresponds to	Result
B / C	29 / 9	3
C MOD E	9 MOD 5	4
D * E	3 * 5	15
D * C	3 * 9	27

Results of step 1:
```
A + 3 > D + 4 AND A + 15 < 27 OR F // G > H
```

2. Arithmetic operators: Addition

Operation	Corresponds to	Result
A + 3	4 + 3	7
D + 4	3 + 4	7
A + 15	4 + 15	19

Results of step 2:
```
7 > 7 AND 19 < 27 OR F // G > H
```

3. Concatenation operator

Operation	Corresponds to	Result
F // G	'ABC' // 'DEF'	'ABCDEF'

Results of step 3:
```
7 > 7 AND 19 < 27 OR 'ABCDEF' > H
```

4. Relational operators

Operation	Result
7 > 7	FALSE
19 < 27	TRUE
'ABCDEF' > 'ABCDE'	TRUE

Results of step 4:
```
FALSE AND TRUE OR TRUE
```

5. Logical operators: AND

Operation	Result
FALSE AND TRUE	FALSE

Results of step 5:
```
FALSE OR TRUE
```

6. Logical operators: OR

Operation	Result
FALSE OR TRUE	TRUE

Results of step 6:
```
TRUE
```

Thus, the condition is fulfilled.

Order of evaluation

The order of evaluation is not defined for any operands. It can occur that the right operand of an AND operation is evaluated first.

Example

```
/DECLARE-VARIABLE I(TYPE = INTEGER)
/IF IS-INITIALIZED ('I') AND (I < 10)
```

Since, at this time, I is not yet initialized, evaluating I < 10 results in an error.

9.4.2 Parentheses

Parentheses ( ) can serve to divide up an expression, thus making its evaluation clearer. They can also be used to control evaluation.

If an expression contains parentheses, the subexpressions in the parentheses are evaluated first, according to how the parentheses are nested. Afterwards, the operators outside the parentheses are processed.

The limit on the number of pairs of parentheses is dependent on the complexity of the expression. However, up to 50 are always accepted.

Example

The example below shows how expression evaluation can be controlled by the use of parentheses:

```
(A + B) / C > (D + C) MOD E AND ((A + D) * E < (D * C) OR (F // G) > H)
```

10 Optimizing S procedures

Taking the performance into consideration, the use of large, complex S procedures is often judged to be less than optimal. The following chapter should provide the programmer of S procedures a few helpful tips and hints for writing fast S procedures. Since the solutions to problems are often implemented in S procedures is different manners, the following tips and hints show the recommended syntax (higher performance) and a less recommended syntax for comparison purposes. The optimization capabilities are shown for the following subjects:

- SDF syntax analysis
- Use of variables
- Procedure calls
- Searching for strings in a list
- Use of comments
- Program calls in the procedure

All optimization capabilities are presented again at the end of the chapter using a sample procedure.

10.1 SDF syntax analysis

SDF provides a number of capabilities to simply command input such as abbreviations (aliases), implicit variable declaration, use of positional and keyword operands. The utilization of these capabilities, which are primarily intended to ease the entering of input in the dialog, requires more steps when analyzing the syntax and can therefore affect the performance in the procedure mode.

Recommended syntax	**Syntax with lower performance**

Example 1:
`/FOR I=*COUNTER(FROM=1, TO=50)`
`/   DECLARE-VARIABLE VAR&I`
`/END-FOR`
	
	`/FOR I=*COUNT(1,50)`
	`/   DEC-VARI VAR&I`
	`/END-FOR`

Example 2:
`/SHOW-VAR VARIABLE-NAME=*ALL,-`
`/   INFORMATION=*PARAMETERS(-`
`/  NAME=*FULL-NAME(LIST-INDEX-NUMBER=*YES))`
	
	`/SHOW-VAR *ALL,LIST-INDEX=Y`

Example 3:
`/DECL-VAR (TST1, TST2, TST3),TYPE=*STRING`
	`/DECL-VAR TST1,TYPE=*STRING`
	`/DECL-VAR TST2,TYPE=*STRING`
	`/DECL-VAR TST3,TYPE=*STRING`

Example 4: `/I = 1`	 `/SET-VARIABLE I = 1`

The syntax analysis is made significantly less complicated in the following cases:

– Names of commands/statements, operands and keywords are fully specified (alternative: use aliases). The use of minimal aliases, which is sometimes recommended to avoid compatibility problems, is insufficient with respect to the performance (see Example 1).
– Suboperands are not specified outside of their structure (see Example 2).
– If several objects can be specified at the same time in a command or statement (e.g. by specifying a list or a wildcard string), then the syntax analysis only needs to be performed once. In contrast to this, the syntax analysis must be performed every time when the command or statement for every object is reentered (see the declarations of the string variables TST1, TST2 and TST3 in Example 3).
– The SDF syntax analysis can also be avoided in the case of the SET-VARIABLE commands when the alias of the command is used (i.e. without the command name, see Example 4).

Taking these points into consideration when developing or writing S procedures may appear to be complicated, but remember that a procedure is only written once, but is called often.

10.2 Using variables

10.2.1 Grouping commands that use a variable

SDF-P uses a buffer for the last variables used. The buffer can only hold a limited number of elements. If possible, commands that use the same variable should be grouped near each other within the procedure. This will reduce the access time for the variable.

10.2.2 Supplying list variables with values

A list variable that consists of simple variables (of type ANY, STRING, INTEGER or BOOLEAN) should be initialized in a command. The initialization of list variables using more than one command should be avoided.

Recommended syntax	**Syntax with lower performance**

`/DECLARE-VARIABLE V(TYPE=*STRING),-`
`/        MULTIPLE-ELEMENT=*LIST`
`/V=*STRING-TO-VAR('(AA,BB,CC)')`
	`/DECLARE-VARIABLE V(TYPE=*STRING),-`
	`/        MULTIPLE-ELEMENT=*LIST`
	`/SET-VARIABLE V = 'AA',*EXTEND`
	`/SET-VARIABLE V = 'BB',*EXTEND`
	`/SET-VARIABLE V = 'CC',*EXTEND`

10.2.3 Creating unneeded variables

The output of the SHOW command is often redirected with the EXECUTE-CMD command (or also ba assigning an S variable stream) into a complex variable so that the variables then created can be evaluated in a FOR loop. You should note, however, that for many SHOW commands the caller can affect the amount of information output, and therefore also the number of variables created, via the appropriate operands (e.g. INFORMATION=... or SELECT=...). The creation of variables while avoiding unneeded variables is preferred to the otherwise necessary IF-THEN-ELSE construction in the FOR loop.

10.3 Using predefined functions

SDF-P recognizes a function by the parentheses that follow the function name. The parentheses are optional if the function call does not contain any input parameters. When the parentheses are left out, SDF-P interprets the function name as a variable first (see section "Function call" on page 223). Using parentheses not only avoids this confusion, but it also yields better performance since SDF-P does not have to search for a variable first.

Recommended syntax	Syntax with lower performance
`/WRITE-TEXT 'It is now: &(TIME())'`	`/WRITE-TEXT 'It is now: &(TIME)'`

10.4 Procedure calls

10.4.1 Calling with CALL-PROCEDURE or INCLUDE-PROCEDURE

Since S procedures (analysis and execution via SDF-P) as well as non-S procedures (analysis and execution via SYSFILE) can be called with the CALL-PROCEDURE command, you cannot tell from the call alone which type of procedure is being called. For historical reasons, it is initially assumed that a CALL-PROCEDURE calls a non-S procedure, i.e. the procedure is analyzed first by the SYSFILE system component. Only after this analysis will an S procedure be analyzed by SDF-P.
In contrast to this, the INCLUDE-PROCEDURE command can only be used to call S procedures, i.e. the analysis is always performed by SDF-P.

If the caller does not care about the different variable models that the two command calls are based on, then an S procedure should be called with the INCLUDE-PROCEDURE command. This will avoid the unnecessary analysis by the SYSFILE component.
The call using INCLUDE-PROCEDURE commands can also be defined with SDF-A ≥ V4.1A for commands implemented in procedures (see the "SDF-A" manual [16]).

10.4.2 Procedures as library elements

Since the introduction of the COMPILE-PROCEDURE command, S procedures that are stored as PLAM library elements can be of element type J or SYSJ. Element type J is recommended for text procedures and element type SYSJ for compiled procedures. The element type can be specified when calling with the CALL-PROCEDURE or INCLUDE-PROCEDURE command. The default value is TYPE=*STD, i.e. only when an element of type SYSJ does not exist will the element of type J be called. An additional library access is required when this method is used. The additional access can be prevented when the caller knows the element type and also explicitly specifies it.

10.4.3 Passing parameters or information

Recommended syntax	**Syntax with lower performance**

Procedure 1:
 	
`/GET-IDOM-LIBRARY-NAME LOGICAL-ID='SYSSPR',-`
`/                   INT-LIB=IDOM-GLB-SYSSPR`
	Procedure 1:
	
	`/ASSIGN-STREAM SYSINF,-`
	`/       TO=*VAR(VAR-NAME=INT-LIB-NAME)`
	`/GET-IDOM-LIBRARY-NAME LOGICAL-ID='SYSSPR'`
	`/ASSIGN-STREAM SYSINF,TO=*SAME`
	`/IDOM-GLB-SYSSPR=INT-LIB-NAME#1.NAME`

Procedure 2 for implementing the command GET-IDOM-LIBRARY-NAME:
 	
`/ . . .`
`/DECLARE-PARAMETER INT-LIB(TYPE=*STRING,-`
`/         TRANSFER-TYPE=*BY-REFERENCE)`
`. . .`
`/INT-LIB = 'xxx'`
	Procedure 2 for implementing the command GET-IDOM-LIBRARY-NAME:
	
	`/INT-LIB.NAME='xxx'`
	`/TRANSMIT-BY-STREAM STREAM-NAME=SYSINF,-`
	`/         VAR-NAME=INT-LIB,-`
	`/         RETURN-VAR=*NONE`

If output information is to be returned in variables by a procedure that is called with CALL-PROCEDURE (directly or as an implemented procedure like in the example), then the following parameter declaration should be used:

```
/DECLARE-PARAMETER <var-name>,(...,TRANSFER-TYPE-*BY-REFERENCE)
```

This declaration is the best way to exchange information for simple variables (type ANY, STRING, INTEGER or BOOLEAN), in contrast to the use of variable streams as shown in the example on the right).

10.5 Searching for a string in a list

Recommended syntax	Syntax with lower performance

`/MATCH = SEARCH-LIST-INDEX(-`
`/    LIST-VARIABLE-NAME = X,-`
`/    PATTERN = P)`
	`/LOOP: FOR I = *COUNTER(1,SIZE('X'),1)`
	`/   IF (INDEX(X#I,P) <> 0)`
	`/      SET-VARIABLE MATCH = I`
	`/      EXIT-BLOCK LOOP`
	`/   END-IF`
	`/END-FOR`

The SEARCH-LIST-INDEX function was developed specially to improve performance. It searches through a list variable for a string or a regular expression in a call and returns the index of the first match. Using this function avoids the time-consuming search in a FOR loop.

10.6 Comments

Recommended syntax	Syntax with lower performance

`/ &* End-of-line comments`	`/REMARK Bad comments possibility`
	`/  “Not so good comments”`

End-of-line comments (introduced by the characters &*) should be the preferred method. Comments that are enclosed in quotation marks should be avoided. REMARK commands that are only used to insert a comment should never be used.

10.7 Program calls in procedures

Using the EDT and LMS utility routines in a procedure

If the EDT and LMS utility routines are called in the same procedure, then you must make sure that both utility routines offer some functions that are also offered by the other utility routine. For example, a library element can be opened and edited in EDT (@OPEN statement), and a a library element can also be edited in LMS (EDIT-ELEMENT statement).

Every time you switch programs while processing a library element, additional time is needed to unload the one program and then load the other program. Direct processing in a single utility routine (EDT or LMS) is preferred over processing in several program calls (see the following example):

1. Call LMS - extract element from file - terminate LMS
2. Call EDT - read in and process file - terminate EDT
3. Call LMS - store file in element again - terminate LMS

Recommended syntax	**Syntax with lower performance**
`/START-LMS`	`/START-LMS`
`//OPEN-LIBRARY LIB=&(LIB-1)`	`//OPEN-LIBRARY LIB=&(LIB-1)`
`//EDIT-ELEMENT ELEM=&(ELEM-1),-`	`//EXTRACT-ELEMENT ELEM=(&(ELEM-1),-`
`//             TYPE=&(TYP)`	`//        TYPE=&(TYP)),TO=#WORK-1`
`...`	`//END`
`//END`	`/START-EDT`
or:	`@READ '#WORK-1'`
`/START-EDT`	`...`
`@OPEN L=&(LIB-1)(E=&(ELEM-1),&(TYP))`	`@WRITE OVERWRITE`
`...`	`@HALT`
`@CLOSE`	`/START-LMS`
`@HALT`	`//OPEN-LIBRARY LIB=&(LIB-1)`
	`//ADD-ELEMENT #WORK-1,-`
	`//      TO-ELEM=(&(ELEM-1),TYPE=&(TYP))`
	`//END`

10.8 Example of an optimized procedure

The following example shows a procedure to which all optimizations mentioned in the previous sections were applied. The optimized procedure (created by the “BS2000 Performance Controlling and Modelling” team) requires only 62% of the CPU time needed by the initial version of the procedure that was written using the lower performance syntax.

Optimized procedure (recommended syntax):

```
/SET-PROCEDURE-OPTIONS
/
/OPEN-VARIABLE-CONTAINER CONTAINER-NAME=CONTFS, -
/                         FROM-FILE=*LIBRARY-ELEMENT(LIBRARY=BAD.LIB)
/DECLARE-VARIABLE VARIABLE-NAME=C-FS-L(TYPE=*STRUCTURE) -
/                , MULTIPLE-ELEMENTS=*LIST, CONTAINER=CONTFS
/
/&* The list variables and the output of show-file-attr
/ DECLARE-VARIABLE VARIABLE-NAME = ( -
/         LIST   ( TYPE = *STRING    ), -
/         TSIL   ( TYPE = *STRING    ), -
/         FS-OUT ( TYPE = *STRUCTURE ), -
/         ), MULTIPLE-ELEMENTS=*LIST
/
/&* List initialization
/ LIST = *STRING-TO-VAR ( '(ABC,DEF,GHI,JKL,MNO,ABC,DEF,GHI,JKL,MNO)' )
/
/&* List in reverse order
/ FOR I = *LIST(LIST)
/   TSIL = I, *PREFIX
/ END-FOR
/
/&* Show list variable in one command
/ SHOW-VARIABLE VARIABLE-NAME = ( LIST, TSIL )
/
/&* Search variable A inside LIST and save the result in B
/ A = 'I'
/ IND = SEARCH-LIST-INDEX('LIST', '^&A.$', PATTERN-TYPE=*REGULAR-EXPRESSION)
/ IF ( IND <> 0 )
/  B = LIST#IND
/ END-IF
/
/&* Get all files beginning with 'A'
/ EXECUTE-CMD CMD = (/SHOW-FILE-ATTRIBUTES A*) -
/            , STRUCTURE-OUTPUT = FS-OUT -
/            , TEXT-OUTPUT = *NONE
/
```

```
/&* Put contents in a container
/ C-FS-L = FS-OUT
/
/&* Calculate total size
/ TSIZE = 0
/ FOR I = *COUNTER( 1, SIZE('FS-OUT'), 1 )
/   TSIZE = TSIZE + FS-OUT#I.F-SIZE
/ END-FOR
/
/&* Save container (close implicit at procedure end )
/ SAVE-VARIABLE-CONTAINER CONTAINER-NAME = CONTFS
/
/&* Display date, time and total file size
/ WRITE-TEXT -
/   'Today &(DATE()) at &(TIME()) we have a total file size of &TSIZE -
/    on user-id &(USER-ID())..'
/
/EXIT-PROCEDURE
```

Initial procedure (less recommended syntax):

```
/SET-PROC-OPT
/
/OP-VAR-CONT CONTFS, *L(BAD.LIB)
/DECL-VAR C-FS-L,TYP=STRUCT,MULT=*L,CONT=CONTFS
/
/REMARK  This list variable
/ DECL-VAR LIST ( TYP = STRI ),MULT=*L
/
/REMARK  Its initialization
/ SET-VAR LIST ='ABC', *EXTEND; SET-VAR LIST ='DEF', *EXTEND
/ SET-VAR LIST ='GHI', *EXTEND; SET-VAR LIST ='JKL', *EXTEND
/ SET-VAR LIST ='MNO', *EXTEND; SET-VAR LIST ='ABC', *EXTEND
/ SET-VAR LIST ='DEF', *EXTEND; SET-VAR LIST ='GHI', *EXTEND
/ SET-VAR LIST ='JKL', *EXTEND; SET-VAR LIST ='MNO', *EXTEND
/
/REMARK  List in reverse order
/ DECL-VAR TSIL (TYP=STRI), MULT=*L
/
/ FOR I = *C( SIZE('LIST'), 1, -1)
/   SET-VAR TSIL#&(SIZE('LIST') - I + 1) = LIST#&I
/ END-F
/
/REMARK  Show list variable
/ SH-VAR LIST
/ SH-VAR TSIL
/
```

```
/REMARK  Search variable A inside LIST
/ SET-VAR A = 'I'
/ SET-VAR FOUND = FALSE
/ FOR I=*C(1,SIZE('LIST'),1), COND=(NOT FOUND)
/  IF ( LIST#&I == A )
/    SET-VAR FOUND = TRUE
/  EN-IF
/ END-F
/
/REMARK  Save it in B
/ IF ( FOUND )
/   SET-VAR B = LIST#&I
/ EN-IF
/
/REMARK  Get all file beginning with 'A'
/ DECL-V FS-OUT ,TYP=STRU,MULT=*L
/ EXEC-CMD (/SH-FIL-ATTR A*),STR-OUTPUT=FS-OUT,TEXT-OUT=*NONE
/
/REMARK  Calculate total size and put contents in a container
/ SET-VAR TSIZE = 0
/ FOR I=*C(1,SIZE('FS-OUT'),1)
/  SET-VAR TSIZE = TSIZE + FS-OUT#&I..F-SIZE
/  SET-VAR C-FS-L#&I = FS-OUT#&I
/ END-F
/
/REMARK  Save the container
/ SAVE-VAR-CONT CONTFS
/
/REMARK  Close the container
/ CLOSE-VAR-C CONTFS
/
/REMARK Display date, time and total file size
/ TDATE = DATE
/ TTIME = TIME
/
/W-T 'Today (&TDATE) at &TTIME we have a total file size of &TSIZE -
/     on user-id &USER-ID..'
/
/EXIT-PROC
```

11 Testing S procedures

This chapter describes the aids available to the programmer of S procedures for debugging purposes.

Firstly, SDF-P provides two commands that provide the programmer with support in debugging during the test phase; secondly, the programmer can use procedure interruptions to query the procedure's current status. The commands provided by SDF-P as debugging aids are TRACE-PROCEDURE and MODIFY-PROCEDURE-TEST-OPTIONS.

The TRACE-PROCEDURE command allows you to trace procedure execution step by step. The MODIFY-PROCEDURE-TEST-OPTIONS is used if the setting for logging the procedure is to be modified (for example, to limit logging to specific parts of the procedure that are to be tested).

The next three sections describe the possible applications for the two SDF-P commands separately. Finally, the fourth section points out the options available to the programmer in the event of a procedure interruption.

11.1 Tracing procedure execution step-by-step

The TRACE-PROCEDURE command can be used to trace the step-by-step execution of a procedure. However, this command can only be used for procedures which are specified in their procedure heads as being interruptible, more specifically in the SET-PROCEDURE-OPTIONS command (by the operand value INTERRUPT-ALLOWED = *YES).

If it is necessary to trace a procedure step-by-step, the user must first specify TRACE-PROCEDURE, before the procedure is called, i.e. before the CALL-PROCEDURE or INCLUDE-PROCEDURE. The procedure is then interrupted at intervals defined by the STEPS operand in TRACE-PROCEDURE. Procedure execution is resumed when the next TRACE-PROCEDURE command is issued.

The STEPS operand should be used to specify how many commands are to be executed before the procedure is next interrupted. The first time that TRACE-PROCEDURE is called, the default setting STEPS = 1 applies, i.e. the procedure will be interrupted after every command. If a different value is specified for STEPS, this new value will apply for the entire execution of the procedure.

When the procedure has been interrupted the programmer can, for example, query the procedure status, or use the MODIFY-PROCEDURE-TEST-OPTIONS command to change the setting for logging or to amend the maximum number of back branches (see below). After this is done, the procedure is again interrupted.

Step-by-step procedure execution can then be resumed with the TRACE-PROCEDURE command or procedure execution can be resumed with the RESUME-PROCEDURE command, causing it to be executed to the end.

If an error occurred in the procedure, the programmer can also use the interruption to modify variable contents, declare variables, etc.

Logging

After the TRACE-PROCEDURE command is entered, the execution of commands that takes place in the procedure is always logged to SYSOUT (provided that logging is allowed for the procedure), regardless of the settings of the LOGGING operand in the CALL-PROCEDURE (or INCLUDE-PROCEDURE) or MODIFY-PROCEDURE-TEST-OPTIONS command; these settings are ignored.

11.2 Modifying logging

If the settings for logging are to be modified during the test phase, the MODIFY-PROCEDURE-TEST-OPTIONS command can be called in dialog.

The following always applies: logging can be set only if logging is allowed for the procedure (in the SET-PROCEDURE-OPTIONS or MODIFY-PROCEDURE-OPTIONS command).

As with the other commands in which logging can be set, logging is controlled in the MODIFY-PROCEDURE-TEST-OPTIONS command by the LOGGING operand. The logging of commands and data can be enabled and disabled separately.

11.3 Preventing endless loops

With the BACK-BRANCH-LIMIT operand, the MODIFY-PROCEDURE-TEST-OPTIONS command provides an option for preventing endless loops.

BACK-BRANCH-LIMIT sets an upper limit on the number of back branches within a procedure. Back branches are both branches from the end of a loop to its beginning (i.e. from END-WHILE to WHILE, from END-FOR to FOR, from UNTIL to REPEAT) and branches executed by means of the GOTO command. Back branches using the SKIP-COMMANDS command are not included.

11.4 Procedure interruption

If procedure interruption is allowed (INTERRUPT-ALLOWED = *YES in the SET-PROCEDURE-OPTIONS or MODIFY-PROCEDURE-OPTIONS command), procedure execution can be interrupted with the K2 key.

It is then possible to check the procedure status, modify the procedure environment etc., and resume procedure execution with the RESUME-PROCEDURE command or resume step-by-step execution with the TRACE-PROCEDURE command.

For example, the procedure status can be checked by means of the predefined functions. Thus, the PROC-LEVEL( ) function can be used to check the nesting level; the SYSFILE environment can be checked with the functions SYSCMD( ), SYSDTA( ) etc.; system files can then be rerouted with the ASSIGN-SYSDTA commands, and so on.

Variable contents can be output with the SHOW-VARIABLE command and structure layouts can be output with the SHOW-STRUCTURE-LAYOUT command.

You can query whether variables are declared using the IS-DECLARED( ) function and whether a declared variable already contains a value with the IS-INITIALIZED( ) function.

If the variables do not have the contents that are currently needed for the correct execution of the procedure, the programmer can assign them the correct contents by means of the SET-VARIABLE command.

When elements of complex variables are accessed, you can check whether the element names are correct, whether the elements are present, etc. The predefined functions FIRST-VARIABLE-NAME( ) and NEXT-VARIABLE-NAME, among others, are provided for this purpose.
These are only a few indications of how the programmer can query and amend the procedure environment interactively during the test phase. While the procedure is interrupted, not only SDF-P functions and commands can be used, but also BS2000 commands. For a detailed description of the SDF-P functions, see chapter “Predefined functions” on page 347ff; for SDF-P commands, see chapter “SDF-P commands” on page 541ff. The BS2000 commands are described in “Commands, Vol. 1-5” [3].

11.5 Simulating the runtime environment

All the functionality of SDF-P is implemented in the two subsystems SDF-P and SDFPBASY (release unit SDF-P-BASYS). Full functionality is only available with the SDF-P subsystem to be purchased separately (see also the section “Brief product description” on page 18).

If a procedure is created in a system in which SDF-P is loaded, then the full SDF-P functionality is available. If this procedure is also to be used in a system in which only the SDF-P-BASYS functionality is available, then the ability to execute the procedure must also be checked for this environment. This runtime environment can be simulated in the calling task without unloading the SDF-P subsystem using the following command:

```
/MODIFY-PROCEDURE-TEST-OPTIONS FUNCTIONALITY=*BASIC
```

After that, only the SDF-P functionality is available to the caller. The only exception to this is the MODIFY-PROCEDURE-TEST-OPTIONS command. It is also executed in simulation mode so that the user can return to full SDF-P functionality:

```
/MODIFY-PROCEDURE-TEST-OPTIONS FUNCTIONALITY=*FULL
```

12 Converting non-S procedures

Non-S procedures cannot use the capabilities of SDF-P unless they are converted.

The conversion of non-S procedures to S procedures can be carried out in steps. Steps 1 to 6 are used only in order for the (former) non-S procedures to run as S procedures.

With SDF-CONV, automatic conversion of non-S procedures to S procedures is possible (PROCEDURE-FORMAT operand). Options which can be specified for this procedure are: whether the input procedure's command language should be retained, or should be converted into the SDF command language; whether data lines are to be converted to statement lines; and whether the output from SDF commands in the converted procedure may be in blocked form. (For further details see the "SDF-CONV" manual [17].)

12.1 Foreground non-S procedures

The following list applies to procedures which are called in the foreground, i.e. to procedures which are to run in interactive mode or are called by other procedures.

1. Procedure head and end of procedure
   a) Generate SDF-P procedure head:
      Remove the BEGIN-PROCEDURE or PROCEDURE command.
      The procedure now has a procedure head implicitly. The default settings of the SET-PROCEDURE-OPTIONS command apply to the procedure attributes. Procedure parameters cannot be transferred.
   b) Terminate procedure correctly:
      Remove the END-PROCEDURE or ENDP command.
      Insert the EXIT-PROCEDURE command.

2. If necessary: declare procedure parameters:

   Generate a DECLARE-PARAMETER block (or call the DECLARE-PARAMETER command).
   If procedure parameters are to be transferred to the S procedure, they must be declared in the procedure head.
   In doing this, each parameter should be declared separately by calling the DECLARE-PARAMETER command. These command calls must then be incorporated into a DECLARE-PARAMETER block which is initiated with the BEGIN-PARAMETER-DECLARATION command and terminated with the END-PARAMETER-DECLARATION command.

3. Initialize procedure parameters:

   Provide a value for the INITIAL-VALUE operand in the DECLARE-PARAMETER command.
   If a procedure parameter is declared with the default value INITIAL-VALUE = *NONE, it must be assigned a value in the procedure call. Otherwise, an error message is output.
   If the procedure parameter is declared with INITIAL-VALUE = *PROMPT, the user is prompted for the value in the dialog, after the procedure call. If prompting is not possible, the procedure parameter is implicitly assigned an empty character string.
   If the procedure parameter is assigned a value with INITIAL-VALUE, it does not have to be assigned a value in or after the procedure call. The defined initial value (INITIAL-VALUE) is then used as the default.

4. If necessary, set job variable replacement:

   Unless otherwise specified, there is no job variable replacement in S procedures. This is possible only if the JV-REPLACEMENTS operand is set to the value AFTER-BUILTIN-FUNCTION, using the SET-PROCEDURE-OPTIONS command (or if this is set later on in the MODIFY-PROCEDURE-OPTIONS command in the procedure body). However, we recommend that the default be left unchanged; instead, &(jobvar) should be replaced with &(JV('jobvar')).

5. If necessary, set procedure attributes:

   Using the SET-PROCEDURE-OPTIONS command in the procedure head, define the procedure attributes which are to be different from the default settings. This applies, for example, to the escape character in data records or behavior when errors occur in data records. The escape character is defined with the DATA-ESCAPE-CHARACTER operand, while the behavior in the event of errors is defined with the DATA-ERROR-HANDLING operand.

6. Convert procedure call:

   a) CALL command:
   Replace with CALL-PROCEDURE or INCLUDE-PROCEDURE. The ISP command CALL is compatible with SDF-P; internally, it is mapped to the extended CALL-PROCEDURE command. As a result, the default settings for the CALL-PROCEDURE command apply to procedure calls with CALL. Nevertheless, procedure calls with CALL should be replaced with CALL-PROCEDURE or INCLUDE-PROCEDURE.

   b) Adapt CALL-PROCEDURE command:
   The CALL-PROCEDURE command was enhanced for SDF-P. If this enhancement is not taken into account in the command call, the appropriate default settings are used. If other settings are to be used, the corresponding operands must be included in the command call.

   c) Convert procedure call with DO command to CALL-PROCEDURE or INCLUDE-PROCEDURE:
   Procedure calls with the DO command continue to be supported. However, since DO does not support true call nesting, procedures should be called only with the command CALL-PROCEDURE or INCLUDE-PROCEDURE.
   When converting the procedure call from DO to CALL- or INCLUDE-PROCEDURE, note that termination behavior is different.

7. Create valid branch tags:

   a) Replace non-S tags (format: .tag) with S tags (tag:)
   SDF-P supports tags in non-S format only on the top block level, but not in nested blocks. Non-S tags can be addressed in branches only with the SKIP-COMMANDS command (SKIP, SKIPJV, SKIPUS) and in commands used for conditional job control (MODIFY-JV-CONDITIONALLY, WAIT-EVENT, ADD-CJC-ACTION, WAIT, ON).

   b) Replace SKIP-COMMANDS command (SKIP, SKIPJV, SKIPUS) with control flow commands.
   If unconditional branches must be carried out with SKIP-COMMANDS, SKIP-COMMANDS can be replaced with the GOTO command: branches with SKIP-COMMANDS can be carried out only on nesting level 0, i.e. not in command blocks. Branches within command blocks or to superordinate command blocks are carried out with GOTO.
   If conditional branches are carried out with SKIP-COMMANDS, SKIP-COMMANDS can be replaced with an IF block (possibly IF-BLOCK-ERROR block or IF-CMD-ERROR block as well).

8. Insert error handling

   Replace SET-JOB-STEP commands (STEP) with IF-BLOCK-ERROR or IF-CMD-ERROR.
   In S procedures, error handling is carried out in error handling blocks which are initiated with the IF-BLOCK-ERROR or IF-CMD-ERROR command. These error handling blocks cannot be preceded by a SET-JOB-STEP or STEP command, since SET-JOB-STEP and STEP eliminate the error situation. In this case, IF-BLOCK-ERROR or IF-CMD-ERROR can no longer detect the original error situation.
   Note that IF-BLOCK-ERROR does not reset any switches!

9. Delete equals sign as first character in command call

   In some commands, the command names can be followed by any character: If this first character after the command name is an equals sign, SDF-P interprets the command line as a value assignment for a variable whose variable name is the command name.

10. Replace job switch with variables:

    The execution of S procedures should not be controlled via job switches. For the purpose of job control, SDF-P provides control flow commands which can be used to program branches and loops. Variables are used in requesting the appropriate conditions.

11. If necessary: mark statements:

    Procedure lines which contain statements must begin with two slashes.

12.2 Enter job

The following list applies to procedures (Enter files) which are called with the ENTER-JOB command, i.e. to procedures which are to run in batch mode.

1. Procedure head and end of procedure

   a) Generate SDF-P procedure head:
   Remove the SET-LOGON-PARAMETERS or LOGON command.
   In S procedures, no distinction is made between foreground and background procedures. For this reason, S procedures cannot contain a SET-LOGON-PARAMETERS or LOGON command.
   If the SET-LOGON-PARAMETERS or LOGON command is removed, the procedure has a procedure head implicitly. The default settings apply to the procedure attributes.

   b) Terminate procedure correctly:
   Remove the EXIT-JOB (with MODE=*NORMAL) or LOGOFF command.
   Insert the EXIT-PROCEDURE command.
   Since no distinction is made between background and foreground procedures in S procedures, the EXIT-JOB or LOGOFF command must also be removed at the end of the procedure.

   c) Terminate procedure abnormally:
   In an enter job, terminating a procedure with /EXIT-PROCEDURE ERROR=*YES in a monitoring MONJV nevertheless leads to the end status $T (“normal end”). If the enter file used to date contains the EXIT-JOB command with MODE=*ABNORMAL or ABEND, a distinction must be made between the following two cases:
   – When the job is not to be monitored by means of a MONJV or if the MONJV end status $T is required, these commands can also be replaced by /EXIT-PROCEDURE ERROR=*YES.
   – When the job is to be monitored by a MONJV and the MONJV end status $A is reached, the EXIT-JOB command with MODE=*ABNORMAL must be used.

   Note
   It must be ensured that the S. files are not seleted when a background procedure is terminated with /EXIT-JOB.

2. If a background procedure is called from another procedure:

   Adapt the procedure call, or replace the ENTER-JOB command with ENTER-PROCEDURE.
   (Procedures which are to run as background procedures under SDF-P are called with the ENTER-PROCEDURE command, which, in turn, issues an ENTER-JOB command internally.)

3. For the remaining conversion steps, see the description of converting Non-S proce- dures.

12.3 Compatibility of commands for procedure control

This section contains a description of the restrictions which apply to non-SDF-P commands in S procedures as well as the restrictions for SDF-P commands during procedure execution.

(Branch) tags and AID sequences

(Branch) tags

Tags in the format .tag can be used only in connection with SKIP commands and the commands used for conditional job control (see below).

Tags in the format tag: can be used only with SDF-P commands.

AID sequences

AID sequences are sequences of commands to AID which are separated by semicolons. SDF-P control flow commands cannot be used in these sequences.
AID commands which are followed by a command or subcommand list cannot be created by means of variable replacement.

Non-SDF-P commands

ADD-CJC-ACTION (or ON)
Only ENTER-JOB, ENTER-PROCEDURE and MODIFY-JV commands are permitted in ADD-CJC-ACTION blocks.

BEGIN-PROCEDURE (or PROCEDURE)
Not supported.

CALL-PROCEDURE (or CALL)
A semicolon not enclosed in brackets and not in a comment is interpreted as a command separator.

CANCEL-PROCEDURE
If the CANCEL-PROCEDURE command is used in order to terminate an error situation, is must be preceded by the following commands:

```
IF-BLOCK-ERROR
END-IF
```

DO (ISP command)
Should not be used when calling procedures in which procedure parameters are declared with TRANSFER-TYPE = *BY-REFERENCE.
A semicolon not enclosed in brackets is interpreted as a command separator.

EXIT-JOB (or ABEND)
If the EXIT-JOB command is used to terminate the job in the case of an error, it must be enclosed within the following commands:

```
IF-BLOCK-ERROR
END-IF
```

INTR (ISP command)
A semicolon not enclosed in brackets is interpreted as a command separator.

LOGOFF
If the LOGOFF command is used to terminate a job in an error situation, it must be enclosed by the following commands:

```
IF-BLOCK-ERROR
END-IF
```

LOGON
Not supported.

PAUSE (ISP command)
The equals sign cannot be the first significant character. Otherwise, the command is interpreted as a value assignment (as SET-VARIABLE without a command name).
A semicolon not enclosed in brackets is interpreted as a command separator.

REMARK
As usual for SDF commands, it must be noted that
– The equals sign cannot be the first significant character. Otherwise, the command is interpreted as a value assignment (as SET-VARIABLE without a command name).
– A semicolon not enclosed in brackets is interpreted as a command separator.
– Parentheses, single and double quotation marks, must not appear singly but only in matching pairs.

SET-JOB-STEP(STEP)
Cannot be created by means of variable replacement.

SET-LOGON-PARAMETERS
Is not supported.

SKIP-COMMANDS (ISP commands SKIP, SKIPJV, SKIPUS)
Can branch only to nesting level 0, i.e. only to the top block level.

TYPE (ISP command)
The equals sign cannot be the first significant character. Otherwise, the command is interpreted as a value assignment (as SET-VARIABLE without a command name).
A semicolon not enclosed in brackets is interpreted as a command separator.

WAIT-EVENT (and WAIT), MODIFY-JV-CONDITIONALLY
Since this permits only branch commands with non-S tags, the WAIT-EVENT command should not be used within blocks.

User-defined commands

You can use SDF to define your own commands. Such commands can be defined with the operand value “command-rest”, in which semantic semicolons are permitted.

If such commands are used in S procedures, it should be noted that - as usual in normal SDF commands:

– The equals sign cannot be the first significant character. Otherwise, the command is interpreted as a value assignment (as SET-VARIABLE without a command name).

– A semicolon not enclosed in brackets is interpreted as a command separator.

– Parentheses, single and double quotation marks, must not appear singly but only in matching pairs.

SDF-P commands

EXIT-PROCEDURE
If the EXIT-PROCEDURE command is used to terminate the job in an error situation, it must be enclosed in the following commands:

```
IF-BLOCK-ERROR
END-IF
```

MODIFY-PROCEDURE-TEST-OPTIONS
Has no effect on non-S procedures.

Restrictions for records

If the S procedure is an ISAM file, the records in this file are passed to the reading program without their ISAM keys.

12.4 Conversion examples

Example 1: Reorganizing storage space

This procedure reorganizes the storage space for all files of a user ID. If the file is a PLAM library, all its elements are copied to a new library.

The user ID must be entered with a leading $ sign and a trailing period.
There are no default values.

a) Non-S procedure

```
/BEGIN-PROCEDURE LOGGING=N,    -
/               PARAMETERS=YES(   -
/                 PROCEDURE-PARAMETERS=(  -
/                    &USERID=),   -
/                 ESCAPE-CHARACTER='&')
/REMARK +----------------------------------------------------------------+"
/REMARK |                                                                |
/REMARK | This procedure compacts all files contained on a user-id.      |
/REMARK | If the file is a PLAM library then all elements are            |
/REMARK | duplicated in a new library.                                   |
/REMARK | The user-id must be given with the leading dollar sign         |
/REMARK | and with the trailing point.                                   |
/REMARK | There is no default value.                                     |
/REMARK |                                                                |
/REMARK +----------------------------------------------------------------+"
/REMARK &USERID
/ASSIGN-SYSOUT TO=*DUMMY
/SHOW-FILE-ATTR &USERID,INFO=NAME-AND-SPACE,   -
/              OUTPUT=#LST(FORM-NAME=FILE-NAME)
/ASSIGN-SYSOUT TO=*PRIMARY
/ASSIGN-SYSDTA TO=*SYSCMD
/MOD-JOB-SWITCHES ON=(1,4,5)
/START-EXECUTABLE-PROGRAM $EDT
@@READ '#LST'
@@COL1ON&INSERT'/CALL-PROC #PROC2,P-P=('
@@SUFFIX&WITH')'
@@RENUM
@@CR0.001W'/BEGIN-PROCEDURE LOGGING=N'
@@CR0.002W'/ASSIGN-SYSOUT TO=#OUTBEFORE'
@@CR0.003W'/SHOW-FILE-ATT &USERID,INFO=SPACE-SUMMARY'
@@CR0.004W'/ASSIGN-SYSOUT TO=*PRIMARY'
@@CR$+.01W'/ASSIGN-SYSOUT TO=#OUTAFTER'
@@CR$+.01W'/SHOW-FILE-ATT &USERID,INFO=SPACE-SUMMARY'
@@CR$+.01W'/ASSIGN-SYSOUT TO=*PRIMARY'
@@CR$+.01W'/END-PROCEDURE'
```

```
@@WR'#PROC1' OVER
@@DELETE
@@CR$+.01W'/BEGIN-PROC LOGGING=N,  -'
@@CR$+.01W'/          PARAMETERS=YES(  -'
@@CR$+.01W'/              PROC-PARAM=(&FILE),-'
@@CR$+.01W'/              ESCAPE-CHAR=''&'')'
@@CR$+.01W'/COPY-FILE &FILE,&FILE..WORK,PROTECTION=SAME'
@@CR$+.01W'/SKIP-COMMANDS TO-LABEL=ERASE'
@@CR$+.01W'/SET-JOB-STEP'
@@CR$+.01W'/SKIP-COMMANDS TO-LABEL=END'
@@CR$+.01W'/.ERASE DELETE-FILE &FILE,OPTION=DATA'
@@CR$+.01W'/SKIP-COMMANDS TO-LABEL=FILE'
@@CR$+.01W'/SET-JOB-STEP'
@@CR$+.01W'/DELETE-FILE &FILE..WORK'
@@CR$+.01W'/SET-JOB-STEP'
@@CR$+.01W'/SKIP-COMMANDS TO-LABEL=END'
@@CR$+.01W'/.FILE SET-JOB-STEP'
@@CR$+.01W'/FILE &FILE,SPACE=(100,20),FCBTYPE=PAM'
@@CR$+.01W'/SKIP-COMMANDS TO-LABEL=LMS'
@@CR$+.01W'/SET-JOB-STEP'
@@CR$+.01W'/COPY-FILE &FILE..WORK,&FILE'
@@CR$+.01W'/DELETE-FILE &FILE..WORK,IGNORE-PROTECTION=ACCESS'
@@CR$+.01W'/SET-JOB-STEP'
@@CR$+.01W'/SKIP-COMMANDS TO-LABEL=END'
@@CR$+.01W'/.LMS SET-JOB-STEP'
@@CR$+.01W'/ASSIGN-SYSDTA TO=*SYSCMD'
@@CR$+.01W'/MOD-JOB-SWITCHES ON=(1,2,3,4,5)'
@@CR$+.01W'/START-EXECUTABLE-PROGRAM $LMS,MONJV=#BIDON'
@@CR$+.01W'LIB &FILE..WORK,IN'
@@CR$+.01W'LIB &FILE,OUT,NEW'
@@CR$+.01W'DUP* */*'
@@CR$+.01W'END'
@@CR$+.01W'/MOD-JOB-SWITCHES OFF=(1,2,3,4,5)'
@@CR$+.01W'/SET-JOB-STEP'
@@CR$+.01W'/SKIP-COMMANDS IF=JV(((#BIDON,4,4) = ''0000'')),-'
@@CR$+.01W'/              TO-LABEL=AVEND'
@@CR$+.01W'/SET-JOB-STEP'
@@CR$+.01W'/COPY-FILE &FILE..WORK,&FILE'
@@CR$+.01W'/DELETE-FILE &FILE..WORK'
@@CR$+.01W'/SET-JOB-STEP'
@@CR$+.01W'/SKIP-COMMANDS TO-LABEL=END'
@@CR$+.01W'/.AVEND SET-JOB-STEP'
@@CR$+.01W'/DELETE-FILE &FILE..WORK'
@@CR$+.01W'/.END SET-JOB-STEP'
@@CR$+.01W'/MOD-FILE-ATT &FILE,SUPPORT=ANY-DISK(RELEASE(9999))
@@CR$+.01W'/W-TEXT ''File &FILE compacted'''
@@CR$+.01W'/SET-JOB-STEP'
@@CR$+.01W'/END-PROCEDURE'
```

```
@@WR'#PROC2'OVER
@@H
/CALL-PROCEDURE #PROC1
/START-EXECUTABLE-PROGRAM $EDT
@@REA'#OUTBEFORE'
@@REA'#OUTAFTER'
@@SET #I5 = SUBSTR 1:38-42:
@@SET #I6 = SUBSTR 2:38-42:
@@SET #I7 = #I5 - #I6
@@DELETE
@@SET #S5 = CHAR #I5
@@SET #S6 = CHAR #I6
@@SET #S7 = CHAR #I7
@@CR #S1:'Space before : ',#S5
@@CR #S2:'Space after  : ',#S6
@@CR #S3:'              ------------'
@@CR #S4:'Space won    : ',#S7
@@PRINT #S1 N
@@PRINT #S2 N
@@PRINT #S3 N
@@PRINT #S4 N
@@H
/MOD-JOB-SWITCHES OFF=(1,4,5)
/ASSIGN-SYSDTA TO=*SYSCMD
/END-PROC
```

b) S procedure

```
/&* +-------------------------------------------------------------+
/&* |                                                             |
/&* | This procedure compacts all files contained on a user-id.   |
/&* | If the file is a PLAM library then all elements are         |
/&* | duplicated in a new library.                                |
/&* | The user-id is given in parameters with or without the      |
/&* | leading dollar sign and with or without the trailing point. |
/&* | If no parameter is given then the current user-id is taken. |
/&* |                                                             |
/&* +-------------------------------------------------------------+
/DECLARE-PARAMETER USER-ID(INITIAL-VALUE=USER-ID,TRANSFER-TYPE=BY-VALUE)
/IF (SUBSTR(USER-ID,1,1)<>'$')
/   "THEN" USER-ID = '$' // USER-ID
/END-IF
/IF (SUBSTR(USER-ID,LENGTH(USER-ID),1)<>'.')
/   "THEN" USER-ID = USER-ID // '.'
/END-IF
/DECLARE-VARIABLE FS(TYPE=STRUCT(*DYNAMIC)),MULT-ELEM=LIST
/DECLARE-VARIABLE VARLOOP(TYPE=STRUCT(*DYNAMIC))
```

```
/EXEC-CMD CMD=(SHOW-FILE-ATTR &USER-ID,INFO=NAME-AND-SPACE),-
/STRUCTURE-OUTPUT=FS,TEXT-OUTPUT=*NONE
/DECLARE-VARIABLE SPACEBEFORE(INIT=0,TYP=*INTEGER)
/FOR VARLOOP = *LIST(FS)
/    SPACEBEFORE = SPACEBEFORE + VARLOOP.F-SIZE
/    IF (IS-LIBRARY(VARLOOP.F-NAME))
/       "THEN" LIBBLOCK: BEGIN-BLOCK DATA-INSERTION=YES
/             COPY-FILE FROM-FILE=&(VARLOOP.F-NAME),-
/                       TO-FILE=&(VARLOOP.F-NAME).WORK,PROTECTION=SAME
/             IF-BLOCK-ERROR
/                   EXIT-BLOCK LIBBLOCK
/             END-IF
/             MODIFY-JOB-SWITCHES ON=(1,4)
/             ASSIGN-SYSDTA TO=*SYSCMD
/             DELETE-FILE &(VARLOOP.F-NAME)
/             IF-BLOCK-ERROR
/               DELETE-FILE &(VARLOOP.F-NAME).WORK,OPTION=DATA
/               EXIT-BLOCK LIBBLOCK
/             END-IF
/             START-EXE FROM-FILE=$LMS
/             SEND-DATA 'LIB &(VARLOOP.F-NAME).WORK,IN'
/             SEND-DATA 'LIB &(VARLOOP.F-NAME).NEW.OUT'
/               SEND-DATA 'DUP* */*'
/               SEND-DATA 'END'
/             ASSIGN-SYSDTA TO=*PRIMARY
/             MODIFY-JOB-SWITCHES OFF=(1,4)
/             IF-BLOCK-ERROR
/                COPY-FILE &(VARLOOP.F-NAME).WORK,-
/                          &(VARLOOP.F-NAME)
/                ELSE
/               WR-TEXT 'Library &(VARLOOP.F-NAME) compacted'
/             END-IF
/             DELETE-FILE &(VARLOOP.F-NAME).WORK,IGNORE-PROTECTION=ACCESS
/             MOD-FILE-ATTR &(VARLOOP.F-NAME),-
/                            SUPPORT=ANY-DISK(SPACE=RELEASE(10000))
/              END-BLOCK
/         ELSE
/              MOD-FILE-ATTR &(VARLOOP.F-NAME),-
/                             SUPPORT=ANY-DISK(SPACE=RELEASE(10000))
/              IF-BLOCK-ERROR
/                  ELSE
/                  WR-TEXT 'File &(VARLOOP.F-NAME) compacted'
/              END-IF
/    END-IF
/END-FOR
/IF-BLOCK-ERROR
/END-IF
/DECLARE-VARIABLE FS2(TYP=STRING),MULT-EL=LIST
```

```
/EXEC-CMD CMD=(SHOW-FILE-ATTR &USER-ID,INFO=SPACE-SUMMARY),-
/         TEXT-OUTPUT=FS2
/SPACEAFTER = INTEGER (SUBSTR (FS2#,37,5))
/WR-TEXT 'Space before : &SPACEBEFORE'
/WR-TEXT 'Space after  : &SPACEAFTER'
/WIN = SPACEBEFORE - SPACEAFTER
/WR-TEXT '               -----'
/WR-TEXT 'Spaced won : &WIN'
/EXIT-PROC
```

Example 2: Copying files with the aid of wildcards

In the following procedure, files are first selected using wildcards. Then, they are copied under a new catalog ID, a new user ID or a new prefix.

For clarification:
`CALL-PROC copy, PROC-PAR = (SOURCE=xxx, TARGET=yyy)`

where xxx could be:
`:CAT: ,$USERID., *STRING*` or anything which is accepted by SHOW-FILE-ATTR.

where yyy could be:
`:CAT: ,$USERID2.` or any prefix with a trailing period.

a) Non-S procedure

```
/BEGIN-PROCEDURE LOGGING=N,                             -
/               PARAMETERS=YES(                         -
/                 PROCEDURE-PARAMETERS=(                -
/                      &SOURCE=,                        -
/                      &TARGET=),                       -
/                 ESCAPE-CHARACTER='&')
/REMARK +----------------------------------------------------------------+
/REMARK |                                                                |
/REMARK | This procedure enables the user to copy some files selected   |
/REMARK | with wildcards under a new cat-id, a new user-id or a new     |
/REMARK | prefix.                                                        |
/REMARK | Examples :                                                     |
/REMARK |   CALL-PROC copy,PROC-PAR=(SOURCE=xxx,TARGET=yyy)              |
/REMARK |   where xxx could be :CAT: , $USERID. , *STRING* or            |
/REMARK |                      everything accepted by SHOW-FILE-ATT     |
/REMARK |         yyy could be :CAT2: , $USERID2. or any PREFIX with    |
/REMARK |                      a trailing point                         |
/REMARK |                                                                |
/REMARK +----------------------------------------------------------------+
/REMARK &SOURCE
/REMARK &TARGET
/ASSIGN-SYSOUT TO=*DUMMY
/ASSIGN-SYSLST TO=#LST
/SHOW-FILE-ATTR &SOURCE,LIST=((SYSLST),FILENAM)
/ASSIGN-SYSLST TO=*PRIMARY
/ASSIGN-SYSOUT TO=*PRIMARY
/ASSIGN-SYSDTA TO=*SYSCMD
/MODIFY-JOB-SWITCHES ON=(1,4,5)
/START-EXECUTABLE-PROGRAM $EDT
@@PROC 1
@@REA '#LST'
@@PROC 2
@@COPY &(1)
```

```
@@ON&FIND '.',1 DELETE PREFIX
@@DELETE&:1-1
@@PREFIX & WITH '/            TO-FILE=&TARGET'
@@RENUM 1.5(1)
@@END
@@SUFFIX & WITH '.-'
@@PREFIX & WITH '/COPY-FILE FROM-FILE  ='
@@COPY &(2)
@@RENUM
@@CR 0.01W'/BEGIN-PROCEDURE LOGGING=N'
@@CR$+.01W'/END-PROCEDURE'
@@WR'#LST' OVER
@@H
/MODIFY-JOB-SWITCHES OFF=(1,4,5)
/ASSIGN-SYSDTA TO=*PRIMARY
/CALL-PROC #LST
/END-PROCEDURE
```

b) S procedure

```
/BEGIN-PARAMETER-DECLARATION
/  DECLARE-PARAMETER SOURCE(INIT=*PROMPT, TYPE=*STRING)
/  DECLARE-PAREMETER TARGET(INIT=*PROMPT, TYPE=*STRING)
/END-PARAMETER-DECLARATION
/&*+-----------------------------------------------------------------+
/&*|                                                                 |
/&*| This procedure enables the user to copy some files selected with |
/&*| wildcards under a new cat-id, a new user-id, or a new prefix.    |
/&*| Examples :                                                       |
/&*|  CALL-PROC copy,PROC-PAR=(SOURCE=xxx,TARGET=yyy)                 |
/&*|  where xxx could be :CAT: , &USERID. , *STRING* or everything    |
/&*|                         accepted by SHOW-FILE-ATTRIBUTES         |
/&*|         yyy could be :CAT2: , &USERID. or any PREFIX with a      |
/&*|                         trailing point.                          |
/&*|                                                                  |
/&*+-----------------------------------------------------------------+
/DECLARE-VARIABLE FS(TYPE=STRUCT(*DYNAMIC)),MULT-ELEM=LIST
/DECLARE-VARIABLE VARLOOP(TYPE=STRUCT(*DYNAMIC))
/DECLARE-VARIABLE TARGET2(TYPE=STRING)
/EXEC-CMD CMD=(SHOW-FILE-ATTR &SOURCE,-
/INFO=NAME-AND-SPACE),-
/STRUCTURE-OUTPUT=FS,TEXT-OUT=*NONE
/FOR VARLOOP = *LIST(FS)
/    TARGET2 = TARGET // -
/              VARLOOP.SHORT-F-NAME
/    COPY-FILE FROM = &(VARLOOP.F-NAME),-
/               TO  = &(TARGET2)
/    IF-BLOCK-ERROR
/      "THEN" WR-TEXT 'File &(VARLOOP.F-NAME) not copied'
/       ELSE
/             WR-TEXT 'File &(VARLOOP.F-NAME) copied on &TARGET2'
/    END-IF
/END-FOR
/EXIT-PROC
```

13 Program interfaces

This chapter describes the program interfaces for systems support and for the non-privileged application programmer.

13.1 Program interfaces for systems support

The sections which follow present details of the program interfaces with which systems support can write user-specific functions, and the exit routines which can be used.

13.1.1 Assembler macros for creating user-written functions

For creating user-specific functions, systems support is provided both with assembler macros and with C macros. Both of these are described below.

BIFDEF

The BIFDEF macro generates the table entries which link the names of functions with the addresses of the entries for their executable modules and syntax specifications.

For further details refer to section “System administration functions” on page 242.

Operation	Operands
BIFDEF	MF = L
	NAME = <name 1..20>
	SYNTAX = <name 1..8>
	CODE = <name 1..8>

Operands

MF = L
LIST format of the macro call.

NAME = <name 1..20>
Name of the system administration function (first letter should be an “X”).

SYNTAX = <name 1..8>
Name of the reference point for the syntax specification (already specified in the BIFDESC macro).

CODE = <name 1..8>
Name of the executable module entry for the function (written by systems support).

BIFDESC

The BIFDESC macro contains the syntax specification for system administration functions.

This macro defines a static structure which will be used exclusively by the SDF-P-BIF subsystem.

For further details, see section “System administration functions” on page 242.

Operation	Operands
BIFDESC	NAME = <name 1..20>
	,ENTRYN = <name 1..8> / (*CSECT,<name 1..8>)
	,PARLIST = *NONE / list-poss(2000): (parameter-specification)
	,PARFORM = *BY-VALUE / *STRING
	,VALTYPE = *STRING / *INTEGER / *BOOLEAN / *ANY

Operands

Name = <string 1..20>
Name of the system administration function (first letter should be an “X”).

ENTRYN
Name of the reference point for the syntax specification.

= <name 1..8>
Identifies the reference point for the syntax specification. In this case no CSECT is generated.

= (*CSECT , <name 1..8>
Specifies the reference point for the syntax specification and then initiates the generation of the CSECT.

PARLIST
Specifies a list of operands.

=*NONE
There are no operands.

= list-poss(2000): (PARAMETER-SPECIFICATION)
Provides the specifications for lists of operands. Parentheses must be used even if there is only one specification.

parameter-specification = parameter-name,parameter-type[,default-value [,keyword-list]]
Specifies the names of operands, the operand types and optional default values and keywords.

parameter-name = <name 1..20>
Name of the operand.

parameter-type = *STRING / *INTEGER / *BOOLEAN / *ANY / *KEYWORD
Type of the operand.

default-value = <integer -2^{31}..2^{31}-1> / <c-string 0..4096> / TRUE / FALSE / ON / OFF / YES / NO / *<name 1..30>
Default value to be used if the user does not specify the operand value. If this starts with an asterisk (*), it is of the type KEYWORD. If it is enclosed within quotation marks, it is of the type STRING.

keyword-list = list-poss(2000): (keyword)
List of acceptable keywords, which must be enclosed within parentheses even if only one keyword is specified.

keyword = *<name 1..30>
Name of the keyword.

PARFORM
Format of the operand

=*BY-VALUE
The operand is a value.

= *STRING
The operand is a string.

VALTYPE = *STRING / *INTEGER / *BOOLEAN / *ANY
Type of the return value.

BIFMDL1

The BIFMDL1 macro call generates the DSECT for the values of the system administration function, or contains the specification of the structure of each element in the operand list for the executable module.

For further details, see section “System administration functions” on page 242.

Operation	Operands
BIFMDL1	MF = D ,PREFIX = <u>B</u> / prefix ,MACID = <u>IF1</u> / macid

Operands

MF = D
DSECT format of the macro call: creates a DSECT for the operand list.

PREFIX = <u>B</u> / prefix
Defines the first character of the generated name. Default: B.

MACID = <u>IF1</u> / macid
A string, up to three characters long, which replaces characters 2 to 4 of the generated name. Default: IF1.

DSECT

```
         BIFMDL1 MF=D
BIF1     DSECT
               *,##### PREFIX=B, MACID=IF1 #####
BIF1VLG  DS    F            VALUE LENGTH
BIF1VPT  DS    A            VALUE POINTER
BIF1VTY  DS    X            VALUE TYPE
BIF1STR  EQU   X'01'        -- VALUE_STRING
BIF1INT  EQU   X'02'        -- VALUE_INTEGER
BIF1BOOL EQU   X'03'        -- VALUE_BOOLEAN
BIF1KEYW EQU   X'04'        -- VALUE_KEYWORD
BIF1RES1 DS    XL1          RESERVED
BIF1RES2 DS    XL1          RESERVED
BIF1RES3 DS    XL1          RESERVED
BIF1#    EQU   *-BIF1VLG    LENGTH
```

This DSECT can be applied to any element in the operand list. It is used to describe operands and return codes.

The field to which BIF1VPT (the value pointer) points is:

- for string values, the string itself
- for integer values:
  - if PARFORM = *BY-VALUE: a “fullword” which represents an integer value
  - if PARFORM = *STRING: a string which contains the EBCDIC representation of an integer value (from 1 to 11 characters)
- for Boolean values:
  - if PARFORM = *BY-VALUE: X´00´ for FALSE, or X´01´ for TRUE
  - if PARFORM = *STRING: a string value, ´FALSE´ or ´TRUE´ (four or five characters)
- for keyword values: a string value with a leading asterisk.

Note

The format of the return code values depends on the PARFORM operand.

BIFMDL2

The BIFMDL2 macro call generates the DSECT for the return code from the system administration function, or contains the specification of the return code which is supplied by the executable module.

For further details, see section “System administration functions” on page 242.

Operation	Operands
BIFMDL2	MF = D ,PREFIX = B / prefix ,MACID = IF2 / macid

Operands

MF = D
DSECT format of the macro call: creates a DSECT for the operand list.

PREFIX = B / prefix
Defines the first character of the generated name. Default: B.

MACID = IF2 / macid
A string, up to three characters long, which replaces characters 2 to 4 of the generated name. Default: IF2.

DSECT

```
BIF2D       DSECT
BIF2SC2     DS       X            Subcode2
BIF2SC1     DS       X            Subcode1
BIF2MID     DS       CL7          Msg-id
BIF2#       EQU      *-BIF2SC2
```

13.1.2 Exit routines

Systems support must take into account the behavior of SDF-P:

- when assigning register 12 in TP mode: register 12 must contain the address of the program manager.
- when using system exits 080 and 081 (SYSCMD exits).

With regard to SYSCMD exits 080 and 081, SDF-P also affects the conversion of SDF commands and the time at which the SYSCMD exit is activated. The SYSCMD exits are described in detail in the manual entitled “System Exits” [24].

Command conversion in SYSCMD exit

SDF-P commands cannot be modified in SYSCMD exit routines; changes to SDF-P commands produce errors or are not recognized.

If an SDF command call following the IF-CMD-ERROR command call in a procedure is divided into several commands in an exit routine (1:n conversion), IF-CMD-ERROR applies to all commands created.

If a command is not changed in an exit routine, this exit routine can no longer be activated for the command.

Activation of SYSCMD exits

If a SYSCMD exit is activated during execution of a procedure, it has no effect on the current procedure.

If an error occurs during the procedure run, a SYSCMD exit is not activated in error mode.

Likewise, SYSCMD exits are not activated when branches are made in the procedure with the SKIP command; the branch destination is provided only once.

A SYSCMD exit for command conversion is no longer activated if the command was not changed the first time the exit routine is called.

13.2 Program interfaces for the user

SDF-P provides the assembler programmer with the following interfaces:

Macro	**Function**
CLIEXPR	Evaluates SDF-P expressions
CLIGET	Interrogates the procedure interruption protection
CLISET	Sets explicit protection against program interruption
CMD	Calls SDF-P commands from within a program
GETVAR	Reads simple and complex variables
PUTVAR	Writes simple variables
SHOWSSA	Displays variable stream assignments
TRANSVV	Transmits variables via a variable stream
VARINF	Processes (modifies) complex variables

The PUTVAR, GETVAR, SHOWSSA, TRANSVV and VARINF macros can be used to address S variables, i.e. variables that can also be addressed via the SDF-P command interface.

Scope of variables

There are currently two scopes for S variables: procedure-local and task-global.

The “procedure-local” scope is set or addressed on the command level in the SCOPE operand with SCOPE = *PROCEDURE / *CURRENT. On the program level, this scope is set or addressed with SCOPE = *VISIBLE. Note that procedure-local variables exist only until the end of the procedure, while the program may continue to run. It is then no longer possible to access procedure-local variables within the program; any attempt to do so results in an error.

The “task-global” scope is set or addressed on the command level in the SCOPE operand with SCOPE = *TASK. On the program level, this scope is extended with SCOPE = *TASKONLY. Task-global variables can be accessed during the whole time that the program is running, regardless of whether they are visible in the surrounding procedure.

CLIEXPR

The CLIEXPR macro evaluates arithmetic, logical and string expressions. The expression is passed in an input field, the result is returned in an output field. It is possible to request a specific output format for the result (binary number, Boolean constant, string).

The macro can also be called using MF = M. Refer to the manual “Executive Macros” [7] for further details concerning the operand MF =... .

Operation	Operands
CLIEXPR	MF = E ,PARAM = <name 1..27> / (<integer 1..15>)
	MF = D [,PREFIX = <u>C</u> / prefix]
	MF = C ,PREFIX = <u>C</u> / prefix [,MACID = <u>LIE</u> / macid]
	MF = L ,INPUT@ = <pointer> ,INPUTL = <integer 0..2147483647> ,OUTPUT@ = <pointer> ,OUTPUTL = <integer 0..2147483647> ,VFORM = <u>*BY-VALUE</u> / *STRING ,OTYPE = <pointer> ,OACTL = <pointer> ,PROT@ = <u>NULL</u> / <pointer> ,PROTL = <u>0</u> / <integer 0..2147483647> ,OPROTL = <u>NULL</u> / <pointer>

Operands

<pointer> as used in the description always stands for an address specification of the format **A(**symbolic address**)** or for a register containing the address. Specification of a register is possible only with MF = M.

MF = E
Execute format of the macro call; generates an SVC.

PARAM
Specifies the address of the operand list to be evaluated for the macro call (macro call with MF = L).

= <name 1..27>
Specifies the symbolic address of the operand list.

= (<integer 1..15>)
Specifies the register which contains the address of the operand list.

MF = D
DSECT format of the macro call; generates a DSECT for the operand list. Each field has its own name plus additional equates where explanations are required.

PREFIX = C / prefix
Defines the first character of the field names; default = C.

MF = C
C format of the macro call; generates the data area (operand list) only. Each field has its own name plus additional equates where explanations are required. The standard header must be initialized by the user.

PREFIX = C / prefix
Defines the first character of the field names; default = C.

MACID = LIE / macid
Defines the second, third and fourth character of the field names; default = LIE.

MF = L
List format of the macro call; generates the data area (operand list) only, taking operand values specified in the macro call into account. The data area contains no field names and no additional equates. The standard header is initialized.

INPUT@ = <pointer>
Address of the field containing the expression to be evaluated. The expression must be specified as a string expression. The field must be word-aligned.

INPUTL = <integer 0..2147483647>
Length of the field containing the expression to be evaluated.

OUTPUT@ = <pointer>
Address of the field to which the evaluation result is to be written. The field must be word-aligned.

OUTPUTL = <integer 0..2147483647>
Length of the result field. The actual length of the result is entered in the field specified with OACTL=... .

VFORM =
Defines the output format of the result (binary number, Boolean constant or string).

VFORM = *BY-VALUE*
Integers are output as binary numbers (4-byte digits).
Boolean constants are output as X’00’ (for FALSE) or X’01’ (for TRUE).

VFORM = *STRING
Integers are output as a string of decimal digits.
Boolean constants are output as either of the strings ’FALSE’ or ’TRUE’.

PROT@ = NULL / <pointer>
Address of the field to which SDF-P messages are to be written. If there are more than one messages, they are written to the field consecutively. Each entry starts with a 2-byte length field, followed by 2 bytes of fill characters and the message text. Default: output to SYSOUT.
Notes
- Only messages of message class SDP are entered here, all other messages are output to SYSOUT.
- The message format (language, short or long form, etc.) depends on the settings made with the /MODIFY-MSG-ATTRIBUTES command.

PROTL = 0 / <integer 0..2147483647>
Length of the message field. If the length of the output exceeds the specified field length, messages are not truncated; the last message is not written to the field instead. Default: no message entered.
The actual length required for message output is entered in the field specified with OPROTL=... .

OACTL = <pointer>
Address of the field to which the actual length of the result is written. The field must have a length of 4 bytes and must be word-aligned.

OTYPE = <pointer>
Address of the field to which the type of the result is written. The field must have a length of 1 byte. Entries start with the character defined with PREFIX=.. and MACID=.. .
Meaning of the entries:

Entry	Meaning (type)
<prefix, macid>VSTR	String
<prefix, macid>VINT	Integer
<prefix, macid>VBOO	Boolean constant

OPROTL = NULL / <pointer>
Address of the field to which the actual message length is written. The field must have a length of 4 bytes. Default: no entry.

Notes

- Only simple values (base terms) are output as results; no complex expressions are returned.
- The expression to be evaluated must not contain any & replacements.
- The field specified with OTYPE=.. always contains the result type, even if VFORM=*STRING was specified. This enables the user to distinguish a string of digits from an integer, or the string “FALSE” from the Boolean constant FALSE.

Return codes

The table below lists the return codes in hexadecimal format.

Subcode 2	Subcode 1	Maincode	Meaning
00	00	0000	Normal execution
01	00	0000	Overflow of PROT field (Warning)
00	40	0001	Syntax error in expression to be evaluated
01	40	0001	Overflow of PROT field
00	40	0002	Error during evaluation
01	40	0002	Overflow of PROT field
00	40	0003	Output field too short
00	01	0004	Input field not specified or not aligned
01	01	0004	Output field not specified or not aligned
02	01	0004	Log field (not aligned)
03	01	0004	Other fields (not aligned)
04	01	0004	Field address specified but field not accessible
00	40	0005	Insufficient space in caller’s address space
01	20	0006	System error
00	40	0007	Invalid procedure format; macro execution has been aborted
00	01	FFFF	Wrong specification for UNIT or FUNCTION in standard header
00	02	FFFF	Requested function is not supported
00	03	FFFF	Wrong version specification in standard header

Layout of the DSECT (operand list)

```
    CLIEXPR MF=D,PREFIX=N
 1          MFTST MF=D,PREFIX=N,MACID=LIE,ALIGN=F,
 1                DMACID=LIE,SUPPORT=(E,D,C,M,L),DNAME=LIEMDL
 2 NLIEMDL  DSECT ,
 2                *,##### PREFIX=N, MACID=LIE #####
 1 *   Which output type
 1 NLIEVSTR EQU   1                        *STRING
 1 NLIEVINT EQU   2                        *INTEGER
 1 NLIEVBOO EQU   3                        *BOOLEAN
 1 *
 1 *   parameter area description
 1 NLIEHDR  FHDR  MF=(C,NLIE),EQUATES=NO                   Standard header
 2 NLIEHDR  DS    0A
 2 NLIEFHE  DS    0XL8            0   GENERAL PARAMETER AREA HEADER
 2 *
 2 NLIEIFID DS    0A              0   INTERFACE IDENTIFIER
 2 NLIEFCTU DS    AL2             0   FUNCTION UNIT NUMBER
 2 *                                  BIT 15    HEADER FLAG BIT,
 2 *                                  MUST BE RESET UNTIL FURTHER NOTICE
 2 *                                  BIT 14-12 UNUSED, MUST BE RESET
 2 *                                  BIT 11-0  REAL FUNCTION UNIT NUMBER
 2 NLIEFCT  DS    AL1             2   FUNCTION NUMBER
 2 NLIEFCTV DS    AL1             3   FUNCTION INTERFACE VERSION NUMBER
 2 *
 2 NLIERET  DS    0A              4   GENERAL RETURN CODE
 2 NLIESRET DS    0AL2            4   SUB RETURN CODE
 2 NLIESR2  DS    AL1             4   SUB RETURN CODE 2
 2 NLIESR1  DS    AL1             5   SUB RETURN CODE 1
 2 NLIEMRET DS    0AL2            6   MAIN RETURN CODE
 2 NLIEMR2  DS    AL1             6   MAIN RETURN CODE 2
 2 NLIEMR1  DS    AL1             7   MAIN RETURN CODE 1
 2 NLIEFHL  EQU   8               8   GENERAL OPERAND LIST HEADER LENGTH
 2 *
 1 *   main return codes
 1 NLIESUCC EQU   0                        No error detected
 1 NLIESYNT EQU   1                        Syntax error
 1 NLIEEVAL EQU   2                        Semantic error
 1 NLIETRUN EQU   3                        Output buffer too small
 1 NLIEAREA EQU   4                        Buffer missing or not aligned
 1 *                                       or not accessible
 1 NLIEREQM EQU   5                        Out of memory
 1 NLIEDUMP EQU   6                        Invalid SDF-P-BASYS
 1 *                                       processing
 1 NLIECTXT EQU   7                        Old procedure context
 1 *
 1 NLIEIPTR DS    A                        SDF-P expression
 1 NLIEOPTR DS    A                         Resulting value
 1 NLIEPPTR DS    A                         Resulting protocol
 1 NLIEILEN DS    F                         SDF-P expression
 1 NLIEOMAX DS    F                         Value attribute (maximum
 1 *                                        length)
 1 NLIEPMAX DS    F                         Protocol attribute (maximum
 1 *                                        length)
 1 NLIEFORM DS    FL1                       Value attribute ( string
 1 *                                        generation )
 1 * Desired output form
 1 NLIEFVAL EQU   0                         *BY-VALUE
 1 NLIEFSTR EQU   1                         *STRING
 1 *
 1 NLIERES1 DS    CL7                       Alignment
 1 NLIEOLEN DS    A                         Value length as FW-aligned
 1 *                                        4-byte field
 1 NLIEOTYP DS    A                         Value type as 1 byte field
 1 NLIEPLEN DS    A                         Protocol length as FW-aligned
 1 *                                        4-byte field
 1 NLIE#    EQU   *-NLIEHDR
```

Example

```
CLIEXPR  START
         BALR  3,0
         USING *,3
         CLIEXPR MF=E,PARAM=OPLIST
WROUT    WROUT OUT,TERM,PARMOD=31
TERM     TERM
*****    DEFINITIONS *****
OPLISTE  CLIEXPR MF=L,INPUT@=A(IF),INPUTL=10,OUTPUT@=A(OF),OUTPUTL=10,V-
               FORM=*BY-VALUE,OACTL=A(H1),OTYPE=A(H2)
         DS    0F
IF       DC    CL10'(8+3)'
         DS    0F
OUT      DC    Y(OUTP-OUT)
         DS    3X
         DC    C'OUTPUT:  '
OF       DS    cl10
OUTP     EQU   *
         DS    CL10
         DS    0F
H1       DS    CL4
         DS    0F
H2       DS    CL1
         END
```

CLIGET

The CLIGET macro enables a program to query whether protection is required against implicit interruptions.

CLIGET returns the setting of the INTERRUPT-ALLOWED operand, as specified in the SET-PROCEDURE-OPTIONS, MODIFY-PROCEDURE-OPTIONS or BEGIN-PROCEDURE commands.

For further details, refer to section “Uninterruptibility” on page 126.

Operation	Operands
CLIGET	MF = E ,PARAM = <string 1..8> / (integer 1..15)
	MF = D [,PREFIX = <u>C</u> / prefix]
	MF = C [,PREFIX = <u>C</u> / prefix] [,MACID = <u>LIS</u> / macid]
	MF = L

Operands

MF = E
Execute format of the macro call; generates an SVC.

PARAM
Specifies the address of the operand list to be evaluated for the macro call (address of the macro call with MF = L).

= <string 1..8>
Specifies the symbolic address of the operand list.

= (<integer 1..15>)
Specifies the register which contains the address of the operand list.

MF = D
DSECT format of the macro call; generates a DSECT for the operand list. The names which are generated begin with the letter C; they can be changed using PREFIX.

PREFIX = <u>C</u> / prefix
Defines the first character of the names which are generated.
Default: the generated names begin with the letter C. This letter can be changed by means of the PREFIX parameter.

MF = C
C format of the macro call; generates an operand list, whose symbolic names begin with the string CLIG. These can be changed using PREFIX and MACID.

PREFIX = <u>C</u> / prefix
Defines the first character of the names which are generated.
Default: the generated names begin with the letter C. This letter can be changed by means of the PREFIX parameter.

MACID = <u>LIG</u> / macid
A string, up to three characters long, which replaces characters 2 to 4 of the generated name. Default: LIG

MF = L
LIST format of the macro call; generates the operand list for the macro call with MF = E (Execute format); the macro call must be addressable by means of a symbolic address.

Output parameters

The output parameters are returned in the prescribed fields in the operand list. If replacement is carried out, the calling program must use the corresponding names in reading the operand list.

&P.INTA=INTERRUPT-ALLOWED
One-byte long field in which the procedure option “INTERRUPT-ALLOWED” is returned by the macro.
&P.INTN means NO, i.e. INTERRUPT-ALLOWED = *NO: the program must be protected against any implicit interruption.
&P.INTY means YES, i.e. INTERRUPT-ALLOWED = *YES: the program must not be protected against an implicit interruption.

Note

For &P.INTN

- $\boxed{\text{K2}}$ is rejected if K2-STXIT is not activated.
- //EXECUTE-SYSTEM-CMD and //HOLD-PROGRAM are rejected if SYSSTMT is not equal to SYSCMD (i.e. either a data terminal or a file or other).

Every other action affecting security (CMD, BKPT, K2-STXIT or SVC etc.) is the responsibility of the calling program. I.e. the program must not activate these macros and cause an interruption if these actions are requested by the end user.

Return codes

The table below lists the return codes in hexadecimal form. User program registers are unchanged.

Subcode2	Subcode1	Maincode	Meaning
00	00	0000	Macro call was successful; no error
00	01	0001	Parameter error; parameter too short
00	20	0004	System error
00	01	FFFF	Unknown unit or function number
00	02	FFFF	Function not available
00	03	FFFF	Wrong version of the operand list

CLISET

The CLISET macro explicitly protects programs against interruptions.

For further details, refer to section “Uninterruptibility” on page 126..

Operation	Operands
CLISET	MF = E ,PARAM = <name 1..8> / (integer 1..15)
	MF = D [,PREFIX = <u>C</u> / prefix]
	MF = C [,PREFIX = <u>C</u> / prefix] [,MACID = <u>LIS</u> / macid]
	MF = L [,EXNINT = *U / *Y/ *N]

Operands

MF = E
Execute format of the macro call; generates an SVC.

PARAM
Specifies the address of the operand list to be evaluated for the macro call (address of the macro call with MF = L).

= <string 1..8>
Specifies the symbolic address of the operand list.

= (<integer 1..15>)
Specifies the register which contains the address of the operand list.

MF = D
DSECT format of the macro call; generates a DSECT for the operand list. The names which are generated begin with the letter C; they can be changed using PREFIX.

PREFIX = <u>C</u> / prefix
Defines the first character of the names which are generated.
Default: the generated names begin with the letter C. This letter can be changed by specifying “prefix”.

MF = C
C format of the macro call; generates an operand list, whose symbolic names begin with the string CLIS. These can be changed using PREFIX and MACID.

PREFIX = <u>C</u> / prefix
Defines the first character of the names which are generated.
Default: the generated names begin with the letter C. This letter can be changed by specifying prefix.

MACID = <u>LIS</u> / macid
A string, up to three characters long, which replaces characters 2 to 4 of the generated name. Default: LIS

MF = L
LIST format of the macro call; generates the operand list for the macro call with MF = E (Execute format); the macro call must be addressable by means of a symbolic address.

EXNINT
This option is used to set the explicitly uninterruptible mode for a program.

= *U
The previous setting is left unchanged. When the first call is made, the value *N will be assumed here.

= *Y
The program is explicitly protected against interruptions.

= *N
The program is not explicitly protected against interruptions.

Notes

– The EXNINT option cannot be stacked when CLISET is called several times.

– EXNINT =*U is a dummy operand setting. It issues no information in SDF-P and also initiates no return codes except for “No error”.

– EXNINT=*Y effects the following system changes:
  – K2 is rejected if K2-STXIT is not activated.
  – //EXECUTE-SYSTEM-CMD is rejected
  – Statement and command HOLD-PROGRAM causes EOF to be returned

– Every other action affecting security (CMD, BKPT, K2-STXIT or SVC etc.) is the responsibility of the calling program.

Return codes

The table below lists the return codes in hexadecimal notation. User program registers are unchanged.

Subcode2	Subcode1	Maincode	Meaning
01	00	0000	No action (EXNINT=*U)
00	00	0000	Macro call was successful; no error
00	01	0001	Parameter error
01	00	0002	EXNINT=*Y already set previously, or EXNINT=*N already set previously
00	20	0004	System error
00	01	FFFF	Unknown unit or function number
00	02	FFFF	Function not available
00	03	FFFF	Wrong version of the operand list

CMD

The CMD macro call can be used to execute commands, including SDF-P commands, in assembler programs. For a detailed description of the CMD macro call, see the “Executive Macros” manual [7].

Command	Function
ASSIGN-STREAM	Assign S variable stream
BEGIN-STRUCTURE	Start static structure declaration
CALL-PROCEDURE	Start procedure
CLOSE-VARIABLE-CONTAINER	Close variable container
DECLARE-CONSTANT	Declare variable with constant value
DECLARE-ELEMENT	Declare structure element
DECLARE-VARIABLE	Declare variable
DELETE-STREAM	Delete S variable stream
DELETE-VARIABLE	Delete variable
END-STRUCTURE	Identify end of structure declaration
ENTER-PROCEDURE	Start procedure as background procedure
FREE-VARIABLE	Delete variable contents
IMPORT-VARIABLE	Import variable contents
INCLUDE-CMD	Pass command sequence
INCLUDE-PROCEDURE	Start INCLUDE procedure
MODIFY-PROCEDURE-OPTIONS	Modify procedure attributes
OPEN-VARIABLE-CONTAINER	Open variable container
READ-VARIABLE	Read in variable values
SAVE-VARIABLE-CONTAINER	Save variable container
SELECT-VARIABLE-ELEMENTS	Select elements of a list variable
SHOW-STREAM-ASSIGNMENT	Display assignment for S variable stream
SHOW-STRUCTURE-LAYOUT	Display element name of structure layout
SHOW-VARIABLE	Display variable contents
SHOW-VARIABLE-ATTRIBUTES	Display variable attributes
SHOW-VARIABLE-CONTAINER-ATTR	Display variable container attributes
SORT-VARIABLE	Sort elements of a list variable
TRANSMIT-BY-STREAM	Transmit variables with S variable stream

GETVAR

The GETVAR macro reads the contents of a variable. GETVAR can be used with simple variables and elements of complex variables.

Operation	Operands
GETVAR	MF = E ,PARAM = <name 1..8> / (<integer 1..15>)
	MF = D ,PREFIX = G / prefix
	MF = C ,PREFIX = G / prefix ,MACID = ETV / macid
	MF = L ,NAMLEN = <integer 1..255> ,NAMADR = <name 1..8> ,SCOPE = *VISIBLE / *TASKONLY ,MAXLEN = <integer 1..4096> ,VALADR = <name 1..8>

Operands

MF = E
Execute format of the macro call; generates an SVC.

PARAM
Designates the address of the operand list that is evaluated for the macro call (address of macro call with MF = L).

= <name 1..8>
Designates the symbolic address of the operand list.

= (<integer 1..15>)
Designates the register that contains the address of the operand list.

MF = D
DSECT format of the macro call; generates a DSECT for the operand list. The names generated begin with the letter G; they can be modified with PREFIX.

PREFIX = G /prefix
Defines the first character of the generated names.
Default: the generated names begin with the letter G.

MF = C
C format of the macro call; generates an operand list whose symbolic names begin with the string GETV. They can be changed with PREFIX and MACID.

PREFIX = G /prefix
Defines the first character of the generated names.
Default: the generated names begin with the letter G.

MACID = ETV / macid
A string of up to three characters that replaces characters 2 to 4 of the generated names. Default: ETV

MF = L
LIST format of the macro call; generates the operand list for the macro call with MF = E (Execute format); the macro call must be addressable by means of a symbolic address.

NAMLEN = <integer 1..255>
Designates the length of the variable name.

NAMADR = <name 1..8>
Designates the symbolic name of the variable name address.

SCOPE
Designates the scope of the variable.

= *VISIBLE
The variable is a procedure-local variable

= *TASKONLY
The variable is a task-global variable.

MAXLEN = <integer 1..4096>
Designates the length of the variable value.

VALADR = <string 1..8>
Designates the symbolic address of the variable value.

Return codes

The table below lists the return codes in hexadecimal notation.

Subcode2	Subcode1	Maincode	Meaning
00	00	0000	Macro call was successful; no errors
00	01	0001	Parameter error
00	01	0002	Syntax error in variable name
00	40	0003	Area too small
00	40	0004	Variable not declared
00	40	0005	Variable container not available
00	40	0006	Data type and variable value do not match
00	40	0008	Variable has no value
00	01	FFFF	Unknown unit or function number
00	02	FFFF	Function not available
00	03	FFFF	Wrong version of operand list

PUTVAR

The PUTVAR macro assigns a value to a variable. PUTVAR can be used with simple variables and elements of complex variables.

If the assignment refers to a simple variable which does not yet exist, it is created either if IMPLICIT-DECLARATION=YES and IMPDEC = *STD apply, or if the macro call specifies IMPDEC=*YES.

Complex variables may also be assigned a value if they are integer, Boolean, string or “any” variables.

Operation	Operands
PUTVAR	MF = E
	,PARAM = <name 1..8> / (<integer 1..15>)
	MF = D
	,PREFIX = P / prefix
	MF = C
	,PREFIX = P / prefix
	,MACID = UTV / macid
	MF = L
	,NAMLEN = <integer 1..255>
	,NAMADR = <name 1..8>
	,SCOPE = *VISIBLE / *TASKONLY
	,IMPDEC = *YES / *NO / *STD
	,VALLEN = <integer 0..4096>
	,VALADR = <name 1..8>
	,VALTYPE = *INTEGER / *BOOLEAN / *STRING

Operands

MF = E
Execute format of the macro call; generates an SVC.

PARAM
Designates the address of the operand list that is evaluated for the macro call (address of macro call with MF = L).

= <name 1..8>
Designates the symbolic address of the operand list.

= (<integer 1..15>)
Designates the register that contains the address of the operand list.

MF = D
DSECT format of the macro call; generates a DSECT for the operand list. The names generated begin with the letter P; they can be modified with PREFIX.

PREFIX = P / prefix
Defines the first character of the generated names.
Default: the generated names begin with the letter P.

MF = C
C format of the macro call; generates an operand list whose symbolic names begin with the string PUTV. They can be changed with PREFIX and MACID.

PREFIX = P / prefix
Defines the first character of the generated names.
Default: the generated names begin with the letter P.

MACID = UTV / macid
A string of up to three characters that replaces characters 2 to 4 of the generated names. Default: UTV

MF = L
LIST format of the macro call; generates the operand list for the macro call with MF = E (Execute format); the macro call must be addressable by means of a symbolic address.

NAMLEN = <integer 1..255>
Designates the length of the variable name.

NAMADR = <name 1..8>
Designates the symbolic name of the variable name address.

SCOPE
Defines the scope of the variable.

= *VISIBLE
The variable is created as a procedure-local variable.

= *TASKONLY
The variable is created as a task-global variable.

IMPDEC = *YES / *NO / *STD
Determines whether the variable is created implicitly if it does not yet exist, regardless of the setting in the surrounding procedure.

= *STD
Specifies that the attributes of IMPLICIT-DECLARATION are used for the current procedure.

VALLEN = <*INTEGER 0..4096>
Designates the length of the variable value.

VALADR = <name 1..8>
Designates the symbolic address of the variable value.

VALTYPE
Determines the data type of the variable.

= *INTEGER
The variable is assigned the data type INTEGER; assignment of anything but integer values results in an error.

= *BOOLEAN
The variable is assigned the data type BOOLEAN; assignment of anything but the value TRUE or FALSE results in an error.

= *STRING
The variable is assigned the data type STRING.
The maximum string length is defined with VALLEN-LENGTH.

Return codes

The table below lists the return codes in hexadecimal notation.

Subcode2	Subcode1	Maincode	Meaning
00	00	0000	Macro call was successful; no errors
00	01	0001	Parameter error
00	01	0002	Syntax error in variable name
00	40	0004	Variable not declared
00	40	0005	Variable container not available
00	40	0006	Data type and variable value do not match
00	01	FFFF	Unknown unit or function number
00	02	FFFF	Function not available
00	03	FFFF	Wrong version of operand list

SHOWSSA

The SHOWSSA macro shows the current assignment of the specified S variable stream. It is functionally equivalent to the SHOW-STREAM-ASSIGNMENT command. However, it is not possible to specify a list of streams with SHOWSSA.

Operation	Operands
SHOWSSA	MF = E ,PARAM = <name 1..8> / (<integer 1..15>)
	MF = D ,PREFIX = <u>S</u> / prefix
	MF = C ,PREFIX = <u>S</u> / prefix ,MACID = <u>HOW</u> / macid
	MF = L / M ,STREAM = <u>*ALL</u> / *STD_STREAMS / <c-string 1..20 with-wild> ,INFO = <u>*ASSIGNMENT</u> / *DESTINATION ,OUTPUT = <u>*RETURN_CODE</u> / *SYSOUT / *SYSLST

Operands

MF = E
Execute format of the macro call; generates an SVC.

PARAM
Designates the address of the operand list that is evaluated for the macro call (address of macro call with MF = L).

= <name 1..8>
Designates the symbolic address of the operand list.

= (<integer 1..15>)
Designates the register that contains the address of the operand list.

MF = D
DSECT format of the macro call; generates a DSECT for the operand list. The names generated begin with the letter S; they can be modified with PREFIX.

PREFIX = <u>S</u> / prefix
Defines the first character of the generated names.
Default: the generated names begin with the letter S.

MF = C
C format of the macro call; generates an operand list whose symbolic names begin with the string SHOW. They can be changed with PREFIX and MACID.

PREFIX = <u>S</u> / prefix
Defines the first character of the generated names.
Default: the generated names begin with the letter S.

MACID = <u>HOW</u> / macid
A string of up to three characters that replaces characters 2 to 4 of the generated names. Default: HOW.

MF = L / M
LIST format of the macro call; generates the operand list for the macro call with MF = E (Execute format); the macro call must be addressable by means of a symbolic address.

STREAM
Name of the S variable stream which is to be output. Lists are not supported.

= <u>*ALL</u>
All the S variable streams which are visible in the current procedure will be listed.

= *STD_STREAMS
All the standard streams which are implemented in the system will be displayed. The names of all these streams are prefixed by “SYS”. The names which will be listed are those contained in the value list of the syntax specification for the STREAM-NAME operand of the ASSIGN-STREAM command.

= <c-string 1..20>
The S variable stream which is to be displayed. When wildcards are used, all the S variable streams which match this search pattern will be displayed.

INFORMATION
Specifies which data items must be output.

= <u>*ASSIGNMENT</u>
The “TO” value in the ASSIGN-STREAM command is to be output.
If this stream name is assigned to another stream name, the latter will be output.

= *DESTINATION
The name of the current server which is linked to the S variable stream will be output. If the variable stream is assigned to another variable stream name, the last assignment will be output.

OUTPUT
Specifies where the output of the command is to be sent. Output to an S variable is not possible.

= RETURN_CODE
Only a return code is returned. This is the default setting. This value is not advisable unless STREAM=<c-string 1..20> (i.e. a name without wildcards) was specified .

= *SYSOUT
Output to SYSOUT.

= *SYSLST
Output to SYSLST.

Return codes

The table below lists the return codes in hexadecimal notation.

Subcode2	Subcode1	Maincode	Meaning
00	00	0000	Macro call was successful; no errors
00	01	0001	Parameter error
00	40	0002	Specified variable stream is incomplete
00	40	0003	SSTA error (SSTA too small, not initialized, ...)
00	40	0004	Terminated by K2 during output to SYSOUT
00	40	0005	Error during output to SYSOUT
00	40	0006	Error during output to SYSLST
02	01	0007	More than one variable stream for OUTPUT=*RETURNCODE
00	20	0008	System error
02	00	000A	Specified variable stream is assigned to *DUMMY
02	00	000B	Specified variable stream is already assigned
02	00	000C	Specified variable stream does not exist
00	01	FFFF	Unknown unit or function number
00	02	FFFF	Function not available
00	03	FFFF	Wrong version of the operand list
00	41	FFFF	SDF-P is not loaded
00	81	FFFF	SDF-P no longer working

TRANSVV

The TRANSVV macro is used by a client to carry out a variable transmission via the specified S variable stream to the server that is currently assigned (ASSIGN-STREAM command). TRANSVV is functionally equivalent to the TRANSMIT-BY-STREAM command. TRANSVV can only use S variable streams that were assigned at the same procedural hierarchy level at which the program was started.

Operation	**Operands**
TRANSVV	MF = E
	,PARAM = <name 1..8> / (<integer 1..15>)
	MF = D
	[,PREFIX = T / prefix]
	MF = C
	[,PREFIX = T / prefix]
	[,MACID = RAN / macid]
	MF = L/M
	,STREAM = <name 1..20>
	[,VNAME = *NONE / <name 1..8> / (integer 1..15)]
	[,VNAMEL = <integer 1..255>]
	[,VSCOPE = *VISIBLE / *TASKONLY]
	[,RNAME = *SAME / *NONE / <name 1..8> / (integer 1..15)]
	[,RNAMEL = <integer 1..255>]
	[,RSCOPE = *VISIBLE / *TASKONLY]
	[,CNAME = *NONE / <name 1..8> / (integer 1..15)]
	[,CNAMEL = <integer 1..255>]
	[,CSCOPE = *VISIBLE / *TASKONLY]
	[,RCNAME = *SAME / *NONE / <name 1..8> / (integer 1..15)]
	[,RCNAMEL = <integer 1..255>]
	[,RCSCOPE = *VISIBLE / *TASKONLY]

Operands

MF = E
Execute format of the macro call; generates an SVC.

PARAM
Designates the address of the operand list that is evaluated for the macro call (address of macro call with MF = L).

= <name 1..8>
Designates the symbolic address of the operand list.

= (<integer 1..15>)
Designates the register that contains the address of the operand list.

MF = D
DSECT format of the macro call; generates a DSECT for the operand list. Each field has its own name plus additional equates where explanations are required.

PREFIX = <u>T</u> / prefix
Defines the first character of the generated names.
Default: the generated names begin with the letter T.

MF = C
C format of the macro call; generates the data area (operand list) only. Each field has its own name plus additional equates where explanations are required. The standard header must be initialized by the user.

PREFIX = <u>T</u> / prefix
Defines the first character of the generated names.
Default: the generated names begin with the letter T.

MACID = <u>RAN</u> / macid
Defines the second, third and fourth character of the field names; default: RAN.

MF = L / M
List format of the macro call; generates the operand list for the macro call with MF = E (Execute format); the macro call must be addressable by means of a symbolic address.

STREAM = <name 1..20>
Name of the S variable stream into which the variable is transmitted.

VNAME
Name of the S variable which is to be transmitted to the server.

= <u>*NONE</u>
No S variable is transmitted.

= <name 1..8>
Address of the field which contains the name of the S variable.

= (<integer 1..15>)
Register with the address of the field that contains the name of the S variable (the register number must be enclosed in parentheses).

VNAMEL = <integer 1..255>
Specifies the length of the variable name which was specified by the caller.

VSCOPE
Defines the scope of the variable.

= *VISIBLE
The variable is created as a procedure-local variable.

= *TASKONLY
The variable is created as a task-global variable.

The following are permissible combinations:

```
VNAME=*NONE
VSCOPE=*VISIBLE
```

or:

```
VSCOPE=*VISIBLE / *TASKONLY,
VNAME =<name 1..8> / (integer 1..15),
VNAMEL=<integer 1..255>
```

RNAME
The S variable or return variable which is sent back by the transmission.

= *SAME
The values of VNAME, VNAMEL and VSCOPE are retained.

= *NONE
No return variable is sent back.

= <name 1..8>
Address of the field which contains the name of the return variable.

= (<integer 1..15>)
Register with the address of the field that contains the name of the return variable (the register number must be enclosed in parentheses).

RNAMEL = <integer 1..255>
Specifies the length of the variable name which was specified by the caller.

RSCOPE = *VISIBLE / *TASKONLY
Defines the pool or container for the return variables.

= *VISIBLE
The return variable is created as a procedure-local variable.

= *TASKONLY
The return variable is created as a task-global variable.

The following are permissible combinations:

```
RNAME=*NONE
```

or:

```
RSCOPE=*VISIBLE / *TASKONLY,
RNAME =<name 1..8> / (integer 1..15),
RNAMEL=<integer 1..255>
```

CNAME
Control variable sent with the transmission.

= *NONE
No control variable is transmitted. This may be specified as either another variable or a register number (which must be enclosed in parentheses).

= <name 1..8>
Address of the field which contains the name of the control variable.

= (<integer 1..15>)
Register with the address of the field that contains the name of the control variable (the register number must be enclosed in parentheses).

CNAMEL = <integer 1..255>
Specifies the length of the control variable name which was specified by the caller.

CSCOPE
Defines the pool or container for the control variables.

= *VISIBLE
The control variable is created as a procedure-local variable.

= *TASKONLY
The control variable is created as a task-global variable.

The following are permissible combinations:

```
CNAME=*NONE
CSCOPE=*VISIBLE / *TASKONLY
CNAME =<name 1..8> / (integer 1..15), CNAMEL=<integer 1..255>
```

RCNAME
The control variable or return control variable which is sent back by the transmission.

= *SAME
The values of CNAME, CNAMEL and CSCOPE are retained.

= *NONE
No return control variable is transmitted.

= <name 1..8>
Address of the field which contains the name of the return control variable.

= (<integer 1..15>)
Register with the address of the field that contains the name of the return control variable (the register number must be enclosed in parentheses).

RNAMEL = <integer 1..255>
Specifies the length of the return control variable name which was specified by the caller.

RCSCOPE
Defines the pool or container for the return control variables.

= *VISIBLE
The return control variable is created as a procedure-local variable.

= *TASKONLY
The return control variable is created as a task-global variable.

The following are permissible combinations:

```
RCNAME=*NONE
RCSCOPE=*VISIBLE / *TASKONLY
RCNAME =<name 1..8> / (integer 1..15), RCNAMEL=<integer 1..255>
```

Return codes

The table below lists the return codes in hexadecimal notation.

Subcode2	Subcode1	Maincode	Meaning
00	00	0000	Transmission successfully completed; no error
01	00	0000	Variable stream was assigned to *DUMMY, no transmission
00	01	0001	Parameter error
00	40	0002	Specified variable stream is incomplete
00	40	0003	Specified variable is incomplete
00	40	0004	RET-SSTA too small (for developers only)
00	01	0005	The data items transmitted (user or control data) do not have a format compatible with one the server can process
00	40	0006	Error message from the server; saved in RCNAME (if specified)
02	00	0007	Warning from the server; saved in RCNAME (if specified)
02	00	0008	Variable stream reset to *DUMMY; server is no longer active
00	20	0009	System error
00	20	000A	Error during server connection
00	01	FFFF	Unknown unit or function number
00	02	FFFF	Function is not available
00	03	FFFF	Wrong version of the operand list
00	41	FFFF	SDF-P is not loaded
00	81	FFFF	SDF-P no longer working

VARINF

The VARINF macro can be used to analyze complex variables whose elements are themselves complex variables.

Operation	Operands
VARINF	MF = E ,PARAM = <name 1..8> / (<integer 1..15>)
	MF = D ,PREFIX = V / prefix
	MF = C ,PREFIX = V / prefix ,MACID = ARI / macid
	MF = L ,NAMLEN = <integer 1..255> ,NAMADR = <name 1..8> ,SCOPE = *VISIBLE / *TASKONLY ,POSIT = *CURRENT / *UP / *DOWN / *NEXT ,MAXLEN = <integer 1..4096> ,RESADR = <name 1..8>

Operands

MF = E
Execute format of the macro call: generates an SVC.

PARAM
Designates the address of the operand list that is evaluated for the macro call (address of macro call with MF = L).

= <name 1..8>
Designates the symbolic address of the operand list.

= (<integer 1..15>)
Designates the register that contains the address of the operand list.

MF = D
DSECT format of the macro call: generates a DSECT for the operand list. The names generated begin with the string VARINF; they can be modified with PREFIX.

PREFIX = V / prefix
Defines the first character of the generated names.
Default: the generated names begin with the letter V.

MF = C
C format of the macro call: generates an operand list whose symbolic names begin with the string VARI. They can be changed with PREFIX and MACID.

PREFIX = V / prefix
Defines the first character of the generated names.
Default: the generated names begin with the letter V.

MACID = ARI / macid
A string of up to three characters that replaces characters 2 to 4 of the generated names. Default: ARI

MF = L
LIST format of the macro call: generates the operand list for the macro call with MF = E (Execute format); the macro call must be addressable by means of a symbolic address.

NAMLEN = <integer 1..255>
Designates the length of the variable name.

NAMADR = <name 1..8>
Designates the symbolic name of the variable name address from which the element names can be queried.

SCOPE
Designates the scope of the variable.

= *VISIBLE
The variable is a procedure-local variable.

= *TASKONLY
The variable is a task-global variable.

POSIT
Determines the variable element whose name is to be returned. Positioning is relative rather than absolute, based on the variable element that was last accessed.

= *CURRENT
Returns the name of the current variable element, i.e. of the variable element that serves as a starting point for positioning (existence check).

= *UP
Returns the name of the complex variable to which the current variable element belongs.

= *DOWN
If the current variable element is itself a complex variable, POSITION=DOWN returns the name of the first element of this complex variable.

= *NEXT
Returns the name of the next complex variable on the same level.

MAXLEN = <integer 1..4096>
Designates the maximum length of the field in which the variable name is returned.

RESADR = <string 1..8>
Symbolic address of the field in which the variable name is returned.

The following output fields will be found in the operand list after the call:

```
<PR> = <prefix><macid>

<PR>VALL : length of the result name

<PR>VTYP : Variable type:
        Possible values:     <PR>VANY: *ANY
                             <PR>VSTR: *STRING
                             <PR>VINT: *INTEGER
                             <PR>VBOO: *BOOLEAN
                             <PR>VSTU: *STRUCTURE

<PR>MULT : MULTIPLE-ELEMENTS:

          Possible values:   <PR>MNO: *NONE
                             <PR>MARR: *ARRAY
                             <PR>MLIS: *LIST

<PR>SINF : STRUCTURE INFORMATION (relevant if: <PR>VTYP=<PR>VSTU)

          Possible values:   <PR>SDYN: *DYNAMIC
                             <PR>SCMD: *BY-SYSCMD
                             <PR>SLAY: LAYOUT
```

Return codes

The table below lists the return codes in hexadecimal notation.

Subcode2	Subcode1	Maincode	Meaning
00	00	0000	Macro call was successful; no error
00	01	0001	Parameter error
00	01	0002	Syntax error in variable name
00	40	0003	Area too small
00	40	0004	Variable not declared
00	40	0005	Variable container not available
00	40	0007	Last variable element was reached; no further variable elements present
00	01	FFFF	Unknown unit or function number
00	02	FFFF	Function not available
00	03	FFFF	Wrong version of operand list

14 Predefined functions

This chapter contains a detailed description of the predefined (or built-in) functions supplied with SDF-P, listed in alphabetical order.

Each entry has the following format:

– function name, including abbreviated name
– assignment to domains (application areas)
– function description
– format representation for the function call
– result type
– description of input parameters
– description of result values
– error messages
– examples

In some instances, the input parameters are assigned keywords as values; the meaning of these keywords is explained in the description of the input parameters.

For most input parameters, however, the current value can be freely defined by the user; the following names are used:

– string_expression: programmers can directly specify a string ('string'), the name of a variable which contains a string (varname) or an expression which supplies a string as the result.
– character: same as string_expression; however, the string to be used consists of only one character.
– arithm_expression: users can directly specify an integer value (integer), the name of a variable which contains an integer (varname) or an expression which supplies an integer as the result.
– abbreviated_name( ): the names of functions cannot be abbreviated. Some functions can be called by means of an abbreviated name in addition to the function name. This abbreviated name is placed directly beneath the function name.

Note

The following syntax error messages can occur for any of the predefined functions, and are not listed separately for each of the functions individually: SDP0005, SDP0006, SDP0008, SDP0009, SDP0010, SDP0039, SDP0099, SDP0300, SDP0304, SDP0306, SDP0402, SDP0431, SDP0444.

ACCOUNT( ) Request account number

Domain: **Task-specific environment information**

The ACCOUNT( ) function determines the account number of the current task which was specified in the SET-LOGON-PARAMETERS command.

Format

ACCOUNT( )

Result type

STRING (<string 1 .. 8>)

Input parameters

None

Result

Account number of no more than eight digits

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

```
/A = ACCOUNT
/SHOW-VARIABLE A

A = K27DKU
```

ARRAY-INDEX( ) Request array index

Domain: **Variable access (variable name)**

The ARRAY-INDEX( ) function can be applied to arrays. ARRAY-INDEX supplies the value of an array index. As a result, other functions can then explicitly request this element via the array index.

Format

```
ARRAY-INDEX( )
--------------------------------------------------------------
 ARRAY-NAME = string_expression
,INDEX = *FIRST / *LAST / *LOWER-BOUND / *UPPER-BOUND
```

Result type

INTEGER

Input parameters

ARRAY NAME = string_expression
Designates an array.

INDEX =
Specifies which array index is to be requested.

INDEX = *FIRST
Array index of the first element in the array containing a valid value.

INDEX = *LAST
Array index of the last element in the array containing a valid value.

INDEX = *LOWER-BOUND
Array index defined in the variable declaration with the DECLARE-VARIABLE command in the operand MULTIPLE-ELEMENTS = *ARRAY (LOWER-BOUND =).

INDEX = *UPPER-BOUND
Array index defined in the variable declaration with the DECLARE-VARIABLE command in the operand MULTIPLE-ELEMENTS = *ARRAY (UPPER-BOUND =).

Result

Index of the array element, returned as an integer.

Error messages

SDP0423	VARIABLE '(&00)' NOT AN ARRAY
SDP1007	NO VARIABLE DECLARED
SDP1052	AGGREGATE ELEMENT NOT PRESENT
SDP1101	SYNTAX ERROR IN VARIABLE NAME

Example

The array AR is declared and contains the following:

```
/DECLARE-VARIABLE AR,TYPE = *STRING, MULTIPLE-ELEMENTS = *ARRAY
/AR#2 = 'abc'
/AR#3 = 'cde'
 /AR#4 = ' '
```

```
/ARIND = ARRAY-INDEX('AR', *FIRST)
/SHOW-VARIABLE ARIND
ARIND = 2
```

```
/ARIND = ARRAY-INDEX('AR', *LAST)
 /SHOW-VARIABLE ARIND
ARIND = 4
```

```
/ARIND = ARRAY-INDEX('AR', *LOWER-BOUND)
 /SHOW-VARIABLE ARIND
ARIND = 0
```

```
/ARIND = ARRAY-INDEX('AR', *UPPER-BOUND)
/SHOW-VARIABLE ARIND
ARIND = 2147483647
```

BOOLEAN( ) Convert to Boolean value

Domain: **Conversion functions**
The BOOLEAN( ) function converts the expression specified in the function call to a BOOLEAN value.

Format

BOOLEAN( )
BOOLE( )

EXPRESSION = expression

Result type

BOOLEAN

Input parameters

EXPRESSION = expression
Determines the expression to be converted. If it was not possible to convert “expression”, an error message is displayed.

Result

Data type	**Result**

expression = ' TRUE' or ’true’
expression = ' FALSE' or ’false’
	TRUE
	FALSE

expression = 0 expression not equal to 0	FALSE TRUE

Error message

```
SDP0429   CONVERSION NOT POSSIBLE
```

Example

```
/A = 0
/B = BOOLEAN(EXPRESSION = A)
/SHOW-VARIABLE B
B = FALSE
```

CHARACTER-TO-INTEGER( ) Convert character to integer

Domain: **Conversion functions**

The CHARACTER-TO-INTEGER( ) function converts *one* character to a decimal number based on the characters EBCDIC code.

If the input string consists of several characters, then only the first character is converted.

All characters in a string can be converted in combination with the corresponding string function (e.g. SUBSTRING).

Format

CHARACTER-TO-INTEGER( )
CHAR-TO-INT( )

STRING = string_expression

Result type

INTEGER (<integer 0..255>)

Input parameters

STRING = string_expression
Designates the string whose first character is to be converted.
If “string_expression” designates a null string, an error message is output.

Result

Integer <integer 0..255>

Error message

```
SDP0417   SPECIFIED STRING EMPTY. FUNCTION NOT EXECUTED
```

Example 1: Converting a character

```
C = CHARACTER-TO-INTEGER(STRING = 'ABC')
/SHOW-VARIABLE C
C = 193
```

In EBCDI code, the first character in the string (the A) is shown as X'C1' in half-byte notation and as B'11000001' in binary notation. This corresponds to the number 193 in the decimal system (= 128 + 64 + 1). CHARACTER-TO-INTEGER( ) thus converts the letter A to the number 193.

Example 2: Converting all characters in a string

```
/BEGIN-BLOCK
/   INT = 1
/   CSTRING = 'ABC'
/   SUBST: CONVSTRING = SUBSTRING(CSTRING, INT)
/   CODE = CHARACTER-TO-INTEGER(STRING = CONVSTRING)
/   SHOW-VARIABLE CODE
/   INT = INT+1
/   IF (INT < 4)
/      GOTO SUBST
/   END-IF
/END-BLOCK
```

SHOW-VARIABLE outputs the numbers 193, 194 and 195 consecutively in a loop.

CHECK-DATA-TYPE( ) Check operand value

Domain: **String functions/checking functions**

The CHECK-DATA-TYPE( ) function checks the data type of strings or operand values to determine whether they satisfy SDF data type requirements (for details, see “Data types” on page 546 and “Suffixes for data types” on page 552ff).
CHECK-DATA-TYPE( ) is used to specify the data type which the input value 'INPUT' must satisfy. This data type conforms to the SDF data type rules, as specified in the //ADD-VALUE statement in the SDF-A program (see the “SDF-A” manual [16]). I.e. the operands of CHECK-DATA-TYPE( ) are fully compatible with those of //ADD-VALUE. Only operand combinations created in conformity with the syntax of //ADD-VALUE are considered. Other combinations will be ignored. Operand combinations are dependent on what is specified for DATA-TYPE. The following list contains a brief summary of all valid operand combinations.

DATA-TYPE=	**Valid operand combinations**
*NOCHECK	VALUE, PATTERN
*INTEGER	VALUE, SHORTEST-LENGTH, LONGEST-LENGTH
*X-STRING	VALUE, SHORTEST-LENGTH, LONGEST-LENGTH, ODD
*C-STRING	VALUE, SHORTEST-LENGTH, LONGEST-LENGTH
*NAME	VALUE, SHORTEST-LENGTH, LONGEST-LENGTH, UNDERSCORE
*ALPHANUMERIC-NAME	VALUE, SHORTEST-LENGTH, LONGEST-LENGTH
*STRUCTURED-NAME	VALUE, SHORTEST-LENGTH, LONGEST-LENGTH
*FILENAME *PARTIAL-FILENAME *POSIX-FILENAME *POSIX-PATHNAME	VALUE, SHORTEST-LENGTH, LONGEST-LENGTH, PATTERN, CAT-ID, USER-ID, VERSION, GENERATION, WILDCARD
*TIME	VALUE
*DATE	VALUE
*COMPOSED-NAME	VALUE, SHORTEST-LENGTH, LONGEST-LENGTH
*TEXT	VALUE, SHORTEST-LENGTH, LONGEST-LENGTH
*CAT-ID	VALUE
*KEYWORD	VALUE, KEYSTAR
*KEYWORD-NUMBER	VALUE, KEYSTAR
*VSN	VALUE
*X-TEXT	VALUE, SHORTEST-LENGTH, LONGEST-LENGTH
*FIXED	VALUE, SHORTEST-LENGTH, LONGEST-LENGTH, DECIMAL-DIGITS-SHORTEST, DECIMAL-DIGITS-LONGEST

DATA-TYPE=	**Valid operand combinations**
*DEVICE	VALUE, ALIAS, VOLUME-ONLY, DEVICE-CLASS, EXCEPT-DISKS, EXCEPT-TAPES
*PRODUCT-VERSION	VALUE, CORRECTION-STATE, USER-INTERFACE

Format

(part 1 of 2)

CHECK-DATA-TYPE( )
INPUT = string_expression
,DATA-TYPE = *NOCHECK / *INTEGER / *X-STRING / *C-STRING / *NAME / *ALPHANUMERIC-NAME/ *STRUCTURED-NAME / *FILENAME / *FULL-FILENAME / *PARTIAL-FILENAME / *POSIX-FILENAME / *POSIX-PATHNAME / *TIME / *DATE / *COMPOSED-NAME / *TEXT / *CAT-ID / *KEYWORD / *KEYWORD-NUMBER / *VSN / *X-TEXT / *FIXED / *DEVICE / *PRODUCT-VERSION
,SHORTEST-LENGTH = *ANY / arithm_ausdruck
,LONGEST-LENGTH = *ANY / arithm_ausdruck
,LONGEST-LOGICAL-LENGTH = *NONE / arithm_ausdruck
,DECIMAL-DIGITS-SHORTEST = 0 / arithm_ausdruck
,DECIMAL-DIGITS-LONGEST = 0 / arithm_ausdruck
,VALUE = *NO / list-poss: string_expression
,PATTERN = *NO / string_expression
,CAT-ID = *YES / *NO
,USER-ID = *YES / *NO
,VERSION = *YES / *NO
,GENERATION = *YES / *NO
,WILDCARD = *NO / *YES
,KEYSTAR = *NO / *YES
,SEPARATORS = *YES / *NO
,UNDERSCORE = *NO / *YES
,ODD = *YES / *NO
,CORRECTION-STATE = *YES / *NO / *ANY
,USER-INTERFACE = *YES / *NO / *ANY
,ALIAS = *YES / *NO
,VOLUME-ONLY = *NO / *YES
,WILDCARD-TYPE = *SELECTOR / *CONSTRUCTOR
,LOWER-CASE = *NO / *YES
,QUOTES = *OPTIONAL / *MANDATORY
,TEMPORARY-FILE = *YES / *NO

continued ➠

(part 2 of 2)

```
,SCOPE = *ALL / *STD-DISK
,DEVICE-CLASS = *DISK / *TAPE / *DISK-OR-TAPE
,EXCEPT-DISKS = *NONE / list-poss: string_expression
,EXCEPT-TAPES = *NONE / list-poss: string_expression
```

Result type

BOOLEAN

Input parameters

INPUT = string_expression
Specifies the operand value to be checked.

DATA-TYPE =
Specifies the data type checking criterion.

DATA-TYPE = *NOCHECK
No check is carried out on the data type of the operand value. The only check will be for a match against the wildcard search pattern.
In this case, PATTERN = *NO must not be specified.

DATA-TYPE = *INTEGER
The operand value will be checked to determine if it has the data type integer.

DATA-TYPE = *X-STRING
The operand value will be checked to determine if it has the data type x-string.

DATA-TYPE = *C-STRING
The operand value will be checked to determine if it has the data type c-string.

DATA-TYPE = *NAME
The operand value will be checked to determine if it has the data type name.

DATA-TYPE = *ALPHANUMERIC-NAME
The operand value will be checked to determine if it has the data type alphanumeric-name.

DATA-TYPE = *STRUCTURED-NAME
The operand value will be checked to determine if it has the data type structured-name.

DATA-TYPE = *FILENAME
The operand value will be checked to determine if it has the data type filename.

DATA-TYPE = *FULL-FILENAME
The operand value will be checked to determine if it has the data type full-filename.
The *FULL-FILENAME specification is supported for compatibility reasons only. As of SDF V 4.1A, data type full-filename will be represented as filename at the user interface.

DATA-TYPE = *PARTIAL-FILENAME
The operand value will be checked to determine if it has the data type partial-filename.

DATA-TYPE = *POSIX-FILENAME
The operand value will be checked to determine if it has the data type posix-filename.

DATA-TYPE = *POSIX-PATHNAME
The operand value will be checked to determine if it has the data type posix-pathname.

DATA-TYPE = *TIME
The operand value will be checked to determine if it has the data type time.

DATA-TYPE = *DATE
The operand value will be checked to determine if it has the data type date.

DATA-TYPE = *COMPOSED-NAME
The operand value will be checked to determine if it has the data type composed-name.

DATA-TYPE = *TEXT
The operand value will be checked to determine if it has the data type text.

DATA-TYPE = *CAT-ID
The operand value will be checked to determine if it has the data type cat-id.

DATA-TYPE = *KEYWORD
The operand value will be checked to determine if it has the data type keyword.

DATA-TYPE = *KEYWORD-NUMBER
The operand value will be checked to determine if it has the data type keyword-number.

DATA-TYPE = *VSN
The operand value will be checked to determine if it has the data type vsn.

DATA-TYPE = *X-TEXT
The operand value will be checked to determine if it has the data type x-text.

DATA-TYPE = *FIXED
The operand value will be checked to determine if it has the data type fixed.

DATA-TYPE = *DEVICE
The operand value will be checked to determine if it has the data type device.

DATA-TYPE = *PRODUCT-VERSION
The operand value will be checked to determine if it has the data type product-version.

SHORTEST-LENGTH = *ANY / arithm_expression
Irrelevant for the date, time, cat-id, keyword and keyword-number data types.
Determines whether the operand value has to satisfy a minimum length in terms of characters or number of bytes (for the x-string data type).
Irrelevant for data types date, time, cat-id, keyword and keyword-number.
For the data type integer, SHORTEST-LENGTH indicates its lowest value.
For the data type fixed, SHORTEST-LENGTH must be combined with DECIMAL-DIGITS-SHORTEST.

LONGEST-LENGTH = *ANY / arithm_expression
Irrelevant for the date, time, cat-id, keyword and keyword-number data types.
Determines whether the operand value has to satisfy a maximum length in terms of characters or number of bytes (for the x-string data type).
Irrelevant for data types date, time, cat-id, keyword and keyword-number.
For the data type integer, LONGEST-LENGTH indicates its highest value.
For the data type fixed, LONGEST-LENGTH must be combined with DECIMAL-DIGITS-LONGEST.

LONGEST-LOGICAL-LENGTH = *NONE / arithm_expression
Only relevant in conjunction with PATTERN = string_expression.
Determines the maximum length within the operand up to which a match is to be sought against a wildcard expression.

LONGEST-LOGICAL-LENGTH = *NONE
The maximum length for the specified data type will be set by SDF.

DECIMAL-DIGITS-SHORTEST = 0 / arithm_expression
Only relevant for the data type fixed.
Determines the minimum number of decimal places which the operand value may have.

DECIMAL-DIGITS-LONGEST = 0 / arithm_expression
Only relevant for the data type fixed.
Determines the maximum number of decimal places which the operand value may have.

VALUE =
Determines what values are permissible as inputs.

VALUE = *NO
Any values which correspond to the specified operand type are permissible. The only restrictions which will be applied are any identified in the operand type specification (e.g. length). *NO is not permitted for operand values of type keyword.

VALUE = list-poss: string_expression
The permitted values are restricted to the values listed. The user can abbreviate the specified values on input. A list of individual values cannot be used for values of type keyword (a specific CHECK-DATA-TYPE must be entered for each individual value).

PATTERN =
Wildcard search pattern, used in searching for the operand value.

PATTERN = <u>*NO</u>
There is no wildcard search pattern.

PATTERN = string_expression
The operand value will be searched to find the specified wildcard search pattern.

CAT-ID = <u>*YES</u> / *NO
Only relevant for filename and partial-filename.
Determines whether the catalog ID may be specified as part of a file name.

USER-ID = <u>*YES</u> / *NO
Only relevant for filename and partial-filename.
Determines whether the user ID may be specified as part of a file name.

VERSION = <u>*YES</u> / *NO
Only relevant for filename and partial-filename.
Determines whether the version designation may be specified as part of a file name.

GENERATION = <u>*YES</u> / *NO
Only relevant for filename and partial-filename.
Determines whether the generation designation may be specified as part of a file name.

WILDCARDS = <u>*NO</u> / *YES
Only relevant for filename, partial-filename, alphanum-name, composed-name and name.
Determines whether the operand value may contain wildcards or placeholder characters.
*YES may not be specified in conjunction with PATTERN = string_expression.

KEYSTAR = <u>*NO</u> / *YES
Only relevant for keyword and keyword-number.
Determines whether the operand value must contain a leading asterisk.

SEPARATORS = <u>*YES</u> / *NO
Only relevant for text.
Determines whether separators may be included.

UNDERSCORE = <u>*NO</u> / *YES
Only relevant for name and composed-name.
Determines whether the operand value may contain underscores.

ODD = <u>*YES</u> / *NO
Only relevant for x-text.
Determines whether an odd number of characters is acceptable.

CORRECTION-STATE = <u>*YES</u> / *NO / *ANY
Only relevant for product-version.
Determines whether the correction state must be specified.*
ANY: No check is made to see if the correction state was specified.

USER-INTERFACE = <u>*YES</u> / *NO / *ANY
Only relevant for product-version.
Determines whether the release state of the user interface may be specified.
*ANY: No check is made to see if the release state was specified.

ALIAS = <u>*YES</u> / *NO
Only relevant for device.
Determines whether alias names may be specified.

VOLUME-ONLY = <u>*NO</u> / *YES
Only relevant for device.
Determines whether the volume type is accepted.

WILDCARD-TYPE = <u>*SELECTOR</u> / *CONSTRUCTOR
Only relevant for filename, name, alphanum-name and structured-name.
Determines whether the specified operand value is to be interpreted as a selection string or as a construction string.

LOWER-CASE = <u>*NO</u> / *YES
Only relevant for name.
Determines whether the operand value is allowed to contain small letters.

QUOTES = <u>*OPTIONAL</u> / *MANDATORY
Only relevant for posix-filename and posix-pathname.
Determines whether the operand value is allowed to contain quotes.

TEMPORARY-FILE = <u>*YES</u> / *NO
Only relevant for filename.
Determines whether the name of a temporary file is allowed as the operand value.

SCOPE = <u>*ALL</u> / *STD-DISK
Only relevant for device.
Determines whether the name of any disk device or of a standard disk device is allowed to be specified as the operand value.

DEVICE-CLASS = <u>*DISK</u> / *TAPE / *DISK-OR-TAPE
Only relevant for device.
Determines which device class (disk and/or tape device) the specified device may belong to.

EXCEPT-DISKS = <u>*NONE</u> / list-poss(50): string_expression
Only relevant for device.
Determines which disk devices from the list of available devices must not be specified.

EXCEPT-TAPES = <u>*NONE</u> / list-poss(50): string_expression
Only relevant for device.
Determines which tape devices from the list of available devices must not be specified.

Result

TRUE
The specified operand value satisfies the check criteria.

FALSE
The specified operand value does not satisfy the check criteria.

Error messages

```
SDP0099   NO MORE VIRTUAL MEMORY AVAILABLE AT THIS MOMENT

SDP0454   INVALID PARAMETER : '(&00)'

SDP0459   PARAMETER ERROR OR INVALID PARAMETERS COMBINATION. ADDITIONAL
          INFORMATION:  ‘(&00)‘
```

Example

```
/A = CHECK-DATA-TYPE(':CAT:$USER.MYFILE', DATA-TYPE=*FILENAME)
/SHOW-VARIABLE A
A = TRUE

/A = CHECK-DATA-TYPE(':CAT:$USER.MYFILE', DATA-TYPE=*FILENAME, CAT-ID=*NO)
/SHOW-VARIABLE A
A = FALSE

/A = CHECK-DATA-TYPE('PAR', DATA-TYPE=*KEYWORD,VALUE='PARAMETERS')
/SHOW-VARIABLE A
A = TRUE

/A = CHECK-DATA-TYPE('PAR', DATA-TYPE=*KEYWORD, VALUE='PARAMETERS',-
/KEYSTAR=*YES)
/SHOW-VARIABLE A
A =FALSE
```

COUNTER( ) Count function calls

Domain: **Task-specific counter**

The COUNTER( ) function keeps track of the number of times COUNTER( ) is called in the current task. The counter is incremented by 1 each time COUNTER( ) is called.

Format

COUNTER( )

Result type

INTEGER (<integer 1 .. 2147483647>)

Input parameters

None

Result

Maximum ten-digit number

Error message

```
SDP0304   OVERFLOW; NUMBER OUT OF RANGE
```

Example

```
/FOR I = *COUNTER(FROM = 2, TO = 28, INCREMENT = 2)
/   A = COUNTER()
/END-FOR

/SHOW-VARIABLE A
A = 14
```

CURRENT-TYPE( ) Request variable type

Domain: **Variable access** (variable name)

The CURRENT-TYPE( ) function returns the current type of the value of a simple variable (this must not be confused with the current type of a variable declaration, which is returned by the VARIABLE-ATTRIBUTE( ) function). If the variable type has not yet been defined (TYPE = *ANY), or if CURRENT-TYPE( ) is applied to a complex variable, *NONE is returned as the result.

Format

CURRENT-TYPE( )
CURR-TYPE( )

VARIABLE-NAME = string_expression

Result type

STRING

Input parameters

VARIABLE-NAME = string_expression
Designates the variable whose type is requested. The variable name must be enclosed in apostrophes if it is specified directly, i.e. as a literal (see also the example on the next page and the last example in the description of IS-DECLARED( )).

Result

**BOOLEAN*
“string_expression” designates a variable which contains a value of the type BOOLEAN (the variable must have been declared with the type BOOLEAN or *ANY).

**INTEGER*
“string_expression” designates a variable which contains a value of the type INTEGER (the variable must have been declared with the type INTEGER or *ANY).

**NONE*
The variable “string_expression” does not yet have a defined variable type, or “string_expression” designates a complex variable.

*STRING
“string_expression” designates a variable which contains a value of the type STRING (the variable must have been declared with the type STRING or *ANY).

Error messages

SDP1007	NO VARIABLE DECLARED
SDP1101	SYNTAX ERROR IN VARIABLE NAME

Example

```
/DECLARE-VARIABLE A (TYPE = *ANY)
/B = CURRENT-TYPE(VARIABLE-NAME = 'A')
/SHOW-VAR B
B = *NONE
```

```
/A = 123
/B = CURRENT-TYPE(VARIABLE-NAME = 'A')
/SHOW-VAR B
B = *INTEGER
```

DATE( ) Output date

Domain: **Environment information** (calendar)

The DATE( ) function determines the current date and returns it in the specified format.

Format

```
DATE( )

 FORMAT = *ISO / *AMERICAN / *GERMAN
,MODE = *LOCAL-TIME / *UNIVERSAL-TIME
```

Result type

STRING (<string 10..13>)

Input parameters

FORMAT = *ISO / *AMERICAN / *GERMAN
Specifies the format in which the date is output.

MODE = *LOCAL-TIME / *UNIVERSAL-TIME
Determines if the date is output in the local time (LOCAL-TIME) or in universal time (UNIVERSAL-TIME).
See also the GTIME macro in the “Executive Macros” manual [7] for more information on LOCAL-TIME (LT) and UNIVERSAL-TIME (UTC).

Result

Input parameter FORMAT =	**Date format <string 10..13>**
*AMERICAN	mm/dd/yyiii
*ISO	yyyy-mm-ddiii
*GERMAN	dd.mm.yyyy

iii	Day in current year (001 .. 366)
yy	Two-digit year number
yyyy	Complete year number
mm	Two-digit month number (01 .. 12)
dd	Day in current month (01 .. 31)

Error messages

No error messages

Example

```
/G = DATE(FORMAT = *GERMAN)
/SHOW-VARIABLE G
G = 16.04.2007
```

```
/A = DATE(FORMAT = *AMERICAN)
/SHOW-VARIABLE A
A = 04/16/07106
```

```
/I = DATE( )
/SHOW-VARIABLE I
I = 2007-04-16106
```

April 16th is the 106th day of the year 2007, which is why the representation of the date in the ISO format and in the American format contains the suffix 106.

DATE-VALUE( ) Output particular date

Domain: **Environment information** (calendar)

The DATE-VALUE( ) function outputs the date which is a specified number of days from the base date (default value for this is the start of the 20th century (1900-01-01)).

Format

```
DATE-VALUE( )

 NUMBER-OF-DAYS = arithm_expression
,BASE = *STD / *TODAY / string_expression
,FORMAT = *ISO / *AMERICAN / *GERMAN
```

Result type

STRING (<string 10..13>)

Input parameters

NUMBER-OF-DAYS = arithm_expression
Number of days from the base date.

BASE =
Designates the base date.
The format of this BASE value is independent of the value of the FORMAT operand.
It may be in the *ISO, *GERMAN or *AMERICAN format, with the day or month in one or two digit form (a leading zero is not required for the first 9 days or months), and with the year in two or four digit form (omitting the number of days in the current year). If the year is specified in two-digit form (these being the last two digits), the first two digits will be determined in the same way as for SDF data type ’date’; e.g

Input year	**First two digits**
00..59	19
60..99	20

BASE = <u>*STD</u>
The base date is the start of the 20th century (1900-01-01).

BASE = *TODAY
The base date is the current date.

BASE = string_expression
Specifies the base date.
The value is a date after 1582-10-15.

FORMAT = *ISO / *AMERICAN / *GERMAN
Defines the format in which the date is output.

Result

Input parameter FORMAT =	Date format <string 10..13>
*AMERICAN	mm/dd/yyiii
*ISO	yyyy-mm-ddiii
*GERMAN	dd.mm.yyyy

iii	Day in current year (001 .. 366)
yy	Two-digit year number
yyyy	Complete year number
mm	Two-digit month number (01 .. 12)
dd	Day in current month (01 .. 31)

Error message

```
SDP0452   INVALID DATE
```

Example

```
/A = DATE-VALUE(NUMBER-OF-DAYS = 23008, FORMAT = *ISO)
/SHOW-VARIABLE A
A = 1962-12-30364
```

```
/A = DATE-VALUE(NUMBER-OF-DAYS = 23008, FORMAT = *AMERICAN)
/SHOW-VARIABLE A
A = 12/30/62364
```

```
/A = DATE-VALUE(NUMBER-OF-DAYS = 23008, FORMAT = *GERMAN)
/SHOW-VARIABLE A
A = 30.12.1962
```

```
/TOMORROW = DATE-VALUE(NUMBER-OF-DAYS = 1, BASE = *TODAY)
/TODAY = DATE( )
/SHOW-VARIABLE (TODAY,TOMORROW)
TODAY = 2001-08-09221
TOMORROW = 2001-08-10222
```

DAY( ) Output day of the week

Domain: **Environment information** (calendar)

The DAY( ) function supplies the name of the current day of the week in the specified language, but only in abbreviated form.

Format

DAY( )
LANGUAGE = <u>*ENGLISH</u> / *GERMAN / *STD

Result type

STRING (<string 2..3>)

Input parameters

LANGUAGE = <u>*ENGLISH</u> / *GERMAN / *STD
Defines the language in which the name of the current day of the week is to be returned. When *STD is specified, the output appears in the language set for the task.

Result

Two- or three-letter abbreviation for the day of the week, depending on the specified language.

Input parameters	**Result**
*ENGLISH	SUN / MON / TUE / WED / THU / FRI / SAT
*GERMAN	SO / MO / DI / MI / DO / FR / SA

Error messages

No error messages

Example

```
/G = DAY(LANGUAGE = *GERMAN)
/SHOW-VARIABLE G
G = MO/E = DAY( )
/SHOW-VARIABLE E
E = MON
```

ELAPSED-DAYS( ) Output number of days difference

Domain: **Environment information** (calendar)

The ELAPSED-DAYS( ) function outputs the number of days difference between two specified dates.

Format

```
ELAPSED-DAYS( )

 DATE = string_expression
,BASE = *STD / *TODAY / string_expression
```

Result type

INTEGER (<integer -3074323 .. 3074323>)

Input parameters

DATE = string_expression
Designates the end date.
The only permitted formats are *ISO, *AMERICAN and *GERMAN. The value must be greater than or equal to 1582-10-15.

BASE =
Designates the base date.
The only permitted formats are *ISO, *AMERICAN and *GERMAN.

BASE = *STD
The base date is the start of the 20th century (1900-01-01).

BASE = *TODAY
The base date is the current date.

BASE = string_expression
Specifies the base date.
The value is the date 1582-10-15 or later.

Notes

When BASE and DATE are specified, the following rules must be observed:

- The format can only be *ISO, *GERMAN or *AMERICAN.
- A leading zero is not required for the first 9 days or months. The year can be abbreviated to two-digit form, in which case the first two digits will be determined in the same way as for SDF data type <date with-compl>; e.g.

Input year	First two digits
00..59	19
60..99	20

Result

An integer number.

Error message

```
SDP0452   INVALID DATE
```

Example

```
/A = ELAPSED-DAYS (DATE=‘1963-12-30‘)
/SHOW-VARIABLE A
A = 23373
```

```
/A = ELAPSED-DAYS (DATE='12/30/1963', BASE = '12/30/1900')
/SHOW-VARIABLE A
A = 23010
```

```
/A = ELAPSED-DAYS (DATE='30.12.1963')
/SHOW-VARIABLE A
A = 23373
```

```
/DIFF = ELAPSED-DAYS (DATE='2001-08-23',BASE='2001-04-01')
/SHOW-VARIABLE DIFF
DIFF = 144
```

```
/DIFF = ELAPSED-DAYS (DATE='2001-08-23',BASE='2001-10-31')
/SHOW-VARIABLE DIFF
DIFF = -69
```

EXPLICIT-CALL( ) Output explicit command call

Domain: **Procedure information**

The EXPILICIT-CALL( ) function outputs the type of call to a procedure. TRUE means that the call was explicit (i.e. by a CALL-PROCEDURE, INCLUDE-PROCEDURE) and FALSE means an implicit call (e.g. when the call is made by a command in a procedure; see also the SDF-A statement //ADD-CMD ... IMPLEMENTOR=*PROCEDURE in the “SDF-A” manual [16]).

Format

EXPLICIT-CALL( )

Result type

BOOLEAN

Input parameters

None

Result

TRUE
The specified call is an explicit one, i.e. it was issued by CALL-PROCEDURE, INCLUDE-PROCEDURE (or an alias name or redefined name for it).

FALSE
The specified call is not an explicit one.

Error messages

No error messages

Example

```
/SET-PROCEDURE-OPTIONS “Procedure MYPROC”
/    WRITE-TEXT 'Explicit call: &(EXPLICIT-CALL)'
/EXIT-PROCEDURE

/CALL-PROCEDURE MYPROC
Explicit call: TRUE

/INCLUDE-PROCEDURE MYPROC
Explicit call: TRUE

/DO MYPROC
Explicit call: TRUE

/MY-COMMAND MYPROC   “User command with implementation of procedure MYPROC”
Explicit call: FALSE
```

EXTEND-SDF-LIST( ) Append list element

Domain: **String processing**

The EXTEND-SDF-LIST( ) function appends a new element to an SDF list. This new element may itself be an SDF list.

Format

EXTEND-SDF-LIST( )
LIST = string_expression ,ELEMENT = string_expression ,POSITION = <u>*LAST</u> / *FIRST / arithm_expression

Result type

STRING

Input parameters

LIST = string_expression
Designates an SDF list. An empty list must be specified as ‘()‘. The validity of the input is checked internally by IS-SDF-LIST.

ELEMENT = string_expression
Designates the element to be appended.

POSITION =
Specifies where the element is to be appended.

POSITION = <u>*LAST</u>
The element is appended at the end of the list.

POSITION = *FIRST
The element is appended before the list.

POSITION = arithm_expression
The element is inserted at the specified position.
If the specified position is outside the permissible range, *LAST is assumed.

Result

Expression as an extended string

Error messages

```
SDP0447   THE GIVEN STRING IS NO SDF-LIST

SDP0481   VALUE OF OPERAND 'POSITION' MUST BE GREATER THAN ZERO
```

Example

```
A=EXTEND-SDF-LIST(LIST='(val1,val2)',ELEMENT='val3',POSITION=*last)
/SHOW-VARIABLE A
A = (val1,val2,val3)

/A=EXTEND-SDF-LIST(LIST=A,ELEMENT='val0',POSITION=*first)
/SHOW-VARIABLE A
A = (val0,val1,val2,val3)

/A=EXTEND-SDF-LIST(LIST=A,ELEMENT='(val4,val5)',POSITION=*last)
/SHOW-VARIABLE A
A = (val0,val1,val2,val3,(val4,val5))
```

EXTRACT-FIELD( ) Extract field

Domain: **String processing**

The EXTRACT-FIELD( ) function extracts a field from an input string.

Format

EXTRACT-FIELD( )
STRING = string_expression ,FIELD-NUMBER = arithm_expression ,FIELD-SEPARATOR = *ANY-BLANKS / string_expression

Result type

STRING

Input parameters

STRING = string_expression
Designates an input string.

FIELD-NUMBER = arithm_expression
Designates the field number.

FIELD-SEPARATOR =
Specifies the separator. Separators are not part of the extracted field.

FIELD-SEPARATOR = *ANY-BLANKS
The default value for the separator is one or more blanks.
(The specification of '␣␣*' is supported as compatible.)

FIELD-SEPARATOR = string_expression
string_expression is the separator.
Here however, string_expression must be a simple regular expression (for further details see “POSIX Commands” [18]).

Result

The extracted field, in the form of a string.

Error messages

SDP0472	NULL BYTE (X'00') NOT ALLOWED IN STRING AND FIELD SEPARATOR OPERANDS
SDP0474	SYNTAX ERROR IN REGULAR EXPRESSION FOR OPERAND FIELD SEPARATOR
SDP0485	VALUE OF OPERAND 'FIELD NUMBER' MUST BE GREATER THAN ZERO

Examples

Example 1

```
/DECLARE-VARIABLE mylist(TYPE=*STRING),MULTIPLE-ELEMENT=*LIST
/mylist = 'Pencil 100',WRITE-MODE=*EXTEND
/mylist = 'Table 5',WRITE-MODE=*EXTEND
/mylist = 'Lamp 20',WRITE-MODE=*EXTEND
/mylist = 'Paper 75',WRITE-MODE=*EXTEND
/mylist = 'Diskette 1000',WRITE-MODE=*EXTEND
/mylist = 'Envelope 1500',WRITE-MODE=*EXTEND

/FOR x = *LIST(mylist)
/  article = EXTRACT-FIELD(STRING=x,FIELD-NUMBER=1)
/  quantity = EXTRACT-FIELD(STRING=x,FIELD-NUMBER=2)
/  WRITE-TEXT '&quantity of &article are available'
/END-FOR
```

Output:

```
100 of Pencil are available
5 of Table are available
20 of Lamp are available
75 of Paper are available
1000 of Diskette are available
1500 of Envelope are available
```

Example 2

```
/A=EXTRACT-FIELD(STRING='field1,field3,field4',FIELD-NUMBER=3,FIELD-
SEPARATOR=',')
/SHOW-VARIABLE A
A = field4
```

FILL( ) Fill string

Domain: **String processing**

The FILL( ) function fills the string specified in the function call with zeros up to the specified length and in the specified direction. This fill character can be defined in the function call.

Format

FILL( )
STRING = string_expression
,LENGTH = arithm_expression
,SIDE = *RIGHT / *LEFT
,FILL-BYTE = C'␣' / character

Result type

STRING

Input parameters

STRING = string_expression
Designates the input string which is to be filled to the length defined with the parameter LENGTH = .

LENGTH = arithm_expression
Positive number which determines the length of the input string.
If the value for LENGTH is greater than the current length of the input string (see STRING parameter) and lies within the valid range of values for string lengths, the input string is filled with the character defined in FILL-BYTE = .
If “arithm_expression” is not within the valid range of values, an error message is output. If LENGTH is shorter than the actual length of STRING, STRING is returned unchanged as the result.

SIDE =
Determines the direction of the input string in the result string, i.e. the direction in which the fill characters are appended; SIDE = is ignored if the input string is returned unchanged.

SIDE = *RIGHT
Appends the fill characters to the right, i.e. after the last character in the input string.

SIDE = *LEFT
Appends the fill characters to the left, i.e. before the first character in the input string.

FILL-BYTE =
Determines which character is used to fill the input string. FILL-BYTE = is ignored if the input string is not lengthened.

FILL-BYTE = C'␣'
Fills the input string with blanks (to the left or right).

FILL BYTE = character
Fills the input string with the character specified in this position (to the left or right). “character” can be any character. If more than one character is specified, only the first character is used as the fill character.
If a null string (C”) is specified instead of a character, an error message is output.

Result

String with the length of LENGTH = number

Error messages

```
SDP0431   ERROR '(&00)' IN BUILTIN FUNCTION '(&01)'

SDP0436   GIVEN LENGTH NOT BETWEEN ZERO AND MAXIMUM POSSIBLE STRING LENGTH

SDP0437   LENGTH OF PARAMETER 'FILL-BYTE' EQUAL TO ZERO
```

Example

```
/A = 'ABCDE'
/SHOW-VARIABLE A
A = ABCDE

/A = FILL(STRING = A, LENGTH = 8, FILL-BYTE = C'.')
/SHOW-VARIABLE A
A = ABCDE...

/A = FILL(STRING = A, LENGTH = 12, SIDE=*LEFT, FILL-BYTE = C'.')
/SHOW-VARIABLE A
A =....ABCDE...
```

FIRST-VARIABLE-NAME( ) Request variable element name

Domain: **Variable access (variable name)**

The FIRST-VARIABLE-NAME( ) function can be used to analyze the format of complex variables, primarily in combination with the NEXT-VARIABLE-NAME( ) function. FIRST-VARIABLE-NAME( ) can be applied to all aggregates and lists. FIRST-VARIABLE-NAME( ) begins with the specified variable or variable element name and then supplies the name of the first variable. If there is no lower level, FIRST-VARIABLE-NAME( ) returns *END.

Format

FIRST-VARIABLE-NAME( )
FIRST-VAR-NAME( )

VARIABLE-NAME = string_expression

Result type

STRING (<composed-name 1..255> or '*END')

Input parameters

VARIABLE-NAME = string_expression
Designates a complex variable or a variable element.
If the complex variable is a list, the name of the first list element (list#1) is supplied.
If the variable element is a list element, *END or the name of the appropriate element is output. The variable name must be enclosed in apostrophes if it is specified directly, i.e. as a literal (see the example on the next page).

Result

elementname, if "string_expression" designates a complex variable.

**END*
"string_expression" designates a variable element on the lowest level which is itself no longer a complex variable; this means that there is no lower level.

Error message

```
SDP1101   SYNTAX ERROR IN VARIABLE NAME
```

Example

The variable FSTAT is declared as an array with dynamic structures as its elements.

```
/DECLARE-VARIABLE FSTAT(TYPE=*STRUCTURE(*DYNAMIC)),MULTIPLE-ELEMENTS=*ARRAY
```

The elements of these dynamic structures are initialized as follows:

```
/FSTAT#1.F-NAME = 'FILE.A'
/FSTAT#1.F-SIZE = 0000003
/FSTAT#2.F-NAME = 'FILE.B'
/FSTAT#2.F-SIZE = 000006
```

FIRST-VARIABLE-NAME can be used to analyze the variable:

```
/A = FIRST-VARIABLE-NAME('FSTAT')
/SHOW-VARIABLE A
A = FSTAT#1

/B = FIRST-VARIABLE-NAME(A)
/SHOW-VARIABLE B
B = FSTAT#1.FNAME

/C = FIRST-VARIABLE-NAME(B)
/SHOW-VARIABLE C
C = *END
```

FROM-C-LITERAL( ) Convert C literal

Domain: **Conversion functions**

The FROM-C-LITERAL( ) function converts a C literal to the corresponding string value. The leading C and the single quotes at the beginning and end of the literal are deleted.

FROM-C-LITERAL( ) is the reverse function of TO-C-LITERAL( ).

Format

FROM-C-LITERAL( )
FROM-C-LIT( )

STRING = string_expression

Result type

STRING

Input parameters

STRING = string_expression
Designates a C literal; the identifying C at the beginning of the literal is removed along with the single quotes at the beginning and end of the literal. Double quotes within the string are reduced to single quotes.

Result

String

Error message

```
SDP0433   GIVEN STRING NOT A C-LITERAL
```

Example

```
/B = FROM-C-LITERAL(STRING = 'C''ABC''')
/SHOW-VARIABLE B
B = ABC

/A = 'ABC'
/B = FROM-C-LITERAL(STRING = A)
```

```
SDP0433  GIVEN STRING NOT A C-LITERAL
SDP0431  ERROR 'SDP0433' IN BUILTIN FUNCTION 'FROM-C-LITERAL'
SDP0239  ERROR DURING EVALUATION OF RIGHT SIDE OF ASSIGNMENT
```

ABC is not a C literal and can thus not be converted.

FROM-X-LITERAL( ) Convert X literal

Domain: **Conversion functions**

The FROM-X-LITERAL( ) function converts an X literal to the corresponding string value.

FROM-X-LITERAL( ) is the reverse function of TO-X-LITERAL( ).

Format

FROM-X-LITERAL( )
FROM-X-LIT( )

STRING = string_expression

Result type

STRING

Input parameters

STRING = string_expression
Designates an X literal.

Result

String

Error message

```
SDP0434   GIVEN STRING NOT A X -LITERAL
```

Example

```
/F = C'X''C1'''
/B = FROM-X-LITERAL(F)
/SHOW-VARIABLE B
B = A
```

HASH-STRING( ) Encrypt expression as string

Domain: **Conversion functions**

The HASH-STRING( ) function converts a string expression to a string with a binary content with any required length (e.g. a password). The algorithm which is used has a high probability of returning different output values for different input strings.

Format

HASH-STRING( )
STRING = string_expression ,LENGTH = <u>4</u> / arithm_expression

Result type

STRING

Input parameters

STRING = string_expression_1..256
Designates the input string.

LENGTH = <u>4</u> / arithm_expression_1..256
Designates the length of the output string.

Result

String

Error messages

```
SDP0455   INVALID LENGTH OF INPUT STRING (ALLOWED : 1..256)

SDP0456   LENGTH PARAMETER IS OUT OF RANGE (ALLOWED : 1..256)
```

Example

```
/A = HASH-STRING(STRING='AB',LENGTH=2)
/SHOW-VAR A, INFORMATION = *PARAMETERS(VALUE=*X-LITERAL)

A = X'8C5E'
```

HASH-VALUE( ) Encrypt expression as integer value

Domain: **Conversion functions**

The HASH-VALUE( ) function converts a string expression to an integer value; the algorithm which is used to do this has a high probability of returning different output values for different input strings.

Format

HASH-VALUE( )
STRING = string_expression

Result type

INTEGER (<integer -2^{31}..2^{31}-1>)

Input parameters

STRING = string_expression_1..256
Designates the input string.

Result

Integer

Error message

```
SDP0455   INVALID LENGTH OF INPUT STRING (ALLOWED : 1..256)
```

Example

```
/RANDOM = HASH-VALUE(TIME( )) MOD 60           "BETWEEN 0 AND 59"
/HOUR = INTEGER(SUBSTRING(TIME(),START=1,LENGTH=2))
/MINUTE = INTEGER(SUBSTRING(TIME( ),START=4,LENGTH=2))
/MINUTE = MINUTE + RANDOM
/IF (MINUTE > 59)
/      MINUTE = MINUTE - 60
/      HOUR = HOUR + 1
/       IF (HOUR > 23)
/            HOUR = 0
/       END-IF
/END-IF
/WRITE-TEXT 'In &RANDOM. minute(s) it will be
    &HOUR.:&(FILL(STRING=STRING(MINUTE),LENGTH=2,SIDE=*LEFT,FILL-BYTE='0'))'
```

```
In 32 minute(s) it will be    10:00
```

HOME-CAT-ID( ) Request catalog ID of home pubset

Domain: **Environment information**

The HOME-CAT-ID( ) function supplies the catalog ID of the home pubset. The catalog ID is assigned by the system administrator and can be up to four characters long. The home pubset of the running system is the pubset from which the system was loaded (see the manual entitled “Introductory Guide to Systems Support” [8] for more information on the home pubset).

Format

HOME-CAT-ID( )

Result type

STRING (<string 1..4>)

Input parameters

None

Result

Catalog ID of the home pubset in the form of a string.

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

```
/A = HOME-CAT-ID()
/SHOW-VARIABLE A
A = 10SB
```

The home pubset thus has the catalog ID 10SB.

HOST( ) Request host name

Domain: **Environment information**

The HOST( ) function supplies the internal name of the host on which the function is called. The system administrator defines this internal name when DCM is started.

Format

HOST( )

Result type

STRING (<string 1..8>)

Input parameters

None

Result

Name of the host in the form of a string.

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

```
/A = HOST()
/SHOW-VARIABLE A
A = D016ZE04
```

D016ZE04 is the name of the host.

INDEX( ) Search for string

Domain: **String functions**

The INDEX( ) function indicates the position of a search string within the overall string. The overall string can be searched from left to right or from right to left; the result value always relates to the beginning of the overall string.

Format

INDEX( )
STRING = string_expression$_1$,PATTERN = string_expression$_2$,DIRECTION = <u>*FORWARD</u> / *REVERSE ,BEGIN-COLUMN = <u>1</u> / arithm_expression ,END-COLUMN = <u>*LAST</u> / arithm_expression

Result type

INTEGER

Input parameters

STRING = string_expression$_1$
Overall string to be searched.

PATTERN = string_expression$_2$
Search string to be located in the overall string.

DIRECTION =
Search direction

DIRECTION = <u>*FORWARD</u>
The overall string is searched in a forward direction, i.e. from left to right.

DIRECTION = *REVERSE
The overall string is searched in reverse, i.e. from right to left.

BEGIN-COLUMN =
Seen from the start of the overall string, the search operation is restricted to a certain range of columns. The first character in the overall string from which the search for the search string begins is specified.
The string being searched is empty if the overall string contains fewer characters than specified for BEGIN-COLUMN.

BEGIN-COLUMN = 1
The search starts from column 1, i.e. the overall string is searched from the beginning.

BEGIN-COLUMN = arithm_expression
The overall string is searched for the search string from the specified column or from this character

END-COLUMN =
Seen from the end of the overall string, the search operation is restricted to a certain range of columns. The last character in the overall string which is included in the search is specified. All subsequent characters are ignored.

END-COLUMN = *LAST
The overall string is searched to the end.

END-COLUMN = arithm_expression
The overall string is searched for the search string up to the specified column or up to and including this character.

Result

Integer
Initial position of the search string in the overall string.
If there are multiple search strings in the overall string, “integer” indicates the first occurrence of the search string in a search from left to right and the last occurrence in a search from right to left.

0
The search string is longer than the overall string, or the search string is not contained in the overall string.

Error messages

SDP0413	ILLEGAL LENGTH
SDP0493	Value of operands BEGIN-INDEX, END-INDEX, BEGIN-COLUMN and END-COLUMN must be greater than zero
SDP0498	BEGIN-COLUMN must not be greater than END-COLUMN

Example 1

```
/A = INDEX(STRING = 'ABCDE', PATTERN = 'C')
/SHOW-VARIABLE A
A = 3

/B = INDEX(STRING = 'ABCDEABC', PATTERN = 'AB')
/SHOW-VARIABLE B
B = 1

/C = INDEX(STRING = 'ABCDEABC', PATTERN = 'AB', DIRECTION = *REVERSE)
/SHOW-VARIABLE C
C = 6
```

Example 2

```
/STRING = '1080:0:0:0:8:800:200C:417A'
/
/WRITE-TEXT '- FROM LEFT TO RIGHT -'
/START = 1
/REPEAT
/   WRITE-TEXT '&(START) => &(SUBSTR( STRING, START ))'
/   START = INDEX( STRING, ':', *FORWARD, START, *LAST ) + 1
/UNTIL ( START == 1 )
/              &* AT THE LAST LOOP ITERATION
/              &* INDEX DOES NOT FIND THE ':'
/              &* AND RETURNED 0
/              &* 1 IS ADDED FROM THIS RETURNED VALUE.
/
/WRITE-TEXT '- FROM RIGHT TO LEFT -'
/END = LENGTH( STRING )
/REPEAT
/   WRITE-TEXT '&(END) => &(SUBSTR( STRING, 1, END ))'
/   END = INDEX( STRING, ':', *REVERSE, 1, END ) - 1
/UNTIL ( END <= 0 )
/              &* AT THE LAST LOOP ITERATION
/              &* INDEX DOES NOT FIND THE ':'
/              &* AND RETURNED 0.
/              &* 1 IS SUBTRACTED FROM THIS RETURNED VALUE.
/
/WRITE-TEXT '- SURROUNDING CUT -'
/START = 1
/END = LENGTH( STRING )
/REPEAT
/   TEXT = '&(START):&(END) => ' // -
/          SUBSTR( STRING, START, END - START + 1 )
/   WRITE-TEXT '&(TEXT)'
/   START = INDEX( STRING, ':', *FORWARD, START, END ) + 1
/   END = INDEX( STRING, ':', *REVERSE, START, END ) - 1
/UNTIL ( START > END )
```

This example shows how a string can be searched step-by-step for separator characters (here colons) and be reduced by the substring which has already been searched. The search and reduction, which is performed in different directions (left to right, right to left and both ways) supplies the following output:

```
- FROM LEFT TO RIGHT -
1 => 1080:0:0:8:800:200C:417A
6 => 0:0:0:8:800:200C:417A
8 => 0:0:8:800:200C:417A
10 => 0:8:800:200C:417A
12 => 8:800:200C:417A
14 => 800:200C:417A
18 => 200C:417A
23 => 417A
- FROM RIGHT TO LEFT -
26 => 1080:0:0:0:8:800:200C:417A
21 => 1080:0:0:0:8:800:200C
16 => 1080:0:0:0:8:800
12 => 1080:0:0:0:8
10 => 1080:0:0:0
8 => 1080:0:0
6 => 1080:0
4 => 1080
```

```
- SURROUNDING CUT -
1:26 => 1080:0:0:0:8:800:200C:417A
6:21 => 0:0:0:8:800:200C
8:16 => 0:0:8:800
10:12 => 0:8
```

INSTALLATION-PATH( ) Output path name

Domain: **Environment information**

The INSTALLATION-PATH( ) specifies the path name assigned from the SCI for the logical name of a file (installation item) that belongs to a specific product version.

The association between the logical name and the path name of a file is only available when the file is part of a product that was installed using IMON. The assignment can also be entered in the SCI by systems support with the SET-INSTALLATION-PATH command. See the “IMON” manual [12] for more detailed information.

An expression must be specified in the DEFAULT-PATH-NAME operand that is returned as a replacement when no assigned path name exists (the product is not registered in the SCI or there is no file for the logical name specified).

Format

INSTALLATION-PATH( )
LOGICAL-ID = string_expression ,INSTALLATION-UNIT = string_expression ,VERSION = <u>*STD</u> / string_expression ,DEFAULT-PATH-NAME = string_expression

Result type

STRING

Input parameters

LOGICAL-ID = string_expression
Designates a file ID (e.g. designates the logical name of the file (installation item) whose path name is to be output (e.g. SYSPRG).

INSTALLATION-UNIT = string_expression
Designates the product name (name of the installation unit).

VERSION = <u>*STD</u> / string_expression
Designates the product version (up to 8 characters).
A version can be specified explicitly in the format *[V][m]m.naso* (see also the SDF data type composed-name). If the specified version is not registered in the SCI, then the function call is cancelled without returning a result (not even a replacement string as the result).

DEFAULT-PATH-NAME = string_expression
Specifies the replacement string (e.g. a path name that surely exists) to be output when no path name assignment is available (i.e. when the product or installation item is not registered in the SCI).

Result

A string, conforming to the rules for the SDF data type <filename 1..54>.

Error messages

```
SDP0469   INVALID PARAMETER ‘(&00)‘ SPECIFIED

SDP0470   INTERNAL ERROR RETURNED BY GETINSP/GETINSV INTERFACE. RETURN CODE
          ‘(&00)‘ RECEIVED

SDP0489   WARNING: INSTALLATION-UNIT '(&00)' NOT FOUND IN IMON SOFTWARE
          INVENTORY. DEFAULT VALUE ASSUMED

SDP0490   INSTALLATION-UNIT '(&00)', VERSION '(&01)' NOT FOUND

SDP0491   WARNING: LOGICAL-ID '(&00)' NOT FOUND IN INSTALLATION-UNIT '(&01)'
          , VERSION '(&02)'. DEFAULT VALUE ASSUMED
```

Examples

```
/A = INSTALLATION-PATH(LOGICAL-ID='SYSLNK',INSTALLATION-UNIT='EDT',
DEFAULT-PATH-NAME='*** No path name present! ***')
/SHOW-VARIABLE A
A = :2OSH:$TSOS.SYSLNK.EDT.166
```

The path name of the load library of the EDT product is output.

```
/A = INSTALLATION-PATH(LOGICAL-ID='SYSRME.D',INSTALLATION-UNIT='EDT',
DEFAULT-PATH-NAME='*** No readme file present! ***')
%  SDP0491 Warning: Logical-id 'SYSRME.D' not found in Installation-Unit
'EDT' version '*STD'. Default value assumed
/SHOW-VARIABLE A
A = *** No readme file present! ***
```

The replacement string as defined in DEFAULT-PATH-NAME is output as no readme file for the current EDT version exists.

```
/B = INSTALLATION-PATH(LOGICAL-ID='SYSLNK',INSTALLATION-UNIT='EDT',
VERSION='16.0',DEFAULT-PATH-NAME='*** no path name present! ***')
%  SDP0490 Installation-Unit 'EDT' version '16.0' not found
%  SDP0431 ERROR 'SDP0490' IN BUILTIN FUNCTION 'INSTALLATION-PATH'
%  SDP0239 ERROR DURING EVALUATION OF RIGHT SIDE OF ASSIGNMENT
/SHOW-VARIABLE B
%  SDP1008 VARIABLE/LAYOUT 'B' DOES NOT EXIST
%  SDP0234 OPERAND 'NAME' INVALID
```

The function call is aborted. The product EDT is certainly installed (see above), but the explicitly specified version V16.0 does not exist. No value is assigned to variable B (and it is not implicitly declared) as no value is returned.

```
/C = INSTALLATION-PATH(LOGICAL-ID='SYSPRC',INSTALLATION-UNIT='USER-TOOLS',
DEFAULT-PATH-NAME='$RZTOOLS.SYSPRC.USER-TOOLS.010')
/SHOW-VARIABLE C
C = $RZTOOLS.SYSPRC.USER-TOOLS.010
```

The replacement string as defined in DEFAULT-PATH-NAME is output (in this case a path name), as there is no product with the name USER-TOOLS registered in the SCI.

INTEGER( ) Convert expression to integer

Domain: **Conversion functions**

The INTEGER( ) function converts any expression to the data type INTEGER. In doing this, STRING expressions are converted according to the rules of implicit conversion. The following applies to BOOLEAN expressions: TRUE is converted to the value 1, FALSE to 0.

The IS-INTEGER function can be used to first check whether a STRING expression can be converted.

Format

INTEGER( )
INT( )

EXPRESSION = expression

Result type

INTEGER

Input parameters

EXPRESSION = expression
“expression” is a STRING, INTEGER or BOOLEAN expression.

Result

Number of type INTEGER

Error message

```
SDP0415   SYNTAX ERROR: INTEGER EXPECTED IN STRING. CONVERSION NOT POSSIBLE
```

Example

The variables A, B, C and D are initialized:

```
/A = '4'
/B = 5
/C = 30
/D = TRUE    “Type: BOOLEAN”

/AINT = INTEGER(EXPRESSION = A)
/SHOW-VARIABLE AINT
AINT = 4

BINT = INTEGER(EXPRESSION = B + C)
/SHOW-VARIABLE BINT
BINT = 35

/CINT = INTEGER(EXPRESSION = D)
/SHOW-VARIABLE CINT
CINT = 1
```

INTEGER-TO-CHARACTER( ) Convert integer to character

Domain: **Conversion functions**

The INTEGER-TO-CHARACTER( ) function converts an integer into a character (C string).

The number is interpreted as the integer value of the EBCDI code for a character, and this character is returned.

INTEGER-TO-CHARACTER( ) is the reverse function of CHARACTER-TO-INTEGER( ).

Format

INTEGER-TO-CHARACTER( )
INT-TO-CHAR( )

INTEGER = arithm_expression

Result type

STRING (<string 1..1>)

Input parameters

INTEGER = arithm_expression
Designates the integer to be converted, where 0 ≤ integer ≤ 255 applies.

Result

Character in EBCDI code in the form of a string.

Error message

```
SDP0416   NUMBER OUT OF RANGE
```

Example

```
/B = INTEGER-TO-CHARACTER(INTEGER = 129)
/SHOW-VARIABLE B
B = a

/C = INTEGER-TO-CHARACTER(INTEGER = 193 + 16)
/SHOW-VARIABLE C
 C = J
```

INTEGER-TO-X-LITERAL( ) Convert integer to X literal

Domain: **Conversion functions**

The INTEGER-TO-X-LITERAL( ) function converts an integer to an X literal containing the 4-byte long coding.

INTEGER-TO-X-LITERAL( ) is the inverse function to X-LITERAL-TO-INTEGER( ).

Format

```
INTEGER-TO-X-LITERAL( )
INT-TO-X-LIT( )
-----------------------------
 STRING = string_expression
```

Result type

STRING

Input parameters

STRING = string_expression
Specifies the string up to 4 bytes long which is to be converted.

Result

String containing an X literal.

Error message

No error messages

Example

```
/DECLARE-VARIABLE A( TYPE= *STRING )
/DECLARE-VARIABLE B( TYPE= *INTEGER )
/A = INT-TO-X-LIT( -235736076 )
/SHOW-VARIABLE A
A = X'F1F2F3F4'

/B = X-LIT-TO-INT(&(A)) &* take account of & replacement
/SHOW-VARIABLE B
B = -235736076

/B = X-LIT-TO-INT('1234')
/SHOW-VARIABLE B
B = -235736076

/A = INT-TO-X-LIT( 0 )
/SHOW-VARIABLE A
A = X'00000000'
```

IS-C-LITERAL( ) Check C literal

Domain: **String functions/test functions**

The IS-C-LITERAL( ) function checks whether the specified string is a C literal and can be converted to a string (using the FROM-C-LITERAL function).

Format

IS-C-LITERAL( )
IS-C-LIT( )

STRING = string_expression

Result type

BOOLEAN

Input parameters

STRING = string_expression
Designates the string expression whose content is to be checked.

Result

TRUE
“string_expression” contains a C literal and can be converted to a string, for example using the FROM-C-LITERAL function.

FALSE
“string_expression” does not contain a C literal.

Error messages

No error messages

Example

```
/A = 'C''abc'''
/D =IS-C-LITERAL(STRING = A)
/SHOW-VARIABLE D
D = TRUE

/B = C'abc'
/D = IS-C-LITERAL(STRING = B)
/SHOW-VARIABLE D
D = FALSE

/C = '''abc'''
/D = IS-C-LITERAL(STRING = C)
/SHOW-VARIABLE D
D = TRUE
```

IS-CATALOGED-FILE( ) Check catalog entry

Domain: **File information**

The IS-CATALOGED-FILE( ) function checks whether there is a catalog entry with the specified file name. The response when an error occurs (invalid file name, etc.) can be defined.

Format

IS-CATALOGED-FILE( )
IS-CAT-FILE( )

FILE = string_expression ,ERROR-REPORTING = <u>*PROC-ERROR-MECHANISM</u> / *RETURN-FALSE ,ERROR-VARIABLE = <u>*NONE</u> / string_expression

Result type

BOOLEAN

Input parameters

FILE = string_expression
Designates a file and must therefore comply with the SDF data type <filename 1...54 without-gen-vers>.

If the string contains catalog and user IDs, a search is made for the catalog entry in the user catalog of the specified user ID. This is done on the pubset with the specified catalog ID.

If the string contains a user ID but not a catalog ID, a search is carried out for the catalog entry in the user catalog of the specified user ID on the pubset assigned to the user ID as the default pubset.

If the string does not contain a user ID, the user ID of the current job, i.e. the user ID of the SET-LOGON-PARAMETERS command, is used. Then, depending on whether a catalog ID was specified, the user catalog on the default pubset or the specified pubset is searched.

ERROR-REPORTING =
You can define if error handling is to be triggered when an error occurs or if the message code of the error message is to be stored in an S variable.

ERROR-REPORTING = ***PROC-ERROR-MECHANISM**
Error handling is to be triggered when an error occurs, see the section “Error handling” on page 43.

ERROR-REPORTING = *RETURN-FALSE
The result *FALSE* is to be output when an error occurs. No error message is output. The message code of the error message is written in the variable specified in ERROR-VARIABLE = ...

ERROR-VARIABLE =
An S variable can be defined for the message code. The message code is only written in the variable when ERROR-REPORTING = *RETURN-FALSE was specified in the function call.

ERROR-VARIABLE = *NONE
No S variable is defined .

ERROR-VARIABLE = string_expression
Name of the S variable in which the message code of the error message is written. Note the following items:
- If the variable name is specified directly, then it must be enclosed in quotes, otherwise the contents of the variable are interpreted as a variable name.
- Data is only written in the S variable when an error occurs (result=FALSE) and ERROR-REPORTING=*RETURN-FALSE was specified. Examples of possible message codes: SDP0439, SDP0440 or DMSxxxx.
- If an error occurs while writing to the S variable, then the corresponding error message is output to SYSOUT regardless of the value specified for ERROR-REPORTING, and the S variable does not contain a return value.

Result

TRUE
The file designated in the FILE parameter is cataloged.

FALSE
The file designated in the FILE parameter is not cataloged or an error occurred during a call with ERROR-REPORTING=*RETURN-FALSE.

Error message

```
SDP0439   LENGTH OF FILE NAME ZERO OR GREATER THAN 54

SDP0440   NAME '(&00)' NOT A FILE NAME OR NOT A SPECIFIC FILE NAME

SDP0441   DMS ERROR '(&00)' WHEN CALLING FSTAT MACRO. IN SYSTEM MODE: HELP-
          MSG (&00)
```

Example 1

A tape file named TAPE.A is to be read. It is necessary to first check whether TAPE.A has already been cataloged or whether TAPE.A must be cataloged before it is imported.

```
/IF (NOT IS-CATALOGED-FILE(FILE = 'BAND.A'))
/IMPORT-FILE SUPPORT=*TAPE(FILE-NAME=BAND.A,DEVICE-TYPE=...,VOLUME=...)
/END-IF
```

Example 2

```
/DECLARE-VARIABLE NAME=(A(TYPE=*BOOL),B(TYPE=*STRING),C(TYPE=*BOOL))
/A = IS-CATALOGED-FILE(FILE=‘A_A',ERROR-REPORTING=*RETURN-FALSE,-
/                                 ERROR-VARIABLE='B')
/C = IS-CATALOGED-FILE(FILE='A_A')
/ . . .
/SET-JOB-STEP
/SHOW-VARIABLE SELECT=*BY-ATTRIBUTES(INITIALIZATION=*ANY)
```

Trace listing

```
% 1  1 /DECLARE-VARIABLE NAME=(A(TYPE=*BOOL), B(TYPE=*STRING), C(TYPE=*BOOL))
% 2  1 /A = IS-CATALOGED-FILE(FILE='A_A',ERROR-REPORTING=*RETURN-FALSE,ERROR-
VARIABLE='B')
% 3  1 /C = IS-CATALOGED-FILE(FILE='A_A')
%  SDP0440 NAME 'A_A' NOT A FILE NAME OR NOT A SPECIFIC FILE NAME
%  SDP0431 ERROR 'SDP0440' IN BUILTIN FUNCTION 'IS-CATALOGED-FILE'
%  SDP0239 ERROR DURING EVALUATION OF RIGHT SIDE OF ASSIGNMENT
%  SDP0004 ERROR DETECTED AT COMMAND LINE: 3 IN PROCEDURE ':R:$CSLTOM.PROC'
% 5  1 /SET-JOB-STEP
% 6  1 /SHOW-VARIABLE SELECT=*BY-ATTR(INIT=*ANY)
A = FALSE
B = SDP0440
C = *NO-INIT
*END-OF-CMD
```

IS-CATALOGED-JV( ) Request job variable

Domain: **Job variables**

The IS-CATALOGED-JV( ) function checks whether a catalog entry exists for the specified job variable names, i.e. whether the specified job variable exists.

This function requires the JV subsystem to be loaded. For more information on job variables refer to the “Job Variables” manual [5].

Format

IS-CATALOGED-JV( )
IS-CAT-JV( )

JV = string_expression ,ERROR-REPORTING = <u>*PROC-ERROR-MECHANISM</u> / *RETURN-FALSE ,ERROR-VARIABLE = <u>*NONE</u> / string_expression

Result type

BOOLEAN

Input parameters

JV = string_expression
Designates a job variable and must therefore correspond to the SDF data type <filename 1...54 without-gen-vers>.

ERROR-REPORTING =
You can define if error handling is to be triggered when an error occurs or if the message code of the error message is to be stored in an S variable.

ERROR-REPORTING = <u>*PROC-ERROR-MECHANISM</u>
Error handling is to be triggered when an error occurs, see the section “Error handling” on page 43.

ERROR-REPORTING = *RETURN-FALSE
The result *FALSE* is to be output when an error occurs. No error message is output. The message code of the error message is written in the variable specified in ERROR-VARIABLE = ...

ERROR-VARIABLE =
An S variable can be defined for the message code. The message code is only written in the variable when ERROR-REPORTING = *RETURN-FALSE was specified in the function call.

ERROR-VARIABLE = *NONE
No S variable is defined .

ERROR-VARIABLE = string_expression
Name of the S variable in which the message code of the error message is written. Note the following items:

- If the variable name is specified directly, then it must be enclosed in quotes, otherwise the contents of the variable are interpreted as a variable name.
- Data is only written in the S variable when an error occurs (result=FALSE) and ERROR-REPORTING=*RETURN-FALSE was specified. Examples of possible message codes: SDP0439, SDP0440 or DMSxxxx.
- If an error occurs while writing to the S variable, then the corresponding error message is output to SYSOUT regardless of the value specified for ERROR-REPORTING, and the S variable does not contain a return value.

Result

TRUE
The job variable designated in the JV parameter is cataloged.

FALSE
The job variable designated in the JV parameter is not cataloged an error occurred during a call with ERROR-REPORTING=*RETURN-FALSE.

Error message

```
SDP0495   '(&00)' NOT A CORRECT JV NAME

SDP1054   JOB VARIABLE ERROR: JVS ERROR CODE '(&00)' WHILE ACCESSING JOB
          VARIABLE '(&01)'. IN SYSTEM MODE: /HELP-MSG JVS(&00)
```

Example

```
/IF (IS-CATALOGED-JV(JV='PS'))
/   WRITE-TEXT 'EXISTS'
/ELSE
/   WRITE-TEXT 'CREATE'
/END-IF
```

Output:
```
CREATE
```

IS-DECLARED( ) Check variable declaration

Domain: **Variable access/test functions**

The IS-DECLARED( ) function checks whether the specified simple or complex variable has already been declared.

Format

IS-DECLARED( )
VARIABLE-NAME = string_expression ,SCOPE = <u>*BY-HIERARCHY</u> / *TASK / *CALLING-PROCEDURES

Result type

BOOLEAN

Input parameters

VARIABLE-NAME = string_expression
Designates a variable. The variable name must be enclosed in apostrophes if it is specified directly, i.e. as a literal (see the example on the next page).

SCOPE =
Designates the scope in which the variable is searched for.

SCOPE = <u>*BY-HIERARCHY</u>
The INCLUDE scope is searched first, then the PROCEDURE scope and finally the task scope. Task variables are only visible if they have been imported.

SCOPE = *TASK
Only the task scope is searched.

SCOPE = *CALLING-PROCEDURES
Checks whether the specified variable has already been declared with IMPORT-ALLOWED = *YES. The search for the variable proceeds from the calling procedure upwards to the dialog level (in a foreground procedure) or up to the first procedure (in a background procedure). If the variable found has been declared with IMPORT-ALLOWED = *NO, the search is resumed, provided that the entire scope has not already been searched.
If this check eventually finds a variable with the specified name which has been declared with IMPORT-ALLOWED = *YES, the value TRUE is returned. If no such variable is found, the value FALSE is returned.

Result

TRUE
The variable designated in the VARIABLE-NAME parameter has been declared in the specified scope (or has been declared with IMPORT-ALLOWED=*YES, as appropriate).

FALSE
The variable designated in the VARIABLE-NAME parameter has not been declared in the specified scope (or has not been declared with IMPORT-ALLOWED=*YES, as appropriate).

Error messages

```
SDP0010   TYPE OF PARAMETER '(&00)' INVALID

SDP1101   SYNTAX ERROR IN VARIABLE NAME
```

Example

The following variable “A” is declared and the procedure “proc2” called in procedure “proc1”:

```
/DECLARE-VARIABLE VARIABLE-NAME=A(TYPE=*INTEGER,INITIAL-VALUE=12),-
/SCOPE=*PROCEDURE(IMPORT-ALLOWED=*YES)
/CALL-PROCEDURE Proc2
```

Procedure “proc2” contains the following:

```
/B=IS-DECLARED(VARIABLE-NAME=‘A‘,SCOPE=*CALLING-PROCEDURES)
/SHOW-VARIABLE VARIABLE-NAME=B
```

Output

```
B=TRUE
```

IS-EMPTY-FILE( ) Check file size

Domain: **Environment information**

The IS-EMPTY-FILE( ) function checks whether a file is empty (last-page pointer points to page 0). This is true if the output field HIGH-US-PA in the output from the SHOW-FILE-ATTRIBUTES command contains the value 0.

Format

IS-EMPTY-FILE( )
FILE-NAME = string_expression

Result type

BOOLEAN

Input parameters

FILE-NAME = string_expression
Designates the file to be checked.

Result

TRUE
Indicates that the file is empty.

FALSE
Indicates that the file is not empty.

Error messages

```
SDP0093   ERROR DURING ACCESS OF FILE/LIBRARY '(&00)' , ERROR '(&01)'
          MORE INFORMATION: /HELP-MSG (&01)

SDP0440   NAME '(&00)' NOT A FILE NAME OR NOT A SPECIFIC FILE NAME

SDP0453   (&00) PARAMETER IS EMPTY OR ITS LENGTH IS GREATER THAN (&01)
          CHARACTERS OR CONTAINS ONE OR MORE SPACES
```

Example

```
/CREATE-FILE newfile
/IF (IS-EMPTY-FILE (‘newfile‘))
/     WRITE-TEXT ‘This file is empty‘
/ELSE
/     WRITE-TEXT ‘This file is not empty‘
/END-IF
```

Output

```
This file is empty
```

IS-INITIALIZED( ) Check variable initialization

Domain: **Variable access/test functions**

The IS-INITIALIZED( ) function checks whether the specified variable is initialized, i.e. whether its content is valid. Even the null string is a valid variable content.

Only simple variables or list variables can be checked.

Format

IS-INITIALIZED( )
VARIABLE-NAME = string_expression

Result type

BOOLEAN

Input parameters

VARNAME = string_expression
Designates a simple variable or list variable. The variable name must be enclosed in apostrophes if it is specified directly, i.e. as a literal (see the example below and the example in the description of IS-DECLARED( )).
A list variable must be specified as ’listname#’. Individual list elements can be specified as ’listname#elementindex’.

Result

TRUE
The variable designated by the VARIABLE-NAME parameter is initialized.

FALSE
The variable designated by the VARIABLE-NAME parameter is not initialized.

Error message

```
SDP1101   SYNTAX ERROR IN VARIABLE NAME
```

Example

```
/DECLARE-VARIABLE X
/DECLARE-VARIABLE A
/A = 'ABC'

/B = IS-INITIALIZED(VARIABLE-NAME = 'A')
/SHOW-VARIABLE B
B = TRUE

/B = IS-INITIALIZED(VARIABLE-NAME = 'AA')
/SHOW-VARIABLE B
B = FALSE

/B = IS-INITIALIZED(VARIABLE-NAME = 'X')
/SHOW-VARIABLE B
B = FALSE

/FREE-VARIABLE(NAME = A)
/B = IS-INITIALIZED(VARIABLE-NAME = 'A')
/SHOW-VARIABLE B
B=FALSE
```

Since variable A has no content after FREE-VARIABLE, the result FALSE is supplied.

IS-INTEGER( ) Check expression

Domain: **Variable access/test functions**

The IS-INTEGER( ) function checks whether the expression specified as a string represents an integer:

- The string may consist only of digits 0 through 9 and the signs + and -.
- The sign + or - must be placed directly in front of the number, i.e. the signs and corresponding numbers should not be separated by blanks.
- The value of the expression must lie within the valid range of -2^{31} to $2^{31}-1$.

Blanks are permitted at the beginning and end of the string, i.e. the string can be filled with right- or left-justified blanks. If the checked string contains an integer, it can, for example, be subsequently converted with the INTEGER( ) function.

Format

IS-INTEGER( )
STRING = string_expression

Result type

BOOLEAN

Input parameters

STRING = string_expression
Designates the string to be checked for integers.

Result

TRUE
The string contains an integer, i.e. it can be converted to an integer value.

FALSE
The string does not contain an integer.

Example

```
/A = IS-INTEGER (STRING = '   -123')
/SHOW-VARIABLE A
A = TRUE

/B = IS-INTEGER(STRING = '+(123-3)')
/SHOW-VARIABLE B
B = FALSE
```

In the first case, the string contains an integer value. The included blanks are permitted. The result is therefore: A = TRUE. In the second case, the string contains an expression, not an integer value. The result is therefore: B = FALSE.

IS-LIBRARY( ) Check library name

Domain: **Test functions**

The IS-LIBRARY( ) function checks whether the specified file is entered in the catalog as a PLAM library.

If the file is not entered in the catalog as a PLAM library, IS-LIBRARY( ) supplies the result FALSE.

If the specified file does not exist, error handling is initiated.

See the “LMS” manual [11] for more information on working with PLAM libraries.

Format

IS-LIBRARY( )
FILE = string_expression

Result type

BOOLEAN

Input parameters

FILE = string_expression
“string_expression” designates a file name according to the SDF file type <filename 1...54>.

Result

TRUE
The file designated with the parameter FILE is entered in the catalog as a PLAM library.

FALSE
The file designated with the parameter FILE is not entered in the catalog as a PLAM library.

Error messages

No error messages

Example

```
/A = IS-LIBRARY('MY-LIBRARY')
/SHOW-VARIABLE A
/A = TRUE
/SHOW-FILE-ATTRIBUTES MY-LIBRARY,INF=ALL
```

The output from the SHOW-FILE-ATTRIBUTES command shows the complete catalog entry for MY-LIBRARY. The field TYPE contains the value PLAM-LIB.

```
%0000000012 :2OSG:$USER1.MY-LIBRARY
%  ------------------------------- HISTORY      ------------------------------
%  CRE-DATE   = 1996-06-03  ACC-DATE   = 2007-04-19  CHANG-DATE = 2006-06-04
%  CRE-TIME   =   13:41:01  ACC-TIME   =   10:51:10  CHANG-TIME =   12:34:10
%  ACC-COUNT  = 40          S-ALLO-NUM = 0
%  ------------------------------- SECURITY     ------------------------------
%  READ-PASS  = NONE        WRITE-PASS = NONE        EXEC-PASS  = NONE
%  USER-ACC   = OWNER-ONLY  ACCESS     = WRITE       ACL        = NO
%  OWNER      = R W X       GROUP      = R - X       OTHERS     = R - X
%  AUDIT      = NONE        FREE-DEL-D = *NONE       EXPIR-DATE = 2009-10-06
%  DESTROY    = NO          FREE-DEL-T = *NONE       EXPIR-TIME =   00:00:00
%  SP-REL-LOCK= NO          ENCRYPTION = *NONE
%  ------------------------------- BACKUP       ------------------------------
%  BACK-CLASS = A           SAVED-PAG  = COMPL-FILE  VERSION    = 2
%  MIGRATE    = ALLOWED
%  ------------------------------- ORGANIZATION ------------------------------
%  FILE-STRUC = PAM         BUF-LEN    = STD(1)      BLK-CONTR  = PAMKEY
%  IO(USAGE)  = READ-WRITE  IO(PERF)   = STD         DISK-WRITE = IMMEDIATE
%  TYPE       = PLAM-LIB
%  AVAIL      = *STD
%  WORK-FILE  = *NO         F-PREFORM  = *NONE       SO-MIGR    = *ALLOWED
%  ------------------------------- ALLOCATION   ------------------------------
%  SUPPORT    = PUB         S-ALLOC    = 24          HIGH-US-PA = 9
%  EXTENTS     VOLUME     DEVICE-TYPE      EXTENTS     VOLUME     DEVICE-TYPE
%      1       GVS2.1      D3435
%  NUM-OF-EXT = 1
%:2OSG: PUBLIC:      1 FILE  RES=        12 FRE=        3 REL=        3 PAGES
```

IS-LIBRARY-ELEMENT( ) Check library element

Domain: **Test functions**

The IS-LIBRARY-ELEMENT( ) function checks whether or not the specified library element exists.

Format

```
IS-LIBRARY-ELEMENT( )
IS-LIB-ELEM( )
---------------------------------------------------------------
 LIBRARY = string_expression
,ELEMENT = string_expression
,TYPE = string_expression
,VERSION = *HIGHEST-EXISTING / string_expression
```

Result type

BOOLEAN

Input parameters

LIBRARY = string_expression
“string_expression” designates a file name which corresponds to the SDF data type <filename 1...54>.

ELEMENT = string_expression
“string_expression” designates a library element which corresponds to the SDF data type <composed-name 1...64>.

TYPE = string_expression
“string_expression” designates a library element type which corresponds to the SDF data type <alphanum-name 1...8>.

VERSION = *HIGHEST-EXISTING / string_expression
Designates a library element version which corresponds to the SDF data type <composed-name 1...24>.

Result

TRUE
The specified library element exists.

FALSE
The specified library element does not exist.

Error messages

```
SDP0093   ERROR DURING ACCESS OF FILE/LIBRARY ‘(&00)‘ , ERROR ‘(&01)‘
          MORE INFORMATION: /HELP-MSG ‘(&01)‘

SDP0454   INVALID PARAMETER : ‘(&00)‘
```

IS-SDF-LIST( ) Analyze string against criteria for SDF lists

Domain: **String functions/test functions**

The IS-SDF-LIST( ) function analyzes whether a string expression is an SDF list of the format ’<element$_1$>,<element$_2$>,...,<element$_n$>)' where <element$_i$> is a character sequence that must not contain any comma (except within parentheses).

Format

IS-SDF-LIST( )
STRING = string_expression

Result type

BOOLEAN

Input parameters

STRING = string_expression
Name of the string to be analyzed.

Result

TRUE
The string expression is an SDF list.

FALSE
The string expression is not an SDF list.

Error messages

No error messages

Example

```
/A=IS-SDF-LIST('(val1,val2)')
SHOW-VAR A
A = TRUE
```

```
/A=IS-SDF-LIST('val')
/SHOW-VAR A
A = FALSE
```

IS-SDF-P( ) Check whether SDF-P is loaded

Domain: **Test functions**

The IS-SDF-P( ) function checks whether SDF-P is loaded in the system. If it is loaded, the result TRUE is returned. FALSE is returned as the result in the following cases:

- SDF-P is not loaded.
- SDF-P is loaded but the SDF-P-BASYS functionality is being simulated in the task at the moment (FUNCTIONALITY=*BASIC setting in the MODIFY-PROCEDURE-TEST-OPTIONS command, see page 697).

Format

IS-SDF-P( )

Result type

BOOLEAN

Input parameters

None

Result

TRUE
SDF-P is loaded in the system.

FALSE
SDF-P is not loaded in the system or the SDF-P-BASYS functionality is currently being simulated in the task.

Error messages

No error messages

Example

```
/A=IS-SDF-P()
/SHOW-VARIABLE A
A = TRUE
```

Application: A procedure is to be executable if only the SDF-P-BASYS functionality is available (for example, procedure parameters can be read in with SDF-P with the READ-VARIABLE command, and further checks and/or corrections can be performed on the input. the parameter can only be entered at a prompt when SDF-P is not used):

```
/SET-PROCEDURE-OPTIONS
/DECLARE-PARAMETER A(INIT-VALUE=*PROMPT)
/...
/IF (IS-SDF-P())
/
/    “READ THE VARIABLE WITH HELP-TEXT AND CHECK THE RESULT”
/  IF (TASK-MODE() == 'DIALOG')
/RE-READ:
/      READ-VARIABLE A,INPUT=*TERMINAL(PROMPT='PLEASE ENTER THE FILE NAME')
/      IF (NOT CHECK-DATA-TYPE (A,*FULL-FILENAME))
/            WRITE-TEXT 'ERROR: &A IS NOT A FILENAME'
/             GOTO RE-READ
/      END-IF
/  ELSE
/      IF (NOT CHECK-DATA-TYPE (A,*FULL-FILENAME))
/            WRITE-TEXT 'ERROR: &A IS NOT A FILENAME'
/             EXIT-PROCEDURE
/       END-IF
/  END-IF
/  “FURTHER CHECKS MAY BE CARIED OUT HERE”
/  ....
/ELSE
/          “BASIC PROCESSING USING SDF-P-BASYS”
/          WRITE-TEXT 'PLEASE ENTER THE FILE NAME:'
           REMARK &A
/END-IF
/
START-LMS
//OPEN &A,MODE=*READ
//..
```

IS-SDF-STRUCTURE( ) Analyze string against criteria for SDF structures

Domain: **String functions/test functions**

The IS-SDF-STRUCTURE( ) function analyzes whether the specified string is an SDF structure.

The specified SDF structure must be introduced by a value. This value must not be omitted, or else the string will be regarded as an SDF list and not as an SDF structure.

Format

IS-SDF-STRUCTURE( )
STRING = string_expression

Result type

BOOLEAN

Input parameters

STRING = string_expression
Name of the string which is to be analyzed.

Result

TRUE
The specified string is an SDF structure.

FALSE
The specified string is not an SDF structure.

Error messages

No error messages

Example

```
/A=IS-SDF-STRUCTURE('*P(val1,val2)')
/SHOW-VAR A
A = TRUE
```

```
/A=IS-SDF-STRUCTURE('(val1,val2)')
A = FALSE
```

In this case the string will be regarded as an SDF list.

```
/A=IS-SDF-STRUCTURE('val')
/SHOW-VAR A
A = FALSE
```

IS-VARIABLE-NAME( ) Check variable name

Domain: **String functions/test functions**

The IS-VARIABLE-NAME( ) function checks whether the specified string is a syntactically correct variable name. This check is a pure syntax check. It does not check whether a variable with this name exists.

Format

IS-VARIABLE-NAME( )
IS-VAR-NAME( )

STRING = string_expression

Result type

BOOLEAN

Input parameters

NAME = string_expression
Designates the string to be checked.

Result

TRUE
“string_expression” is a valid variable name.

FALSE
“string_expression” does not satisfy the syntax rules for variable names and is therefore not a valid variable name.

Error messages

No error messages

Example

```
/DECLARE-VARIABLE A
/A = '1234'
/B = IS-VARIABLE-NAME(STRING = A)
/SHOW-VARIABLE B
B = FALSE
```

The variable A is declared and the value '1234' is assigned to it. IS-VARIABLE-NAME( ) now checks to see if variable A contains a valid variable name: ’1234’ is not a valid variable name, since variable names cannot begin with a number.

```
/B = IS-VARIABLE-NAME(STRING = 'A')
/SHOW-VARIABLE B
B = TRUE
```

IS-VARIABLE-NAME( ) checks the string ’A’: ’A’ is a valid variable name.

IS-X-LITERAL( ) Check X literal

Domain: **String functions/test functions**

The IS-X-LITERAL( ) function checks whether the specified string expression contains an X literal and can be converted with FROM-X-LITERAL( ).

Format

IS-X-LITERAL( )
IS-X-LIT( )

STRING = string_expression

Result type

BOOLEAN

Input parameters

STRING = string_expression
Designates the string expression to be checked.

Result

TRUE
"string_expression" contains an X literal.

FALSE
"string_expression" does not contain an X literal.

Error messages

No error messages

Example

```
/A = 'X''01FF'''
/B = X'01FF'
/C = IS-X-LITERAL(STRING = A)
/SHOW-VARIABLE C
C = TRUE
/C = IS-X-LITERAL(STRING = B)
/SHOW-VARIABLE C
C = FALSE
```

In the first case (variable A), the string to be checked contains an X literal. In the second case (variable B), it does not, since the internal value of the variable is 01FF, which is not an X literal.

JOB-CLASS( ) Request job class

Domain: **Job information**

The JOB-CLASS( ) function requests the job class to which the current task belongs.

Format

JOB-CLASS( )

Result type

STRING

Input parameters

None

Result

Job class in the form of a string.

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

```
/A = JOB-CLASS( )
/SHOW-VARIABLE A
A = JCDSTD
```

For comparison (output format: BS2000/OSD-BC V5.0):

```
/show-job-status
%TSN:     29XX       TYPE:    3 DIALOG   NOW:     2007-04-26.110747
%JOBNAME:            PRI:     0 210
%USERID:  USER1      JCLASS:  JCDSTD     LOGON:   2007-04-26.1053
%ACCNB:   89001      CPU-MAX:   9999     CPU-USED:000000.6447
%STATION: $$$06580   PROC:    FIREBALL
%TID:     000101AB   UNP/Q#:    00/000
%CMD:     SHOW-JOB-STATUS
%MONJV:   *NONE
```

JOB-MONJV( ) Request MONJV

Domain: **Job variable functions**

The JOB-MONJV( ) function supplies the name of the job variable which monitors the job.

This function can be used only if the JV subsystem has been loaded in the system. For more information on job variables refer to the “Job Variables” manual [5].

Format

JOB-MONJV( )

Result type

STRING

Input parameters

None

Result

Name of the job variable.

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

JOB-NAME( ) Request job name

Domain: **Job information**

The JOB-NAME( ) function supplies the job name of the current task, i.e. the name specified in the SET-LOGON-PARAMETERS command.

Format

JOB-NAME( )

Result type

STRING (<string 1..8>)

Input parameters

None

Result

Job name as specified in the SET-LOGON-PARAMETERS command.

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

```
/SET-LOGON-PARAMETERS USER1,ACC01,'PASSWORD',JOB-NAME=BERTA
/ ...
/B = JOB-NAME( )
/SHOW-VARIABLE B
B = BERTA
```

For comparison (output format BS2000/OSD-BC V5.0):

```
/show-job-status
%TSN:     29XX       TYPE:    3 DIALOG   NOW:     2007-04-26.110747
%JOBNAME: BERTA      PRI:     0 210
%USERID:  USER1      JCLASS:  JCDSTD     LOGON:   2007-04-26.1053
%ACCNB:   ACC01      CPU-MAX:   9999     CPU-USED:000000.6447
%STATION: $$$06580   PROC:    FIREBALL
%TID:     000101AB   UNP/Q#:    00/000
%CMD:     SHOW-JOB-STATUS
%MONJV:   *NONE
```

The JOBNAME field displays the name BERTA that was specified in the SET-LOGON-PARAMETERS command.

JV( ) Request job variable

Domain: **Job variable functions**

The JV( ) function supplies the content of the specified job variable.

This function can be used only if the JV subsystem is loaded in the system. See the “Job Variables” [5] manual for more information on job variables.

Format

JV( )
JV-NAME = string_expression ,START = 1 / arithm_expression1 ,LENGTH = *REST-LENGTH / arithm_expression2

Result type

STRING

Input parameters

JV-NAME = string_expression
Designates a job variable; “string_expression” must therefore be a valid job variable name or a JV link name identified by a preceding asterisk (*).

START= 1 / arithm_expression1
Designates the start position of the JV contents to be extracted. Unless otherwise specified, this is the first character. arithm_expression1 must be a positive integer value which is less than the total length of the JV. If the value specified for arithm_expression1 is not a valid one, a null string will be returned.

LENGTH = *REST-LENGTH / arithm_expression2
Designates the length of the JV contents to be extracted. The default value of *REST-LENGTH assumes that the JV contents to be extracted start at the position indicated by START and extend to the end. If a different length is specified in arithm_expression2, and if this is too long, then LENGTH = *REST-LENGTH is implicitly assumed.

Result

Contents of the job variable designated by “string_expression”, or that part of it designated by “arithm_expression1” and “arithm_expression2”.

Error messages

SDP0412	START POSITION OUT OF RANGE
SDP0414	WARNING: *REST-LENGTH VALUE USED FOR LENGTH OPERAND
SDP1022	JV: JOB VARIABLE '(&00)' NOT ACCESSIBLE
SDP1024	JV: JOB VARIABLE '(&00)' DOES NOT EXIST
SDP1027	VALUE FOR JOB VARIABLE '(&00)' IS NOT A STRING
SDP1054	JOB VARIABLE ERROR: JVS ERROR CODE '(&00)' WHILE ACCESSING JOB VARIABLE '(&01)'. IN SYSTEM MODE: /HELP-MSG JVS(&00)

Example

```
/CREATE-JV JV-NAME=HUGO
/MODIFY-JV JV-CONTENTS=HUGO,SET-VALUE=c'switch is on'
/A = JV('HUGO')
/SHOW-VARIABLE A
A = switch is on

/B = JV('HUGO',4,3)
/SHOW-VARIABLE B
B = tch
```

LAYOUT-SCOPE( ) Request layout scope

Domain: **Variable access (variable name)**

The LAYOUT-SCOPE( ) function applies only to structure layouts and supplies their scope. The scope of a structure layout is defined in the declaration of structure elements in the BEGIN-STRUCTURE command.

Format

LAYOUT-SCOPE( )
LAYOUT-NAME = string_expression

Result type

STRING

Input parameters

LAYOUT-NAME = string_expression
Designates a structure layout.

Result

**TASK*
The layout is declared with the scope TASK.

**PROCEDURE*
The layout is declared with the scope PROCEDURE.

**INCLUDE*
The layout is declared with the scope INCLUDE.

Error messages

```
SDP0442   LAYOUT DOES NOT EXIST

SDP1101   SYNTAX ERROR IN VARIABLE NAME
```

Example

```
/BEGIN-STRUCTURE LAY1
/   DECLARE-ELEMENT ELEM1
/   DECLARE-ELEMENT ELEM2
/   DECLARE-ELEMENT ELEM3
/END-STRUCTURE
/A = LAYOUT-SCOPE(LAYOUT-NAME='LAY1')
/SHOW-VARIABLE A
A = *PROCEDURE
```

The structure layout LAY1 is declared in the structure declaration block between BEGIN-STRUCTURE and END-STRUCTURE. The scope PROCEDURE is the default assignment for the structure layout.

LENGTH( ) Request string length

Domain: **String analysis**

The LENGTH( ) function supplies the length of the specified string.

Format

LENGTH( )
STRING = string_expression

Result type

INTEGER

Input parameters

STRING = string_expression
Designates the string whose length is to be requested.

Result

Number of type INTEGER

Error message

```
SDP1010   VARIABLE '(&00)' HAS NO VALUE
```

Example 1

```
/SET-VARIABLE A = 'ANNAMARIA'
/SET-VARIABLE B = LENGTH(A)
/SHOW-VARIABLE B
B = 9
```

Example 2

```
/DECLARE-VARIABLE VARLIST,MULTIPLE-ELEMENTS = *LIST
/  VARLIST = *STRING-TO-VAR('(Terminal,Printer,Keyboard,Processor)')
/SHOW-VARIABLE VARLIST
VARLIST(*LIST) = Terminal
VARLIST(*LIST) = Printer
VARLIST(*LIST) = Keyboard
VARLIST(*LIST) = Processor
/FOR A = *LIST(VARLIST)
/     B = LENGTH(A)
/     SHOW-VARIABLE B
/END-FOR
```

Output:

```
B = 8
B = 7
B = 8
B = 9
```

A list variable VARLIST is declared with default values. The first four elements of VARLIST are then assigned a value, in this case a word (the four values are specified in one assignment in this case, see the SET-VARIABLE command, page 740). In the FOR loop, the length of each list element is checked and displayed.

LIMIT( ) Request maximum list size

Domain: **Variable access (variable name)**

The LIMIT( ) function can be applied only to lists. LIMIT( ) requests the number of elements which a list variable can contain. This limit is defined during declaration of list variables using the DECLARE-VARIABLE command in the operand MULTIPLE-ELEMENTS = *LIST(LIMIT = integer).

Format

LIMIT( )
LIST-NAME = string_expression

Result type

INTEGER (<integer 1 .. 2147483647>)

Input parameters

LIST-NAME = string_expression
Designates a list.

Result

Number of permissible elements; returned in the form of an integer.

Error messages

```
SDP0426   VARIABLE '(&00)' NOT A LIST
SDP1007   NO VARIABLE DECLARED
SDP1101   SYNTAX ERROR IN VARIABLE NAME
```

Example

```
/DECLARE-VAR LIST3, MULT-ELEM=*LIST(LIMIT=10)
/DECLARE-VAR LIST4, MULT-ELEM=*LIST

/A = LIMIT('LIST3')
/SHOW-VAR A
A = 10

/A = LIMIT('LIST4')
/SHOW-VARIABLE A
/A = 2147483647
```

The list variable LIST3 was declared with LIMIT=10. LIMIT( ) thus also supplies the value 10.
The list variable LIST4 was declared with the default values. LIMIT( ) thus supplies the default value for the maximum list size (2^{31}-1).

LOGGING-MODE( ) Check logging

Domain: **Procedure information**

The LOGGING-MODE( ) function indicates whether the logging function was activated when calling the CALL-PROCEDURE or INCLUDE-PROCEDURE command.

Logging of procedure execution is defined when calling the CALL- or INCLUDE-PROCEDURE command in the LOGGING operand. Logging can be set independently for command sequences and data stream. If BY-PROC-TEST-OPTION applies to one or both of the above, the current log status is determined by means of the MODIFY-PROC-TEST-OPTIONS command.

Please note that the LOGGING-MODE( ) must be called separately for commands and data.

Format

LOGGING-MODE( )
LOG-MODE( )

STREAM = *CMD / *DATA

Result type

STRING ('YES' / 'NO')

Input parameters

STREAM =
Defines the logging type to be requested.

STREAM = *CMD
Requests whether the command sequence is to be logged.

STREAM = *DATA
Requests whether the data stream is to be logged.

Result

YES
Commands/data are logged.

NO
Commands/data are not logged.

Error messages

No error messages

Example

```
/CALL-PROCEDURE PROC, LOGGING = *PAR(DATA = *BY-PROC-TEST-OPTION)
```

The log status is checked in the procedure PROC:

```
/IF (LOGGING-MODE (STREAM = *DATA) = 'NO')
/MODIFY-PROCEDURE-TEST-OPTIONS LOGGING = *YES
/END-IF
```

If the data stream is not logged, the logging function is activated with MODIFY-PROCEDURE-TEST-OPTIONS.

LOWER-CASE( ) Convert uppercase letters to lowercase

Domain: **String functions/conversion functions**

The LOWER-CASE( ) function converts all uppercase letters in the specified string to lowercase.

The letters which are to be converted must correspond to the standard EBCDI code. There is no support for language-specific variants.

Format

LOWER-CASE( )
STRING =string_expression ,TRANSLATE = <u>*ALL</u> / *OUTSIDE-QUOTES-ONLY / *INSIDE-QUOTES-ONLY

Result type

STRING

Input parameters

STRING = string_expression
Designates the string to be converted.

TRANSLATE =
Designates which characters are to be converted.

TRANSLATE = <u>*ALL</u>
Specifies that all characters are to be converted.

TRANSLATE = *OUTSIDE-QUOTES-ONLY
Specifies that only characters outside the apostrophes are to be converted.

TRANSLATE = *INSIDE-QUOTES-ONLY
Specifies that only characters inside the apostrophes are to be converted.

Result

A string which consists only of lowercase letters, digits and special characters.

Error message

```
SDP0486   ODD NUMBER OF APOSTROPHES IN STRING VALUE
```

Example

```
/A = 'AbCD123' // 'Ghi'
/B = LOWER-CASE(STRING = A)
/SHOW-VARIABLE B
B = abcd123ghi
```

```
/A = 'ABC‘‘DEF‘‘GHI'
/B = LOWER-CASE(STRING = A,TRANSLATE=*OUTSIDE-QUOTES-ONLY)
/SHOW-VARIABLE B
B = abc‘DEF‘ghi
```

```
/A = 'ABC‘‘DEF‘‘GHI'
/B = LOWER-CASE(STRING = A,TRANSLATE=*INSIDE-QUOTES-ONLY)
/SHOW-VARIABLE B
B = ABC‘def‘GHI
```

MAINCODE( ) Request error code

Domain: **Command return code**

The MAINCODE( ) function accesses the return code of the last command which resulted in an error or which was followed by a /SAVE-RETURNCODE. It returns the seven-byte error code of the return code, which is also the message code for error messages (the remaining components of the command return code are requested with the SUBCODE1( ) and SUBCODE2( ) functions).

The error code supplied by the MAINCODE( ) function consists of two parts: the first three bytes designate the message class, while the last four bytes specify the error. The error code can subsequently be used as the message code in the MSG( ) function; MSG( ) then supplies the corresponding message text if this is available.

MAINCODE( ) is not available, and general command return codes cannot be requested, outside dialog blocks and procedures.

Format

MAINCODE( )
MC( )

Result type

STRING (<string 7..7>)

Input parameters

None

Result

Error code in the form of a string.

Error messages

```
SDP0428   COMMAND RETURN CODE NOT AVAILABLE IN DIALOG

SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

Error handling with MAINCODE( )

```
/BL1: BEGIN-BLOCK
/...
/  IF-BLOCK-ERROR
/    WRITE-TEXT '&(MSG(MAINCODE()))'
/  END-IF
/...
/END-BLOCK BLOCK = BL1
```

Irrespective of whether or not an error occurs in the related block (BL1 here) , the current maincode is evaluated and the corresponding message is displayed.

It should be noted, however, that the message &(MSG(MAINCODE( ))) may itself contain parentheses and apostrophes, and that this will produce problems in specifying the WRITE-TEXT. These problems can be avoided by using the TO-C-LITERAL( ) function, e.g.:

```
/WRITE-TEXT &(TO-C-LITERAL('*** ' // MSG(MAINCODE()) // ' ***'))
```

MONTH( ) Output name of month

Domain: **Environment information** (calendar)

The MONTH( ) function supplies the name of the current month in the specified language and in the form of a three-character abbreviation. The MONTH( ) function can be used in conjunction with the other calendar functions in order to construct a complete date entry.

Format

MONTH( )
LANGUAGE = <u>*ENGLISH</u> / *GERMAN / *STD

Result type

STRING (<string 1..3>)

Input parameters

LANGUAGE = <u>*ENGLISH</u> / *GERMAN / *STD
Determines the language in which the name of the month is to be output.
*STD: The output is displayed in the language contained in the default language setting for the task.

Result

Three-character abbreviation in the form of a string.

Input parameter	**Result**
LANGUAGE = *ENGLISH	JAN / FEB / MAR / APR / MAY / JUN / JUL / AUG / SEP / OCT / NOV / DEC
LANGUAGE = *GERMAN	JAN / FEB / MRZ / APR / MAI / JUN / JUL / AUG / SEP / OKT / NOV / DEZ

Error messages

No error messages

Example

```
/A = MONTH(LANGUAGE = *GERMAN)
/SHOW-VARIABLE A
A = FEB
```

MSG( ) Output message text

Domain: **Command return codes** (messages)

The MSG( ) function supplies the message text assigned to the specified message code; this is done in the specified language and in the specified output format.

For some SDF-P commands, the message code may have been previously requested from the command return code, using the MAINCODE( ) function.

Format

MSG( )
MSG-IDENTIFICATION = string_expression
,LANGUAGE = *STD / *ENGLISH / *GERMAN
,INSERT-00 = *NONE / string_expression
,INSERT-01 = *NONE / string_expression
: : :
,INSERT-29 = *NONE / string_expression
,MSG-STRUCTURE-OUTPUT = *NONE / *SYSMSG

Result type

STRING (<string>)

Input parameters

MSG-IDENTIFICATION = string_expression
string_expression contains the 7-byte message code in the following format:

Bytes 1-3:	Letters identifying the message class
Bytes 4-7:	Digits 0-9, letters A-F as the hexadecimal representation of the exact error number

LANGUAGE = *STD / ENGLISH / *GERMAN
The English or German message text is output.
The default setting is *STD, i.e. the output is displayed using the default language setting of the task. The (previous) operand values *E for *ENGLISH and *D for *GERMAN will continue to be supported for compatibility reasons.

INSERT-nn = *NONE / string_expression
Designates the additional content of a message.

MSG-STRUCTURE-OUTPUT =
Specifies whether or not variables for the output of messages must be created and sent on.

MSG-STRUCTURE-OUTPUT = *NONE
Variables for the output of messages are not sent on via S variable stream SYSMSG. The message text is not supplied by the command /HELP-MSG-INFORMATION MSG-ID=*LAST.

MSG-STRUCTURE-OUTPUT = *SYSMSG
If messages are guaranteed, variables for the output of messages are sent on via S variable stream SYSMSG. The message text can be obtained by the command /HELP-MSG-INFORMATION MSG-ID=*LAST.

Result

Message text in the form of a string.

Null string ('') means:
No text has been assigned to this message code.

Error messages

```
SDP0413   ILLEGAL LENGTH

SDP0418   INVALID MSG-IDENTIFICATION
```

Example

```
/DECLARE-VARIABLE MIP(TYPE=*STRUCTURE(*DYNAMIC)),MULTIPLE-ELEMENTS=*LIST
/ASSIGN-STREAM SYSMSG,TO=*VARIABLE(MIP)
/A=MSG(MSG-IDENTIFICATION='SDP1018',“This message is guaranteed” -
/                INSERT-00='MY-VARIABLE', -
/                MSG-STRUCTURE-OUTPUT = *SYSMSG)
/B=MSG(MSG-IDENTIFICATION='SDP1010',“This message is NOT guaranteed” -
/                INSERT-00='MY-SECOND-VARIABLE', -
/                MSG-STRUCTURE-OUTPUT = *SYSMSG)
/SHOW-VARIABLE *ALL
A = %  SDP1018 VARIABLE 'MY-VARIABLE' ALREADY EXISTS, BUT WITH OTHER
ATTRIBUTES
B = %  SDP1010 VARIABLE 'MY-SECOND-VARIABLE' HAS NO VALUE
MIP(*LIST).MSG-TEXT = % SDP1018 VARIABLE 'MY-VARIABLE' ALREADY EXISTS BUT
WITH OTHER ATTRIBUTES'
MIP(*LIST).MSG-ID = SDP1018
MIP(*LIST).IO = MY-VARIABLE
*END-OF-CMD
```

NEXT-VARIABLE-NAME( ) Request variable level

Domain: **Variable access** (variable name)

The NEXT-VARIABLE-NAME( ) function can be used to analyze the layout of complex variables, primarily in connection with the FIRST-VARIABLE-NAME( ) function.

The NEXT-VARIABLE-NAME( ) function supplies the name of the next variable element on the same level. If there are no more variable elements on this level, NEXT-VARIABLE-NAME( ) returns *END.

Format

NEXT-VARIABLE-NAME( )
NEXT-VAR-NAME( )

VARIABLE-NAME = string_expression

Result type

STRING

Input parameters

VARIABLE-NAME = string_expression
Designates a variable.
If VARIABLE-NAME designates a list, *END or the appropriate element name is output.
If VARIABLE-NAME designates a list element (list#i), the name of the next element is output (list#i+1), as long as one exists. If the list element list#i+1 does not exist, *END is output.
The variable name must be enclosed in apostrophes if it is specified directly, i.e. as a literal (see the following example and the examples in the description of
IS-DECLARED( )).

Result

Name of the element following “string_expression” in the complex variable on the same level.

**END*
“string_expression” was the last element on the level.

Error message

```
SDP1101   SYNTAX ERROR IN VARIABLE NAME
```

Example 1

A compound variable AR contains the following elements.

```
AR#1
AR#2
AR#3

/A = NEXT-VARIABLE-NAME(VARIABLE-NAME = 'AR#1')
/SHOW-VARIABLE A
A = AR#2

/A = NEXT-VARIABLE-NAME(VARIABLE-NAME = 'AR#3')
/SHOW-VARIABLE A
A = *END
```

If NEXT-VARIABLE-NAME( ) is used with AR#1, AR#2 will be supplied (the name of the element following AR#1).
AR#3 is the last element of the array; therefore, NEXT-VARIABLE-NAME( ) supplies *END.

Example 2

A compound variable ARR contains the following elements:

```
ARR#1
ARR#22
ARR#30

/A = NEXT-VAR-NAME('ARR#1')
/SHOW-VARIABLE A
A = ARR#22
```

PROC-LEVEL( ) Request nesting level

Domain: **Procedure information**

The PROC-LEVEL( ) function supplies the current nesting level of the S procedure.

Format

PROC-LEVEL( )

Result type

INTEGER

Input parameters

None

Result

Number of type INTEGER

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

In interactive mode:

```
/A = PROC-LEVEL( )
/SHOW-VARIABLE A
A = 0
```

In nested, called procedures:

The three procedures C.PROC1, C.PROC2 and C.PROC3 are nested when called. The nesting level is requested in each procedure.

C.PROC1

```
/A = PROC-LEVEL( )
/SHOW-VARIABLE A
/CALL-PROCEDURE C.PROC2
```

C.PROC2

```
/B = PROC-LEVEL( )
/SHOW-VARIABLE B
/CALL-PROCEDURE C.PROC3
```

C.PROC3

```
/C = PROC-LEVEL( )
/SHOW-VARIABLE C
```

The following lines are output during execution:

```
A = 1
B = 2
C = 3
```

PROCESSOR( ) Request processor name

Domain: **Environment information** (TIAM)

The PROCESSOR( ) function supplies the processor name of the TIAM station, i.e. the terminal at which the current task was started.

Format

PROCESSOR( )

Result type

STRING (<string 1..8>)

Input parameters

None

Result

TIAM station name in the form of a string.

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

```
/A = PROCESSOR( )
/SHOW-VARIABLE A
A = FIREBALL
```

For comparison: The name of the TIAM station is indicated in the STATION field (BS2000/OSD-BC V7.0 output format):

```
/show-job-status
%TSN:     29XX       TYPE:    3 DIALOG   NOW:     2007-04-26.110747
%JOBNAME: BERTA      PRI:     0 210
%USERID:  USER1      JCLASS:  JCDSTD     LOGON:   2007-04-26.1053
%ACCNB:   ACC01      CPU-MAX:   9999     CPU-USED:000000.6447
%STATION: $$$06580   PROC:    FIREBALL
...
```

PROG-MONJV( ) Request MONJV program

Domain: **Environment information/job variable functions**

The PROG-MONJV( ) function supplies the name of the job variable which monitors the loaded program.

This function can be used only if the software product “Job variables” has been loaded in the system. For more information on job variables refer to the “Job Variables” manual [5].

Format

PROG-MONJV( )

Result type

STRING (<string 1..255>)

Input parameters

None

Result

Job variable name in the form of a string.

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

PROG-NAME( ) Request program name

Domain: **Program information**

The PROG-NAME( ) function supplies the internal name (truncated to eight characters) of the currently loaded program. If the name of an object module from the object module library is requested, 0 characters are output.

Format

PROG-NAME( )

Result type

STRING

Input parameters

None

Result

Program name in the form of a string.

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

The utility LMS is started in a procedure:

```
/START-LMS
...
```

The program run is interrupted and the program name checked:

```
/IF PROG-NAME() = 'LMSSDF'
/   RESUME-PROGRAM
/ELSE
/   ...
/END-IF
```

RENAME( ) Generate new name using wildcards

Domain: **String functions**

The RENAME( ) function provides a new name. This new name is generated on the basis of the input name, using wildcards.

Format

RENAME( )
INPUT-NAME = string_expression ,WILDCARD-PATTERN = string_expression ,CONSTRUCTION-WILDCARD = string_expression ,NO-MATCH = <u>*WARNING</u> / *IGNORE / *ERROR ,WILDCARD-MODE = <u>*BS2000</u> / *POSIX

Result type

STRING

Input parameters

INPUT-NAME = string_expression
Designates the string which is to be replaced.

WILDCARD-PATTERN = string_expression
Specifies the search pattern.

CONSTRUCTION-WILDCARD = string_expression
Specifies the rules by which the new name is to be constructed. (For details see the “LMS” manual [11].)

NO-MATCH =
Specifies what is to be done if the search pattern is not found.

NO-MATCH = <u>*WARNING</u>
A warning is output.

NO-MATCH = *IGNORE
No action.

NO-MATCH = *ERROR
An error message is output.

WILDCARD-MODE = *BS2000 / *POSIX
Specifies how wildcards are to be interpreted during replacement; either in the BS2000 wildcard syntax or in the POSIX wildcard syntax.

Result

The new name, in the form of a string

Error messages

```
SDP0467   NO NAME FOUND; PROCESSING CONTINUES

SDP0468   NO NAME FOUND

SDP0482   AN INPUT STRING IS TOO LONG (1..255)

SDP0483   INCORRECT CONSTRUCTION-WILDCARD VALUE

SDP0484   TOO LONG RESULT STRING (1..255)
```

Example

```
/A = RENAME(‘A.B‘,‘A.*‘,‘NEWA.*‘)
/SHOW-VARIABLE A
A = NEWA.B

/A = RENAME(‘B.A‘,‘A.*‘,‘NEWA.*‘)
SDP0467 NO NAME FOUND; PROCESSING CONTINUES
/SHOW-VARIABLE A
A = B.A

/A = RENAME(‘B.A‘,‘A.*‘,‘NEWA.*‘,NO-MATCH=*IGNORE)
/SHOW-VARIABLE A
A = B.A

/A = RENAME(‘B.A‘,‘A.*‘,‘NEWA.*‘,NO-MATCH=*ERROR)
SDP0239 ERROR DURING EVALUATION OF RIGHT SIDE OF ASSIGNMENT.
SDP0431 ERROR '(&00)' IN BUILTIN FUNCTION '(&01)'
SDP0468 NO NAME FOUND

/A = RENAME(‘A.B.C‘,‘A///C‘,‘NEWA///NEWC‘)
/SHOW-VARIABLE A
A = NEWA.B.NEWC

/A = RENAME(‘A.B‘,‘*.*‘,‘**‘)
/SHOW-VARIABLE A
A = AB

/A = RENAME(‘A.B‘,‘*.*‘,‘<1><1>‘)
/SHOW-VARIABLE A
A = AA

/A = RENAME(‘A.B‘,‘/./‘,‘XYZ<2>‘)
/SHOW-VARIABLE A
A = XYZB
```

REPLACE( ) Overwrite or replace substring

Domain: **String processing**

The REPLACE( ) function overwrites or replaces a substring within a string with another string. This can make the original string longer.

Format

REPLACE( )
STRING = string_expression$_1$,START = <u>1</u> / integer ,REPLACE = string_expression$_2$,SUPPRESSED-LENGTH = <u>*REPLACE-LENGTH</u> / integer

Result type

STRING

Input parameters

STRING = string_expression$_1$
Designates the string in which a substring is to be replaced.

START = <u>1</u> / integer
Designates the position from which the (input) string is to be overwritten; “integer” is a positive integer value or an arithmetic expression which is evaluated as a positive integer value.

REPLACE = string_expression$_2$
Designates the string to be inserted at the start position.

SUPPRESSED-LENGTH =
Specifies whether string_expression$_2$ overwrites or replaces parts of the (input) string.

SUPPRESSED-LENGTH = <u>*REPLACE-LENGTH</u>
Starting at the position specified with START=.. , the input string is to be overwritten with string_expression$_2$ (in the length of string_expression$_2$).

SUPPRESSED-LENGTH = digit
Specifies the number of characters which are to be suppressed and to be replaced by string_expression$_2$. Suppression is to start at the position specified with START=.. .

Result

Modified string.

Error messages

```
SDP0412   START POSITION OUT OF RANGE

SDP0413   ILLEGAL LENGTH
```

Examples

```
/A = 'ABCDEFGHIJ'
/B = REPLACE(STRING = A, REPLACE = '**')
/SHOW-VARIABLE A
A = ABCDEFGHIJ
/SHOW-VARIABLE B
B = **CDEFGHIJ

/C = 10
/B = REPLACE(STRING = A, START = C, REPLACE = 'KLMN')
/SHOW-VARIABLE B
B = ABCDEFGHIKLMN

/B = REPLACE(STRING = A, START = 0, REPLACE = '**')
SDP0412 START POSITION OUT OF RANGE
SDP0431 ERROR 'SDP0412' IN BUILTIN FUNCTION 'REPLACE'
SDP0239 ERROR DURING EVALUATION OF RIGHT SIDE OF ASSIGNMENT
/A = REPLACE(STRING = A, REPLACE = '****')
/SHOW-VARIABLE A
A = ****EFGHIJ
```

The last assignment to variable B results in error SDP0412 since an incorrect value was entered for START.

```
/WHILE (INDEX(TESTSTRING,X'00') > 0)
/    TESTSTRING = REPLACE(TESTSTRING,INDEX(TESTSTRING,X'00'),X'40')
/END-WHILE
```

Within the TESTSTRING variable, all X’00’ are replaced by blanks (X’40’).

Examples with the operand SUPPRESSED-LENGTH=..

```
/A = 'I am the king of the replace()'
/B1= 'developer'              "REPLACE"
/B2 = REPLACE (A,10,B1,4)     " 10th position is 'k' "
/SHOW-VAR B2
B2 = I am the developer of the replace()

/C1 = 'not '                  "INSERT"
/C2 = REPLACE (A,6,C1,0)      " 6th position is 't' "
/SHOW-VAR C2
C2 = I am not the king of the replace()

/D1 = 'replacement'           "OVERWRITE (like before)"
/D2 = REPLACE (A,22,D1)       " 22th position is 'r' "
/SHOW-VAR D2
D2 = I am the king of the replacement
```

RUN-PRIORITY( ) Request runtime priority

Domain: **Job information**

The RUN-PRIORITY( ) function supplies the priority level of the current job.
The supplied value can then be checked and - if necessary - the priority of the task changed.

Format

RUN-PRIORITY( )
RUN-PRIO( )

Result type

INTEGER (<integer 0..255>)

Input parameters

None

Result

Number of type INTEGER, 0 ≤ number ≤ 255

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

```
/A = RUN-PRIORITY( )
/SHOW-VARIABLE A
A = 210
```

For comparison the field PRI indicates the runtime priority for the job (output format: BS2000/OSD-BC V7.0):

```
/show-job-status
%TSN:     29XX       TYPE:    3 DIALOG   NOW:     2007-04-26.110747
%JOBNAME: BERTA      PRI:     0 210
%USERID:  USER1      JCLASS:  JCDSTD     LOGON:   2007-04-26.1053
......
```

SDF-P-VERSION( ) Request SDF-P version

Domain: **Procedure information**

The SDF-P-VERSION( ) function supplies information about the installed version of (chargeable) subsystem SDF-P or about the current version of the SDF-P-BASYS subsystem (included in the basic configuration).

Format

SDF-P-VERSION( )
FUNCTION-RANGE = <u>*STD</u> / *BASIC

Result type

STRING

Input parameters

FUNCTION-RANGE = <u>*STD</u> / *BASIC
Operand value *STD: current version of (chargeable) subsystem SDF-P.
Operand value *BASIC: current version of the SDF-P-BASYS subsystem (included in the basic configuration).

Result

Version information in the form of a string.

Error messages

No error messages

Example

```
/A = SDF-P-VERSION
/B = SDF-P-VERSION(FUNCTION-RANGE=*BASIC)

/SHOW-VARIABLE A
A = V02.4A10

/SHOW-VARIABLE B
B = V02.4A10
```

SDF-STRUCTURE-VALUE( ) Output value of structure

Domain: **String functions/test functions**

The SDF-STRUCTURE-VALUE( ) function supplies part or all of the contents of an SDF structure.

Format

SDF-STRUCTURE-VALUE( )
STRING = string_expression ,OPERAND-NAME = <u>*ROOT</u> / string_expression / arithm_expression ,ATTACHED-STRUCTURE = <u>*YES</u> / *NO

Result type

STRING (<string>)

Input parameters

STRING = string_expression
Name of the string which is to be analyzed. This is checked internally by an IS-SDF-STRUCTURE( ). Refer therefore to the description of this latter function.

OPERAND-NAME = <u>*ROOT</u> / string_expression / arithm_expression
Name of the operand or position where the value is to be found.
Only directly addressed operands are checked, or the operands on the directly addressed level. Other operands or operands on other levels must be addressed separately.

ATTACHED-STRUCTURE= <u>*YES</u> / *NO
Specifies whether the structure affected should be specified or not.

Result

The required expression in the form of a string.

Error messages

SDP0460	THE GIVEN STRING IS NOT A STRUCTURE
SDP0461	THE NUMERIC VALUE FOR THE OPERAND MUST BE GREATER THAN ZERO
SDP0462	‘(&00)‘ IS NOT A STRUCTURED-NAME
SDP0463	THE GIVEN OPERAND ‘(&00)‘ IS UNKNOWN
SDP0464	TOO MANY AMBIGUITIES FOR THE GIVEN OPERAND
SDP0465	OPERAND OF TYPE BOOLEAN NOT ALLOWED

Example 1

```
/START-SDF-A
//OPEN syssdf.myuser,type=user,mode=create
//ADD-CMD my-cmd-1,IMPL=*PROC('myproc')
//ADD-OPERAND op
//ADD-VALUE *NAME
//ADD-VALUE *KEYWORD(STAR=*MANDATORY),STRUCTURE=*YES,VALUE='PARAMETERS'
//ADD-OPERAND op1,RESULT-OPERAND-LEVEL=2
//ADD-VALUE *NAME
//ADD-OPERAND op2,RESULT-OPERAND-LEVEL=2
//ADD-VALUE *FILENAME
//CLOSE-STRUCTURE
//END
```

First, an SDF syntax file with the name SYSSDF.MYUSER is created in which the command MY-CMD-1 is defined. This command is implemented by the MYPROC procedure (see the SDF-A statement //ADD-CMD).

Contents of the MYPROC procedure:

```
/SET-PROCEDURE-OPTIONS "Procedure: myproc"
/BEGIN-PARAMETER-DECLARATION
/   DECLARE-PARAMETER op
/END-PARAMETER-DECLARATION

/value=SDF-STRUCTURE-VALUE(op,*ROOT,*NO)
/WRITE-TEXT 'root value : &value'

/value=SDF-STRUCTURE-VALUE(op,'OP1')
/WRITE-TEXT 'operand op1 value : &value'

/value=SDF-STRUCTURE-VALUE(op,'OP2')
/WRITE-TEXT 'operand op1 value : &value'

/EXIT-PROCEDURE
```

The syntax file must be activated in order to call the MY-CMD-1 command:

```
/MODIFY-SDF-OPTIONS *ADD(ADD-NANE=syssdf.myuser)
```

Call for the MY-CMD-1 command:

```
/MY-CMD-1 OP=*PARAMETERS(OP1=VALUE1,OP2=VALUE2)
```

Output

```
root value: *PARAMETERS
operand op1 value: VALUE1
operand op2 value: VALUE2
```

The command MY-CMD-1 calls the procedure MYPROC and returns the contents of the specified OP structure.

Example 2

```
/A=‘*PARAMETERS(OP1=val1(OP11=val11,OP12=val12),OP2=val2(val21,val22))‘

/B=SDF-STRUCTURE-VALUE(A,‘OP1‘,*YES)
/SHOW-VAR B
B=val1(OP11=val11,OP12=val12)

/C=SDF-STRUCTURE-VALUE(B,‘OP11‘)
/SHOW-VAR C
C=val11

/D=SDF-STRUCTURE-VALUE(B,2)
/SHOW-VAR D
D=val12
```

SEARCH-LIST-INDEX( ) Search for string in list

Domain: **String functions**

The function SEARCH-LIST-INDEX( ) searches a list variable for a string or for a regular expression which has been formed in accordance with POSIX rules. The return value indicates the number of the first list element containing this expression or search string.

Normally, if this type of operation is executed using a loop in the INDEX function, it requires a considerable amount of time. The introduction of this predefined function reduces this execution time considerably since the search is performed in a single step.

Format

SEARCH-LIST-INDEX( )
LIST-VARIABLE-NAME = string_expression$_1$,PATTERN = string_expression$_2$,BEGIN-INDEX = <u>1</u> / arithm_expression ,END-INDEX = <u>*LAST</u> / arithm_expression ,BEGIN-COLUMN = <u>1</u> / arithm_expression ,END-COLUMN = <u>*LAST</u> / arithm_expression ,PATTERN-TYPE = <u>*STRING</u> / *REGULAR-EXPRESSION ,DIRECTION = <u>*FORWARD</u> / *REVERSE

Result type

INTEGER

Input parameters

LIST-VARIABLE-NAME= string_expression$_1$
Designates the variable consisting of a list of strings.

PATTERN = string_expression$_2$
Designates the search string or regular expression for which a sequential search operation is to be performed within the list variable specified in LIST-VARIABLE-NAME.

BEGIN-INDEX =
Specifies the first list element or index at which the search is to be started.
List elements which have a smaller index are not checked. If the specified index is greater than the number of elements, the value “0” is returned, i.e. “not found”.

BEGIN-INDEX = 1
The search starts with list element 1, i.e. at the beginning of the list.

BEGIN-INDEX = arithm_expression
The search starts at the specified list element.

END-INDEX =
Specifies the last list element or index up to which the search is to be performed.
If nothing is found up to and including this index, the value “0” is returned.

END-INDEX = *LAST
The search ends with the last list element.

END-INDEX = arithm_expression
The search ends with the specified list element.

BEGIN-COLUMN =
The search operation is restricted to a certain range of columns and is to start at a specified column.
BEGIN-COLUMN indicates the character in the string at which the search for the search string specified in PATTERN is to start in the element involved.
If PATTERN-TYPE = *REGULAR-EXPRESSION is specified, the ‘^’ at the beginning of the search string is no longer part of the actual search string, but is the position exactly in front of the value specified in BEGIN-COLUMN.
The string being searched is empty if the list element contains fewer characters than specified for BEGIN-COLUMN.

BEGIN-COLUMN = 1
The search starts at column 1, i.e. the entire list element is searched.

BEGIN-COLUMN = arithm_expression
The search starts at the specified column, i.e. character, of the list element.

END-COLUMN =
The search is restricted to a certain column range and is to end at a certain column.
END-COLUMN indicates the character in the string at which the search for the string specified in PATTERN is to end in the element involved. The characters of the list element which follow the position specified in END-COLUMN are ignored during the search.
If PATTERN-TYPE = *REGULAR-EXPRESSION is specified, the “$” character at the end of the searched string corresponds to exactly the position after the value specified in END-COLUMN (i.e. this character no longer belongs to the actual search string) or to the end of the list element if the list element contains fewer characters than specified in END-COLUMN.

END-COLUMN = *LAST
The search ends at the end of the string or at the end of the list element.

END-COLUMN = arithm_expression
The search ends at the specified column or character in the list element.

PATTERN-TYPE =
Specifies the data type of the search string or regular expression being searched for.

PATTERN-TYPE = <u>*STRING</u>
The data type is STRING.

PATTERN-TYPE = *REGULAR-EXPRESSION
The data type corresponds to that of a regular expression created in accordance with POSIX rules. This replaces any string, even an empty string.
Dieser ersetzt eine beliebige, auch leere Zeichenfolge.

DIRECTION =
Designates the direction in which the search is to take place in the list variable.

DIRECTION = <u>*FORWARD</u>
The search starts at the list element specified in BEGIN-INDEX and ends at the list element specified in END-INDEX.

DIRECTION = *REVERSE
The search runs in the opposite direction, i.e. it begins at the list element specified in END-INDEX and ends at the list element specified in BEGIN-INDEX.

Result

Positive integer indicating the first list element containing the search string specified in PATTERN. This value is greater than or equal to the value specified in BEGIN-INDEX and smaller than or equal to the number of list elements in the specified variable.

0
The search string was not found.

Error messages

```
SDP0492   NULL BYTE (X'00') NOT ALLOWED IN PATTERN OPERAND AND LIST ELEMENTS

SDP0493   VALUE OF OPERANDS BEGIN-INDEX, END-INDEX, BEGIN-COLUMN AND END-
          COLUMN MUST BE GREATER THAN ZERO

SDP0494   SYNTAX ERROR IN REGULAR EXPRESSION FOR OPERAND PATTERN

SDP1008   VARIABLE/LAYOUT '(&00)' DOES NOT EXIST

SDP1096   VARIABLE '(&00)' MUST BE A LIST OF TYPE STRING OR ANY CONTAINING
          ONLY STRING VALUES
```

Example 1

```
/DECLARE-VARIABLE FILE-NEW(TYPE=*STRING), MULTIPLE-ELEMENTS=*LIST
/READ-VARIABLE *LIST(FILE-NEW), INPUT=*SYSDTA
First line in the file
Second
Third line in the file
4.
5. ........
6. in the file
7. ... in the ...
*END-OF-CMD
/MATCH = 0
/REPEAT
/   MATCH=SEARCH-LIST-INDEX('FILE-NEW',PATTERN='line', BEGIN-INDEX=MATCH+1)
/   SHOW-VARIABLE MATCH
/UNTIL (MATCH == 0)
```

Output

```
MATCH = 1
MATCH = 3
MATCH = 0
```

In this example, SEARCH-LIST-INDEX( ) is used to search a list variable for a string (‘line’) starting at a certain character. The REPEAT loop ensures that the value for BEGIN-INDEX is incremented by 1 until the entire file has been searched.
If the following expression is included in the REPEAT loop, the output is different:

Example 2

```
/MATCH=SEARCH-LIST-INDEX('FILE-NEW', -
/                          PATTERN =' in ', -
/                          BEGIN-COLUMN=7,-
/                          END-COLUMN=11)
/SHOW-VARIABLE MATCH
```

Output

```
MATCH = 7
```

In this example the list variable FILE-NEW (see example 1) is searched through for the string “ in ”. The search is limited to columns 7 up to and including column 11. Only list element 7 fulfills the search conditions, but list elements 1, 3 and 6 do not.

Example 3

```
/DECLARE-VARIABLE RECORD-LIST(TYPE=*STRING), MULTIPLE-ELEMENTS=*LIST
/READ-VARIABLE *LIST(RECORD-LIST), INPUT=*SYSDTA
WIEDEMANN BERNHARD 64528
BACHMANN MICHAEL 37214
ARTMANN HELMUT 74634
HEUBACH HUGO 97884
BACH ANDREAS 12012
KIRSCHNER ANITA 76325
*END-OF-CMD
/NAME = 'BACH'
/MATCH=SEARCH-LIST-INDEX('RECORD-LIST',
/                          PATTERN ='^&NAME. ', -
/                          PATTERN-TYPE=*REGULAR-EXPRESSION -
/                        )
/NUMBER = INTEGER(EXTRACT-FIELD(RECORD-LIST#MATCH,3))
/WRITE-TEXT '&NAME. HAS NUMBER &NUMBER.'
```

Output

```
BACH HAS NUMBER 12012
```

RECORD-LIST is searched for the name ‘BACH‘ and one hit is reported. ‘HEUBACH‘ and ‘BACHMANN‘ do not satisfy the criteria for the search string because PATTERN-TYPE = *REGULAR-EXPRESSION was specified: the first does not start with a “B” and the second does not end with a blank after ‘BACH’.

Example 4

```
/DECLARE-VARIABLE A-LIST (TYPE=*STRING), MULTIPLE-ELEMENTS=*LIST
/                 A-LIST#1  = 'ACTIVE  '
/                 A-LIST#2  = 'WAITING '
/                 A-LIST#3  = 'INACTIVE'
/                 A-LIST#4  = 'ABORTED '
/                 A-LIST#5  = 'ACTIVE  '
/                 A-LIST#6  = 'LOCKED  '
/                 A-LIST#7  = 'WAITING '
/                 A-LIST#8  = 'ACTIVE  '
/                 A-LIST#9  = 'ACTIVE  '
/                 A-LIST#10 = 'INACTIVE'
/WAITING-IDX=SEARCH-LIST-INDEX('A-LIST','WAITING',DIRECTION=*FORWARD)
/SHOW-VARIABLE WAITING-IDX
WAITING-IDX = 2
/WAITING-IDX=SEARCH-LIST-INDEX('A-LIST','WAITING',DIRECTION=*REVERSE)
/SHOW-VARIABLE WAITING-IDX
WAITING-IDX = 7
```

The “WAITING” string is searched in list variable A-LIST. The forward search reports list element 2 as a hit. The reverse search reports list element 7 as a hit.

SESSION-NUMBER( ) Request system sequence number

Domain: **System information**

The SESSION-NUMBER( ) function supplies the system sequence number of the system currently running (for example, the system sequence number can be part of the CONSLOG file name).

Format

SESSION-NUMBER( )

Result type

STRING (<string 3..3>)

Input parameters

None

Result

System sequence number in the form of a string.

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

```
/A = SESSION-NUMBER( )
/SHOW-VARIABLE A
A = 012
```

SIZE( ) Request size of complex variables

Domain: **Variable access** (variable attributes)

The SIZE( ) function requests the number of elements comprising the specified variable. SIZE( ) can be applied to arrays, lists and structures.

In conjunction with the NEXT-VARIABLE-NAME( ) function, the result of SIZE( ) can be used, for example, as a loop count when analyzing the layout of complex variables.

Format

SIZE( )
VARIABLE-NAME = string_expression

Result type

INTEGER

Input parameters

VARIABLE-NAME = string_expression
Designates a variable (array, list or structure). The variable name must be enclosed in apostrophes if it is specified directly, i.e as a literal (see the following example and the example in the description of IS-DECLARED( )).

Result

Number of elements comprising the variable “string_expression”.

0
The value “0” is returned in the following cases:
– “string_expression” does not contain an element.
– “string_expression” does not designate a complex variable, or no complex variable with this name exists.

Error message

```
SDP1101   SYNTAX ERROR IN VARIABLE NAME
```

Example

In the current task, a global variable list VARLIST for the task has already been declared and initialized. In the current procedure, VARLIST can contain exactly 10 elements.

```
/IF SIZE('VARLIST') > 10
/  WRITE-TEXT 'too many elements'
/  FOR I = *LIST(VARLIST)
/    SHOW-VARIABLE I
/  END-FOR
/  GOTO TOOMANY
/ELSE
/ COUNT = SIZE('VARLIST')
/  WHILE COUNT < 10
/    VARLIST = COUNT+1, WRITE-MODE = *EXTEND
/    COUNT = COUNT+1
/  END-WHILE
/END-IF
...
/TOOMANY: “Error handling > 10 list elements”
...
```

The size of the list variable is checked in the first line. If it contains more than 10 elements, the contents of all elements are output in a FOR loop and the procedure is continued with the error handling procedure TOOMANY.
If VARLIST does not contain more than 10 elements, the ELSE branch of the IF block is executed. This is a WHILE loop which appends elements to VARLIST until VARLIST contains precisely 10 elements. The procedure is then continued with the command which follows the END-IF command.

STATION( ) Request TIAM station name

Domain: **Environment information** (TIAM)

The STATION( ) function supplies the station name of the TIAM station from which the procedure was initiated.

Format

STATION( )

Result type

STRING(<string 1..8>)

Input parameters

None

Result

Station name in the form of a string.

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

```
/A = STATION( )
/SHOW-VARIABLE A
A = $$$06580
```

For comparison: the STATION field indicates the name of the TIAM station (output format BS2000/OSD-BC V7.0):

```
/show-job-status
%TSN:     29XX       TYPE:    3 DIALOG   NOW:     2007-04-26.110747
%JOBNAME: BERTA      PRI:     0 210
%USERID:  USER1      JCLASS:  JCDSTD     LOGON:   2007-04-26.1053
%ACCNB:   ACC01      CPU-MAX:   9999     CPU-USED:000000.6447
%STATION: $$$06580   PROC:    FIREBALL
...
```

STATION-TYPE( ) Request TIAM device type

Domain: **Environment information**

The STATION-TYPE( ) function returns the generated device type of the TIAM station from which the procedure was called.

Format

STATION-TYPE( )

Result type

STRING(<string 1...8>)

Input parameters

None

Result

Device type of the TIAM station in the form of a string.

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

```
/A = STATION-TYPE( )
/SHOW-VARIABLE A
A = DSS-9763
```

STD-CAT-ID( ) Request catalog ID

Domain: **User information**

The STD-CAT-ID( ) function supplies the ID of the pubset assigned to the current user ID as the default pubset.

The default pubset is the pubset on which the data of a user are stored, i.e. cataloged, if the user does not specify a catalog ID when creating the catalog entry.

Format

STD-CAT-ID( )

Result type

STRING (<string 1..4>)

Input parameters

None

Result

Catalog ID of up to four characters in the form of a string.

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

```
/A = STD-CAT-ID( )
/SHOW-VARIABLE A
A = 1OSN
```

STMT-SPINOFF( ) Request statement spin-off

Domain: **Procedure information**

The STMT-SPINOFF( ) function indicates whether a statement spin-off has been activated for the loaded program.

A statement spinoff is triggered when SDF statements are read by the system file SYSSTMT in a program (read statement with the CMDRST or RDSTMT macro call, see the “SDF-A” manual [16]) and an error occurs.
At the statement level, the statement spin-off can be intercepted with the //STEP statement. At the command level, the STMT-SPINOFF( ) function is the only way to query whether a statement spin-off has taken place or not.

Note

Using this function is senseless in command blocks in which return codes are interpreted like command return codes by program statements (see the BEGIN-BLOCK command, operand PROGRAM-INPUT=*MIXED-WITH-CMD(PROPAGATE-STMT-RC=*TO-CMD-RC)). The reason for this is that the value YES will never be returned because the return code processing does not differentiate between statements and commands.

Format

STMT-SPINOFF( )

Result type

STRING (YES / NO / UNDEFINED)

Input parameters

None

Result

YES
A statement spin-off has been initiated for the loaded program.

NO
No statement spin-off has been initiated for the loaded program.

UNDEFINED
No program is loaded.

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

The following procedure starts the program SDF-A and opens a syntax file for reading; the file type is unknown.

```
/DECLARE-PARAMETER NAME=SYNTAX-FILE(INITIAL-VALUE=*PROMPT)
/BEGIN-BLOCK PROGRAM-INPUT=*MIXED-WITH-CMD
/   START-SDF-A
//  OPEN-SYNTAX-FILE &(SYNTAX-FILE),TYPE=SYSTEM,MODE=READ
/   IF (STMT-SPINOFF='YES')
//      STEP
//      OPEN-SYNTAX-FILE &(SYNTAX-FILE),TYPE=GROUP(*NO),MODE=READ
/       IF (STMT-SPINOFF='YES')
//         STEP
//         OPEN-SYNTAX-FILE &(SYNTAX-FILE),TYPE=USER(*NO, *NO),MODE=READ
/       END-IF
/   END-IF
//  SHOW-STATUS
/   EXIT-PROCEDURE RESUME-PROGRAM=*YES
/END-BLOCK
```

STRING( ) Convert expression to string

Domain: **Conversion functions/string functions**

The STRING( ) function converts an INTEGER, BOOLEAN or STRING expression to data type STRING. The rules for implicit conversion apply.

Format

STRING( )
STR( )

EXPRESSION = expression

Result type

STRING

Input parameters

EXPRESSION = expression
Designates the expression to be converted.

Result

Converted expression in the form of a string.

Error messages

No error messages

Example

```
/DECLARE-VARIABLE A,TYPE=*INTEGER     "Data type: INTEGER"
/A = 30
/C = STRING(A)
/SHOW-VARIABLE C
C = 30

/D = CURRENT-TYPE('C')
/SHOW-VARIABLE D
D = *STRING    "Data type is no longer integer; it is now string"
```

SUBCODE1( ) Request subcode1

Domain: **Command return code**

The SUBCODE1( ) function supplies the error class of the current command return code, i.e. the return code of the last command which resulted in an error or which was followed by the /SAVE-RETURNCODE command.

Command return codes consist of three components: Subcode1 and Subcode2, which indicate the error class and error severity, and the maincode, which contains the seven-character error code. The functions SUBCODE1( ), SUBCODE2( ) and MAINCODE( ) are used to evaluate these components. The MSG( ) function can be used to output the message corresponding to the error code of MAINCODE( ).

SUBCODE2( ) is not available outside procedures and dialog blocks.

Format

SUBCODE1( )
SC1( )

Result type

INTEGER (<integer 0..255>)

Input parameters

None

Result

Designation of the error class in the form of an integer <integer 0..255>

0
No errors have yet occurred in the current procedure or the command whose return code was saved with /SAVE-RETURNCODE was executed without an error.

Error messages

```
SDP0428   COMMAND RETURN CODE NOT AVAILABLE IN DIALOG

SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

Error handling with SUBCODE1( )

```
/DECLARE-VARIABLE MYVAR
/...
/DECLARE-VARIABLE MYVAR    "already declared"
/SAVE-RETURNCODE
/IF ((SUBCODE1=0) AND (SUBCODE2=1))
/   WRITE-TEXT 'variable already declared'
/END-IF
```

SUBCODE2( ) Request subcode2

Domain: **Command return code**

The SUBCODE2( ) function supplies the severity of the current command return code, i.e. the return code of the last command which resulted in an error or which was followed by /SAVE-RETURNCODE.

Command return codes consist of three components: Subcode1 and Subcode2, which indicate the error class and error severity, and the maincode, which contains the seven-character error code. The functions SUBCODE1( ), SUBCODE2( ) and MAINCODE( ) are used to evaluate these components. The MSG( ) function can be used to output the message corresponding to the error code of MAINCODE( ).

SUBCODE2( ) is not available outside procedures and blocks.

Format

SUBCODE2( )
SC2( )

Result type

INTEGER (<integer 0..255>)

Input parameters

None

Result

Value of subcode2 in the form of an integer (<integer 0..255>)

Error messages

```
SDP0428   COMMAND RETURN CODE NOT AVAILABLE IN DIALOG
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

Error handling with SUBCODE2( )

```
/DECLARE-VARIABLE MYVAR
/...
/DECLARE-VARIABLE MYVAR   “already declared”
/SAVE-RETURNCODE
/IF ((SUBCODE1=0) AND (SUBCODE2=1))
/   WRITE-TEXT 'variable already declared'
/END-IF
```

SUBLIST( ) Select element from SDF list

Domain: **String functions**

The SUBLIST( ) function returns the contents of the selected element of an SDF list. An SDF list is a string which is interpreted in accordance with the syntactical rules for operand lists in commands. Evaluation of the string with this function is meaningful only if it has the format '(<element$_1$>,<element$_2$>,...<element$_n$>)', where <element$_i$> is a sequence of characters which contain no commas outside pairs of parentheses.
The IS-SDF-LIST( ) function can be used to check whether a string is a list.

Format

```
SUBLIST( )

 LIST = string_expression
,POSITION = arithm_expression
```

Result type

STRING

Input parameters

LIST = string_expression
Designates an SDF list.

POSITION =
Designates the element of the SDF list whose contents are to be output.

POSITION = arithm_expression
Designates an element of an SDF list with its element number.
“arithm_expression” must be a valid element number.

Result

The contents of the element in the form of a string.

Error messages

```
SDP0411   STRING EMPTY

SDP0447   THE GIVEN STRING IS NOT A SDF-LIST

SDP0448   THE PARAMETER “NUMBER” OUT OF RANGE
```

Example

```
/A = SUBLIST('(abc,def,jkl,uvw)',3)
/SHOW-VARIABLE A
A = jkl
```

SUBLIST-NUMBER( ) Request number of elements in SDF list

Domain: **String functions**

The SUBLIST-NUMBER( ) function provides information on how many elements an SDF list contains. An SDF list is a string which is interpreted in accordance with the syntactical rules for operand lists in commands. Evaluation of the string with this function is meaningful only if it has the format ’(<element$_1$>,<element$_2$>,...<element$_n$>)’, where <element$_i$> is a sequence of characters which contain no commas outside pairs of parentheses.
The IS-SDF-LIST( ) function can be used to check whether a string is a list.

Format

SUBLIST-NUMBER( )
LIST = string_expression

Result type

INTEGER

Input parameters

LIST = string_expression
Designates an SDF list.

Result

The number of elements in the SDF list as an integer value

Error messages

```
SDP0411   STRING EMPTY

SDP0447   THE GIVEN STRING IS NOT A SDF-LIST
```

Example

```
/A = SUBLIST-NUMBER('(abc,def,jkl,uvw)')
/SHOW-VARIABLE A
A = 4
```

SUBSTRING( ) Output substring

Domain: **String functions**

The SUBSTRING( ) function extracts a substring from the specified string. The start position of the substring and its length are determined by means of the input parameters.

Format

SUBSTRING( )
SUBSTR( )

STRING = string_expression ,START = <u>1</u> / arithm_expressionr$_1$,LENGTH = <u>*REST-LENGTH</u> / arithm_expression$_2$

Result type

STRING

Input parameters

STRING = string_expression
Designates the input string from which a substring is to be extracted.

START = <u>1</u> /arithm_expression$_1$
Designates the start position of the substring, i.e. first character in the substring; “arithm_expression$_1$” is a positive INTEGER and must be smaller than the length of the input string. The default for “arithm_expression$_1$” is 1. If “arithm_expression$_1$” does not designate a valid start position in the input string, the null string is returned.

LENGTH =
Length of the substring.

LENGTH = <u>*REST-LENGTH</u>
Implicitly designates the length of the substring as the rest length, starting at the character position specified by START (string length - “arithm_expression$_1$” + 1 = rest length).

LENGTH = arithm_expression$_2$
Explicitly designates the length of the substring; if the length entry is too large, LENGTH = *REST-LENGTH applies implicitly.

Result

Substring of the input string with the length specified by LENGTH.

Null string (”) means:
START = “arithm_expression$_1$” was not a valid start position, or LENGTH = 0 applied implicitly or explicitly.

Error messages

```
SDP0412   START POSITION OUT OF RANGE

SDP0414   WARNING: *REST-LENGTH VALUE USED FOR LENGTH OPERAND
```

Example

```
/A = 'ABCDEFGH'
/B = SUBSTRING(STRING = A, START = 2, LENGTH = 4)
/SHOW-VARIABLE B
B = BCDE

/C = 2
/B = SUBSTRING(STRING = A, START = C)
/SHOW-VARIABLE B
B = BCDEFGH

/D = 4
/B = SUBSTRING(STRING = A, LENGTH = D)
/SHOW-VARIABLE B
B = ABCD

/B = SUBSTRING(STRING = A, START = 9)
SDP0412  START POSITION OUT OF RANGE
/SHOW-VARIABLE B
B =
```

See VERIFY( ) on page 534 for another example.

SYSCMD( ) Request SYSCMD assignment

Domain: **SYSFILE information**

The SYSCMD( ) function supplies the name of the file (alternative: a library element or a list variable) assigned to the system file SYSCMD. The function can be used to choose between the SYSFILE environment of the procedure and the SYSFILE environment of the task.

Format

SYSCMD( )
SYSTEM-FILE-CONTEXT = <u>*OWN</u> / *CALLER

Result type

STRING

Input parameters

SYSTEM-FILE-CONTEXT =
Designates the SYSFILE environment, see also page 82.

SYSTEM-FILE-CONTEXT = <u>***OWN**</u>
The SYSFILE environment is that of the procedure.

SYSTEM-FILE-CONTEXT = *CALLER
The SYSFILE environment is that of the caller's task.

Result

The format of the output corresponds to the output of the /SHOW-SYSTEM-FILE-ASSIGNMENT command (see the “Commands, Vol. 1-5” manual [3]). If SYSCMD is read from a procedure (i.e. SYSCMD is assigned to a file, a library element or a list variable), then the type of procedure call is also displayed (for /INCLUDE-PROCEDURE with *INCLUDE*, for /CALL-PROCEDURE with *PROCEDURE*).

File (call type)
Name of the file that SYSCMD is assigned to.

**LIB-ELEM(library,element(version),type) (call type)*
Library element (designated by the name of the library, the name of the element with its version and the element type) that SYSCMD is assigned to.

**VAR(variable) (call type)*
List variable that SYSCMD is assigned to.

**PRIMARY*
The primary assignment applies to SYSCMD (data station in the dialog or the SPOOLIN file in batch mode).

**PRIMARY (DIALOG-BLOCK)*
The primary assignment applies to SYSCMD (like **PRIMARY* except that the query is was made in a dialog block).

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

In dialog:

```
/C = SYSCMD()
/SHOW-VARIABLE C
C = *PRIMARY
```

In the dialog block:

```
/begin-block
%BEGIN-BLOCK/a=syscmd()
%BEGIN-BLOCK/show-variable a
%BEGIN-BLOCK/end-block
A = *PRIMARY (DIALOG-BLOCK)
```

In procedures:

The file C.PROC and the list variable PROC-1 each contain the following commands:

```
/A = SYSCMD()
/SHOW-VARIABLE A
```

Calls

```
   /CALL-PROCEDURE C.PROC

   /INCLUDE-PROCEDURE *VAR(PROC-1)
```

Output:

```
   A = :2OSG:$USER1.C.PROC (PROCEDURE)

   A = *VAR(PROC-1) (INCLUDE)
```

SYSDTA( ) Request SYSDTA assignment

Domain: **SYSFILE information**

The SYSDTA( ) function supplies the name of the file (alternative: a library element or a list variable) assigned to the system file SYSDTA.

Format

SYSDTA( )

Result type

STRING

Input parameters

None

Result

The format of the output corresponds to the output of the /SHOW-SYSTEM-FILE-ASSIGNMENT command (see the “Commands, Vol. 1-5” manual [3]).

File
Name of the file that SYSDTA is assigned to.

**LIB-ELEM(library,element(version),type)*
Library element (designated by the name of the library, the name of the element with its version and the element type) that SYSDTA is assigned to.

**VAR(variable)*
List variable that SYSDTA is assigned to.

**PRIMARY*
The primary assignment applies to SYSDTA (data station in the dialog or the SPOOLIN file in batch mode).

**SYSCMD*
When SYSDTA was explicitly assigned to the system file SYSCMD.

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Examples

In the dialog:

```
/A = SYSDTA()
/SHOW-VARIABLE A
A = *PRIMARY
```

In the procedure:

The procedure C.PROC contains the following commands:

```
/A = SYSDTA()
/SHOW-VARIABLE A
```

The following line is output when the procedure is run:

```
A = *SYSCMD
```

This is the default value for SYSDTA in S procedures.

Output for various assignments:

– SYSDTA is assigned to a file

```
/ASSIGN-SYSDTA TO=TEST.INPUT-DATA.1
/A = SYSDTA()
/SHOW-VARIABLE A
A = :2OSG:$USER1.TEST.INPUT-DATA.1
```

– SYSDTA is assigned to a library element

```
/ASSIGN-SYSDTA TO=*LIB-ELEM(LIB=ASS.PLAMLIB,ELEM=TEST.DTA.1,TYPE=S)
/A = SYSDTA()
/SHOW-VARIABLE A
A = *LIB-ELEM(:2OSG:$USER1.ASS.PLAMLIB,TEST.DTA.1(*UPPER-LIMIT),S)
```

– SYSDTA is assigned to a list variable

```
/ASSIGN-SYSDTA TO=*VARIABLE(DATA-1)
/A = SYSDTA()
/SHOW-VARIABLE A
A = *VAR(DATA-1)
```

SYS-ID( ) Request system identification

Domain: **System information**

The SYS-ID( ) function supplies the system identification, i.e. the system ID of the current system.

Format

SYS-ID( )

Result type

STRING (<string 1..4>)

Input parameters

None

Result

System identification in the form of a string.

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

```
/A = SYS-ID()
/SHOW-VARIABLE A
A = 160
```

SYSLST( ) Request SYSLST assignment

Domain: **SYSFILE information**

The SYSLST( ) function supplies the name of the file (alternative: a library element or a list variable) assigned to the system file SYSLST.

Format

SYSLST( )

Result type

STRING

Input parameters

None

Result

The format of the output corresponds to the output of the /SHOW-SYSTEM-FILE-ASSIGNMENT command (see the “Commands, Vol. 1-5” manual [3]).

File
Name of the file that SYSLST is assigned to

**DUMMY*
SYSLST is assigned to a pseudo-file.

**LIB-ELEM(library,element(version),type)*
Library element (designated by the name of the library, the name of the element with its version and the element type) that SYSLST is assigned to.

**VAR(variable)*
List variable that SYSLST is assigned to.

**PRIMARY*
The primary assignment applies to SYSLST (temporary SPOOLOUT file (EAM file)).

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

Output for various assignments:

– SYSLST is assigned to a file

```
/ASSIGN-SYSLST TO=PROTOCOL.1
/A = SYSLST()
/SHOW-VARIABLE A
A = :2OSG:$USER1.PROTOCOL.1
```

– SYSLST is assigned to a dummy file

```
/ASSIGN-SYSLST TO=*DUMMY
/A = SYSLST()
/SHOW-VARIABLE A
A = *DUMMY
```

– SYSLST is assigned to a library element

```
/ASSIGN-SYSLST TO=*LIB-ELEM(LIB=ASS.PLAMLIB,ELEM=PROTOCOL.1)
/A = SYSLST()
/SHOW-VARIABLE A
A = *LIB-ELEM(:2OSG:$USER1.ASS.PLAMLIB,PROTOCOL.1(*UPPER-LIMIT),P)
```

– SYSLST is assigned to a list variable

```
/ASSIGN-SYSLST TO=*VARIABLE(PROT-1)
/A = SYSLST()
/SHOW-VARIABLE A
A = *VAR(PROT-1)
```

SYSOUT( ) Request SYSOUT assignment

Domain: **SYSFILE information**

The SYSOUT( ) function supplies the name of the file (alternative: a library element or a list variable) assigned to the system file SYSOUT.

Format

SYSOUT( )

Result type

STRING

Input parameters

None

Result

The format of the output corresponds to the output of the /SHOW-SYSTEM-FILE-ASSIGNMENT command (see the “Commands, Vol. 1-5” manual [3]).

File
Name of the file that SYSOUT is assigned to

**DUMMY*
SYSOUT is assigned to a pseudo-file.

**LIB-ELEM(library,element(version),type)*
Library element (designated by the name of the library, the name of the element with its version and the element type) that SYSOUT is assigned to.

**VAR(variable)*
List variable that SYSOUT is assigned to.

**PRIMARY*
The primary assignment applies to SYSOUT (data station in the dialog or the SPOOLOUT file (S.OUT file) in batch mode).

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

Output for various assignments:

– SYSOUT is assigned to a file

```
/ASSIGN-SYSOUT TO=OUT.LOG.1
/A = SYSOUT()
/SHOW-VARIABLE A
A = :2OSG:$USER1.OUT.LOG.1
```

– SYSOUT is assigned to a dummy file

```
/ASSIGN-SYSOUT TO=*DUMMY
/A = SYSOUT()
/SHOW-VARIABLE A
A = *DUMMY
```

– SYSOUT is assigned to a library element

```
/ASSIGN-SYSOUT TO=*LIB-ELEM(LIB=ASS.PLAMLIB,ELEM=OUT.LOG.1)
/A = SYSOUT()
/SHOW-VARIABLE A
A = *LIB-ELEM(:2OSG:$USER1.ASS.PLAMLIB,OUT.LOG.1(*UPPER-LIMIT),P)
```

– SYSOUT is assigned to a list variable

```
/ASSIGN-SYSOUT TO=*VARIABLE(LOG-1)
/A = SYSOUT()
/SHOW-VARIABLE A
A = *VAR(LOG-1)
```

SYSTEM-CALL( ) Output command source

Domain: **Procedure information**

Within a procedure which has been called as the implementor of a command, the SYSTEM-CALL( ) function determines the source of the command: i.e. whether it originates from the system syntax file or the group syntax file. For further details see the “SDF-A” manual [16].

Format

SYSTEM-CALL( )

Result type

BOOLEAN

Input parameters

None

Result

TRUE
The procedure was called by the system, i.e. it supports a command defined in a system or group syntax file by means of an SDF-A //ADD-COMMAND statement with IMPLEMENTOR=*PROCEDURE. For further details see the “SDF-A” manual [16].

FALSE
The procedure is called explicitly via the CALL-PROCEDURE or INCLUDE-PROCEDURE call or via a procedure call that is implemented as a command in the user syntax file.

Note
It is superfluous to specify /IF (SYSTEM-CALL( ) AND NOT EXPLICIT-CALL( )), because only the first check by SYSTEM-CALL( ) is necessary.

Error messages

No error messages

Example

```
/SET-PROCEDURE-OPTIONS “Procedure MYPROC”
/
/WRITE-TEXT ‘System call: &(SYSTEM-CALL)‘
/EXIT-PROCEDURE

/CALL-PROCEDURE MYPROC
System call: TRUE

/MY-COMMAND MYPROC   “Command from a user syntax file with implementation”
/                                                   “of the procedure MYPROC”
System call: FALSE

/A-GROUP-COMMAND MYPROC   "Command from group syntax file with "
/                         "implementation of the procedure MYPROC"
System call: TRUE
```

SYSTEM-INFORMATION( ) Request system information

Domain: **Environment information**

The SYSTEM-INFORMATION( ) function can be used to request system information and system parameters. One value can be queried per call.

Restrictions

The SYSTEM-INFORMATION( ) function is equivalent to the Executive macro SINF at the program level (see the manual entitled “Executive Macros” [7] for information on the SINF macro call) which is only supported for compatibility reasons. The SINF macro is not developed further and is replaced by the NSIINF and NSIOPT macros. Consequently system information and system parameters which were introduced only after this macro was replaced **cannot** be queried with the SYSTEM-INFORMATION( ) function. The values which can be queried are listed in the “Overview of possible parameter values” on page 507.

The system information and system parameters which currently exist can be requested on command level using the commands SHOW-SYSTEM-INFORMATION and SHOW-SYSTEM-PARAMETERS. These commands also support structured output to S variables (see the manual “Commands, Volume 6” [4]).

Format

SYSTEM-INFORMATION( )
SYS-INF( )

INFORMATION = string_expression

Result type

STRING

Input parameters

INFORMATION = string_expression
Designates the name of a system parameter or a system parameter. You can only specify those names that are also supported by the SINF macro call (see the overview on page 507).

Result

The item of system information or system parameter is returned in the form of a string.

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Overview of possible parameter values

The permissible parameter values are the same as the values which may be specified for the INFO operand in this macro. The system parameters and system information are listed in abbreviated form in the following two tables.
The non-privileged system parameters are described in the SHOW-SYSTEM-PARAMETERS command in the “Commands, Vol. 1-5” manual [3]. The “Introductory Guide to Systems Support” manual [8] contains a complete description of all system parameters. The system information is described in the “Executive Macros” manual [7].

System parameters

System parameter	Meaning
BLKCTRL	Default value for the file attribute BLKCTRL
BLSCOPYN	Default value for the operand COPYRIGHT in the utility routine BINDER.
BLSCOPYR	Default value for the operand COPYRIGHT in the utility routine TSOSLNK.
DEFLUID	The default value for :catid:$userid in file path names for some commands and utility routines.
DMCMAXP	Maximum number of entries in the MRS catalog of the home pubset.
DUMPCL5P	Indicates whether the privileged class 5 memory is included in the user and area dumps output by CDUMP.
DUMPSEPA	Indicates whether secret pages are included in user and system dumps.
ENCRYPT	Indicates whether passwords are encrypted internally in the system.
SECSTART	Indicates whether secure system start is active or inactive.
SECSTENF	Indicates whether system initiation is aborted if the REPs cannot be logged completely.
SHUTARCH	Indicates whether the system checks, when SHUTDOWN is initiated, whether the program ARCHIVE is still being used.
SSMLGOF1	Indicates how the spoolout of the system files SYSLST, SYSOPT, SYSOUT is executed at the end of a job.
SSMLGOF2	Indicates whether messages are output when spooling out system files.
SVC79	Indicates restrictions for the use of SVC79 (switching from the non-privileged (TU) to the privileged system status (TPR)).
TEMPFILE	The character which identifies temporary (user) files or job variables (one of the characters #, / , @ or NO).

System information

System information	**Meaning**
CONFNAME	System type (model range) e.g.: H120-S (in the old format).
CONFNAMX	System type (model range) in the new extended format, e.g.: 7.500-C40-F
CPUID	The CPU identifier. The output consists of 8 elements, each 8 bytes in length.
CPUSER	Serial number (6 digits each) of the first, second, third and fourth CPU. If a CPU does not exist, X' 000000' is entered in the corresponding field. The entries are not related to the CPU addresses.
HSIBASE	The HSI base type.
HSILINE	Additional information about the HSI CFCS3.
HSITYPE	Attributes of the current HSI type. BS2000 V11.0 supports only XS31 hardware.
HSIVM	Indicates whether this is a real or a virtual machine. Possible values: V2 The operating system is running on a virtual machine under VM2000. NV The operating system is running on a real machine.
MEMSIZE	Size of the (physical) main memory available for the software (specified in bytes).
OSAMODE	Indicates the addressing mode of the operating system.
OSID	Byte 0-7: Program name of the operating system, e.g. ' BS2V095 '. Byte 8-11: Version, e.g. ' V095' .
SYSBASE	Start address of the operating system in the virtual address space.

Example 1

```
/A = SYSTEM-INFORMATION(INFORMATION='MEMSIZE')
/SHOW-VARIABLE A
A = 1073741824
```

The size of the physical main memory that can be used by the software is output in variable A: 1,073,741,824 bytes

Example 2

Procedure for querying several pieces of system information or several system parameters:

```
/DECLARE-VARIABLE VALUE-LIST,TYPE=*STRING,-
/                 INITIAL-VALUE='(-
/BLKCTRL,DEFLUID,OSID,CONFNAMX,CPUID,HSIVM,SSMLGOF1,SSMLGOF2,TEMPFILE)'
/DECLARE-VARIABLE ACT-VALUE,TYPE=*STRING
/DECLARE-VARIABLE I,TYPE=*INTEGER
/FOR  I = *COUNTER(FROM=1,TO=SUBLIST-NUMBER(VALUE-LIST))
/  ACT-VALUE = SUBLIST(VALUE-LIST,I)
/  SHV ACT-VALUE,VALUE=C-LIT
/  WRITE-TEXT '&ACT-VALUE: &(SYSTEM-INFORMATION(ACT-VALUE))'
/  IF-BLOCK-ERROR
/     WRITE-TEXT 'CURRENTLY NO INFORMATION AVAILABLE FOR &ACT-VALUE'
/  END-IF
/END-FOR
```

Output after calling the procedure:

```
ACT-VALUE = 'BLKCTRL'
BLKCTRL: PAMKEY
ACT-VALUE = 'DEFLUID'
DEFLUID: $TSOS
ACT-VALUE = 'OSID'
OSID: M12BXS  V140
ACT-VALUE = 'CONFNAMX'
CONFNAMX: 7.500- S150-40
ACT-VALUE = 'CPUID'
CPUID:
3002000188000000301200018800000030220001880000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
ACT-VALUE = 'HSIVM'
HSIVM: V2
ACT-VALUE = 'SSMLGOF1'
SSMLGOF1: REQ-SPOOL
ACT-VALUE = 'SSMLGOF2'
SSMLGOF2: YES
ACT-VALUE = 'TEMPFILE'
TEMPFILE: #
```

TASK-MODE( ) Request task mode

Domain: **Task-specific environment information**

Supplies the mode of the current task.

Format

TASK-MODE( )

Result type

STRING (<string 2..6>)

Input parameters

None

Result

BATCH
The current task is running in batch mode, i.e. asynchronously as a batch job in the background.

DIALOG
The current task is an interactive task.

SYSTEM
The current task is a system task.

TP
The current task is running in TP mode.

Error messages

No error messages

Example

```
/A = TASK-MODE()
/SHOW-VARIABLE A
A = DIALOG
```

TIME( ) Request time

Domain: **Environment information** (calendar / time)

The TIME( ) function supplies the current time of day; the separator between the entries for hours, minutes and seconds can be freely selected.

Format

TIME( )
SEPARATOR = <u>‘:‘</u> / character ,MODE = <u>*LOCAL-TIME</u> / *UNIVERSAL-TIME

Result type

STRING (<string 8..8>)

Input parameters

SEPARATOR = <u>’:’</u> / character
Determines the separator between the individual time entries; a colon (:) is used as the default.
“character” can be any character in the form of a C literal.

MODE = <u>*LOCAL-TIME</u> / *UNIVERSAL-TIME
Determines if the time is output in the local time (LOCAL-TIME) or in universal time (UNIVERSAL-TIME).
See also the GTIME macro in the “Executive Macros” manual [7] for more information on LOCAL-TIME (LT) and UNIVERSAL-TIME (UTC).

Result

hh:mm:ss
The current time, where “hh” indicates the hours, “mm” the minutes and “ss” the seconds (the colon used as the separator can be replaced by any other character).

Error messages

No error messages

Example

Output of the local time and of the UTC time:

```
/A = TIME()
/B = TIME(MODE=*UNIVERSAL-TIME)

/SHOW-VARIABLE (A,B)
A = 10:46:51
B = 09:46:51
```

TO-C-LITERAL( ) Convert string to C literal

Domain: **Conversion functions**

The TO-C-LITERAL( ) function converts the specified string to a C literal by placing single quotes at the beginning and end of the string and doubling single quotes within the string. The FROM-C-LITERAL( ) function is the inverse of the TO-C-LITERAL( ) function.

Format

TO-C-LITERAL( )
TO-C-LIT( )

STRING = string_expression

Result type

STRING

Input parameters

STRING = string_expression
Designates the expression to be converted.

Result

String literal in the form of a string.

Error messages

No error messages

Example

```
/A = 'AB''C'
/SHOW-VARIABLE A
A = AB'C
/B = TO-C-LITERAL(STRING = A)
/SHOW-VARIABLE B
B = 'AB''C'
```

```
/ADD-PASSWORD &(TO-C-LITERAL(A))
```

The command /ADD-PASSWORD ’AB”C’ is issued.

TO-X-LITERAL( ) Convert string to X literal

Domain: **Conversion functions**

The TO-X-LITERAL( ) function converts the hexadecimal value of the specified string into the external representation: it places single quotes at the beginning and end of the string and also places an X in front of the string.

The FROM-X-LITERAL( ) function is the inverse of the TO-X-LITERAL( ) function.

Format

TO-X-LITERAL( )
TO-X-LIT( )

STRING = string_expression

Result type

STRING

Input parameters

STRING = string_expression
Designates the string to be converted.

Result

X literal in the form of a string.

Error messages

No error messages

Example

```
/T = 'HELLO'
/B = TO-X-LITERAL(T)
/SHOW-VARIABLE B
B = X'C8C5D3D3D6'
```

TRANSLATE( ) Assign result values to input values

Domain: **Conversion functions**

The TRANSLATE( ) function can be used to assign any result data to any input data. Up to ten translation operations can be specified; WHEN and THEN clauses must be present in pairs.

Algorithm: the compare strings specified in the WHEN clauses are checked consecutively to see if they match the input string. If they do match, the result string specified in the corresponding THEN clause is supplied as the result. If none of the compare strings indicated in the WHEN clauses match the input string, the string specified in the ELSE branch is supplied as the result.

Format

TRANSLATE( )
STRING = string_expression$_0$
,WHEN1 = <u>*NONE</u> / string_expression$_1$, THEN1 = <u>*NONE</u> / *SAME / string_expression$_{11}$
,WHEN2 = <u>*NONE</u> / string_expression$_2$, THEN2 = <u>*NONE</u> / *SAME / string_expression$_{12}$
,WHEN3 = <u>*NONE</u> / string_expression$_3$, THEN3 = <u>*NONE</u> / *SAME / string_expression$_{13}$
,WHEN4 = <u>*NONE</u> / string_expression$_4$, THEN4 = <u>*NONE</u> / *SAME / string_expression$_{14}$
,WHEN5 = <u>*NONE</u> / string_expression$_5$, THEN5 = <u>*NONE</u> / *SAME / string_expression$_{15}$
,WHEN6 = <u>*NONE</u> / string_expression$_6$, THEN6 = <u>*NONE</u> / *SAME / string_expression$_{16}$
,WHEN7 = <u>*NONE</u> / string_expression$_7$, THEN7 = <u>*NONE</u> / *SAME / string_expression$_{17}$
,WHEN8 = <u>*NONE</u> / string_expression$_8$, THEN8 = <u>*NONE</u> / *SAME / string_expression$_{18}$
,WHEN9 = <u>*NONE</u> / string_expression$_9$, THEN9 = <u>*NONE</u> / *SAME / string_expression$_{19}$
,WHEN10 = <u>*NONE</u> / string_expression$_{10}$, THEN10 = <u>*NONE</u> / *SAME / string_expression$_{20}$
,ELSE = <u>*NONE</u> / *SAME / string_expression$_{21}$

Result type

STRING

Input parameters

STRING = string_expression$_0$
Designates the input string to which the compare strings in the WHEN clauses are to be compared.

WHENi = *NONE
1≤ i≤ 10; designates a null string.

WHENi = string_expression$_i$
1≤ i≤ 10; designates the compare string and determines the translation conditions: if a “string_expression$_i$” matches the input string, the corresponding THEN$_i$ branch is executed, i.e. the result string designated in that branch is supplied.

THENi = *NONE
1≤ i≤ 10; designates a null string.

THENi = *SAME
1 ≤ i ≤ 10; designates the input string as the result when the input string matches the compare string.

THENi = string_expression$_j$
1≤ i≤ 10; 11≤ j≤ 20, designates the result string output as the result when the input string (STRING =) matches the compare string (WHENi =).

ELSE = *NONE
When none of the compare strings in the WHENi clauses match the input string, an empty string is returned as the result.

ELSE = *SAME
When none of the compare strings in the WHENi clauses match the input string, the input string is output as the result.

ELSE = string_expression$_{21}$
Designates the result string output when none of the compare strings in the WHENi clauses match the input string.

Result

Result string from the THENi or WHENi clause.

Error messages

```
SDP0410   INCONSISTENCY BETWEEN 'WHEN' AND 'THEN' PARAMETERS
```

Example

```
/A = 'ABC'
/C = TRANSLATE(STRING = A,-
/,WHEN1 = 'AB', THEN1 = '12'-
/,WHEN2 = 'BC', THEN2 = '23'-
/,WHEN3 = 'ABC', THEN3 = '123'-
/,ELSE = 'NO_MATCH')
/SHOW-VARIABLE C
C = 123
/ B = 'grass-green'
/ D = TRANSLATE(STRING = B
/,WHEN1 = 'gruen', THEN1 = 'green'-
/,WHEN2 = 'rot', THEN2 = 'red'-
/,WHEN3 = 'blau', THEN3 = 'blue'-
/,ELSE = 'NO_MATCH')
/SHOW-VARIABLE D
D = NO_MATCH
```

TRANSLATE-BOOLEAN( ) Check Boolean expression

Domain: **Conversion functions**

The TRANSLATE-BOOLEAN( ) function checks whether the input expression is true or false. If it is true, the value specified in the THEN clause is supplied as the result. If the input expression is not true (= false), the value specified in the ELSE clause is supplied.

Format

TRANSLATE-BOOLEAN( )
IF =expression$_1$,THEN =expression$_2$,ELSE = expression$_3$

Result type

BOOLEAN / INTEGER / STRING

Input parameters

IF = expression$_1$
Designates a BOOLEAN expression.

THEN = expression$_2$
Designates a BOOLEAN, INTEGER or STRING expression.

ELSE = expression$_3$
Designates a BOOLEAN, INTEGER or STRING expression.

Result

– String, if the expression in the THEN/ELSE clause is a string expression.

– Integer, if the expression in the THEN/ELSE clause is an arithmetic expression.

– TRUE / FALSE if the expression in the THEN/ELSE clause is a Boolean expression.

Error messages

No error messages

Example

```
/A = 6
/B = 5
/C = TRANSLATE-BOOLEAN(IF=(A > B), THEN='A greater', ELSE='B greater')
/SHOW-VARIABLE C
C = A greater

/A = 5
/B = 6
/C = TRANSLATE-BOOLEAN(IF=(A > B), THEN='A greater', ELSE='B greater')
/SHOW-VARIABLE C
C = B greater
```

TRIM( ) Remove matching characters at the beginning or end of a string

Domain: **string functions**

The TRIM( ) function removes matching characters at the beginning, end or both ends of a string.

Format

TRIM( )
STRING = string_expression ,SIDE = <u>*BOTH</u> / *LEFT / *RIGHT ,TRIM-BYTE = <u>C'␣'</u> / character

Result type

STRING

Input parameters

STRING = string_expression
Designates an expression of type STRING is to be processed.

SIDE = <u>*BOTH</u> / *LEFT / *RIGHT
The character specified in the TRIM-BYTE parameter is removed from the beginning of the string (*LEFT), from its end (*RIGHT) or from both ends (*BOTH) over and over until a different character appears.

TRIM-BYTE = <u>C'␣'</u> / character
Designates the character (as a C literal) to be removed. Default value: blank (space character). Blank is assumed when a null string (C'') is entered.

Result

String shortened form

Error messages

No error messages

Example

```
/A = '  ABC   '
/B = '  ABC000'

/A = TRIM(STRING=A)
/B = TRIM(STRING=B,SIDE=*LEFT,TRIM-BYTE=' ')
/B = TRIM(STRING=B,SIDE=*RIGHT,TRIM-BYTE='0')

/SHOW-VARIABLE
A = ABC
B = ABC
```

TSN( ) Request TSN

Domain: **Job information**

The TSN( ) function supplies the task sequence number of the current job.

Format

TSN( )

Result type

STRING (<string 4..4>)

Input parameters

None

Result

Job number in the form of a string.

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

```
/A = TSN()
/SHOW-VARIABLE A
A = 29XX
```

For the sake of comparison, the job number in the TSN field (BS2000/OSD-BC V7.0 output format):

```
/show-job-status
%TSN:     29XX      TYPE:    3 DIALOG   NOW:     2007-04-26.110747
%JOBNAME: BERTA     PRI:     0 210
%USERID:  USER1     JCLASS:  JCDSTD     LOGON:   2007-04-26.1053
%ACCNB:   ACC01     CPU-MAX:   9999     CPU-USED:000000.6447
...
```

UPPER-CASE( ) Convert lowercase letters into uppercase

Domain: **String functions/conversion functions**

The UPPER-CASE( ) function converts all lowercase letters in the specified string into uppercase letters.

The letters which are to be converted must correspond to the standard EBCDI code. There is no support for language-specific variants.

Format

UPPER-CASE( )
STRING = string_expression ,TRANSLATE = <u>*ALL</u> / *OUTSIDE-QUOTES-ONLY / *INSIDE-QUOTES-ONLY

Result type

STRING

Input parameters

STRING = string_expression
Designates the string to be converted.

TRANSLATE =
Specifies which characters are to be converted.

TRANSLATE = <u>*ALL</u>
Specifies that all characters are to be converted.

TRANSLATE = *OUTSIDE-QUOTES-ONLY
Specifies that only characters outside the apostrophes are to be converted.

TRANSLATE = *INSIDE-QUOTES-ONLY
Specifies that only characters inside the apostrophes are to be converted.

Result

A string consisting solely of uppercase letters, digits and special characters.

Error messages

No error messages

Example

```
/A = 'abcd123' // 'gHI'
/B = UPPER-CASE(STRING = A)
/SHOW-VARIABLE B
B = 'ABCD123GHI'
```

USER-IDENTIFICATION( ) Request user identification

Domain: **Job information**

The USER-IDENTIFICATION( ) function supplies the user identification of the current job, i.e. the user identification from the SET-LOGON-PARAMETERS command.

Format

USER-IDENTIFICATION( )
USER-ID( )

Result type

STRING (<string 1..8>)

Input parameters

None

Result

User identification in the form of a string.

Error message

```
SDP0435   DESIRED INFORMATION NOT AVAILABLE
```

Example

```
/A = USER-IDENTIFICATION( )
/SHOW-VARIABLE A
A = USER1
```

For the sake of comparison, the user ID in the USERID field (BS2000/OSD-BC V7.0 output format):

```
/show-job-status
%TSN:     29XX      TYPE:    3 DIALOG   NOW:     2007-04-26.110747
%JOBNAME: BERTA     PRI:     0 210
%USERID:  USER1     JCLASS:  JCDSTD     LOGON:   2007-04-26.1053
%ACCNB:   ACC01     CPU-MAX:   9999     CPU-USED:000000.6447
...
```

USER-SWITCH( ) Evaluate user switch

Domain: **Job information**

The USER-SWITCH( ) function checks the value of the specified user switch.

User switches are used, for example, to synchronize batch jobs, i.e. in background procedures. Each user ID has available to it 32 user switches, which apply for all the jobs running under the user ID. That is, user switches which are set in one job can be evaluated by a different job which is running under the same user ID. User switches are set by means of the MODIFY-USER-SWITCHES command.

Format

USER-SWITCH( )
NUMBER = number
,USER-ID = *OWN / <string 1..8>

Result type

BOOLEAN

Input parameters

NUMBER = number
0 ≤ number ≤ 31; designates the user switch to be evaluated.

USER-ID = *OWN
The caller’s own user identification.

USER-ID = <string 1..8>
Designates the user identification to which the user switch to be requested belongs.

Result

TRUE
The specified user switch is assigned the value ’ON’ and is therefore “switched on”.

FALSE
The specified user switch is assigned the value ’OFF’ and is therefore “switched off”.

Error message

```
SDP0304   OVERFLOW, NUMBER OUT OF RANGE
```

Example

In dialog:

```
/MODIFY-USER-SWITCHES ON = 1
/B = USER-SWITCH(1, USER-ID = 'US123')
/SHOW-VARIABLE B
B = TRUE
```

Within a procedure, user switches 1, 3, 5 and 8 are on, while user switches 2 and 4 are off:

```
/MODIFY-USER-SWITCHES ON = (1,3,5,8), OFF = (2,4)
```

User switches are requested later on:

```
/A = USER-IDENTIFICATION()
/IF (USER-SWITCH(2, USER-ID = A))
/   CALL-PROCEDURE C.PROC1.2
/END-IF
/...
/IF (USER-SWITCH(1, USER-ID = A))
/   CALL-PROCEDURE C.PROC1.1
/ELSE-IF USER-SWITCH(4, USER-ID = A)
/   CALL-PROCEDURE C.PROC1.4
/ELSE
/   CALL-PROCEDURE C.PROC2
/END-IF
```

User switch 2 is not set (OFF), which means that the condition in the first IF command is not met. PROC1.2 is therefore not called; instead, the command following END-IF is executed immediately.

In the second IF command, user switch 1 is checked; this switch is set (ON), which means that the condition is met. The procedure C.PROC1.1 is called.

VARIABLE-ATTRIBUTE( ) Request variable attributes

Domain: **Variable access (variable attributes)**

The VARIABLE-ATTRIBUTE( ) function supplies the value of the specified attribute for the specified variable. Attributes are the variable attributes defined with SDF-P command DECLARE-VARIABLE. Keywords for the request are normally the operand names of the command.

Detailed information on structures must be requested with ATTRIBUTE = *STRUCTURE-INFO (not ATTRIBUTE = *TYPE).

Format

```
VARIABLE-ATTRIBUTE( )
VAR-ATTR( )
--------------------------------------------------------------------------------
 VARIABLE-NAME = string_expression
,ATTRIBUTE = *TYPE / *CONTAINER / *CONTAINER-NAME / *CONTAINER-SCOPE /
                  *MULTIPLE-ELEMENTS / *SCOPE / *STRUCTURE-INFO
```

Result type

STRING

Input parameters

VARIABLE-NAME = string_expression
Designates a variable. The variable name must be enclosed in apostrophes if it is specified directly, i.e as a literal (see the following example and the last example in the description of IS-DECLARED( )).

ATTRIBUTE =
Designates a variable attribute.

ATTRIBUTE = *TYPE
Supplies the variable type.
If the variable designated by “string_expression” is a structure, only the value *STRUCTURE is supplied.

ATTRIBUTE = *CONTAINER
Supplies the type of the variable container.

ATTRIBUTE = *CONTAINER-NAME
Supplies the name of the variable container.

ATTRIBUTE = *CONTAINER-SCOPE
Supplies the scope of the variable container.

ATTRIBUTE = *MULTIPLE-ELEMENTS
Supplies the type of the complex variable.

ATTRIBUTE = *SCOPE
Supplies the scope of the variable.

ATTRIBUTE = *STRUCTURE-INFO
Supplies the attributes of the complex variable designated by “string_expression” and having the type “structure”. These attributes are defined with the operand TYPE = *STRUCTURE(DEFINITION = ...) in the DECLARE-VARIABLE command.

Result

Value of the attribute in the form of a string.

Input parameter ATTRIBUTE =	**Result**
*TYPE	'*ANY' / '*BOOLEAN' / '*INTEGER' / '*STRING' Variable type '*STRUCTURE' “string_expression” designates a structure.
*CONTAINER	'*STD' / '*VARIABLE' / '*JV' / 'composed-name' Type of variable container
*CONTAINER-NAME	'name' / “ Name of the variable container or job variable If the variable does not have a container: error message.
*CONTAINER-SCOPE	'*INCLUDE' / '*PROCEDURE' / '*TASK' Scope of the variable container If the variable does not have a container: error message.
*MULTIPLE-ELEMENTS	'*ARRAY' / '*LIST' Type of the complex variable '*NO' “string_expression” is neither an array nor a list.

Input parameter ATTRIBUTE =	**Result**
*SCOPE	'*INCLUDE' / '*PROCEDURE' / '*TASK' Scope of the variable designated with “string_expression”.
*STRUCTURE-INFO	'*BY-SYSCMD' “string_expression” designates a static structure. '*DYNAMIC' “string_expression” designates a dynamic structure. 'name' Name of the structure layout.

Error messages

```
SDP0424   NO CONTAINER ASSIGNED TO VARIABLE '(&00)'

SDP0425   BUILTIN FUNCTION VAR-ATTRIBUTES: VARIABLE NOT A STRUCTURE

SDP1007   NO VARIABLE DECLARED

SDP1101   SYNTAX ERROR IN VARIABLE NAME
```

Example 1

```
/BEGIN-STRUCTURE PERSON
...
/END-STRUCTURE
/DECLARE-VARIABLE A (TYPE = *STRUCTURE(PERSON))
...
/B = VARIABLE-ATTRIBUTE(VARIABLE-NAME = 'A', ATTRIBUTE = *TYPE)
/SHOW-VARIABLE B
B = *STRUCTURE

/B = VARIABLE-ATTRIBUTE(VARIABLE-NAME = 'A', ATTRIBUTE = *STRUCTURE-INFO)
/SHOW-VARIABLE B
B = PERSON
```

Example 2

```
/OPEN-VARIABLE-CONTAINER mycontainer,*LIB(mylibrary)
/DECLARE-VARIABLE myvar, CONTAINER= mycontainer
/A = VARIABLE-ATTRIBUTE('myvar',*CONTAINER)
/SHOW-VARIABLE A
A = MYCONTAINER

/A = VARIABLE-ATTRIBUTE('myvar',*CONTAINER-NAME)
/SHOW-VARIABLE A
A = “null string”
```

VARIABLE-TO-STRING( ) Convert variable

Domain: **Conversion functions**

The VARIABLE-TO-STRING( ) function converts an S variable of type structure into a string (for further details see section “Converting SDF command strings to S variables and vice versa” on page 180).

Format

VARIABLE-TO-STRING( )
VAR-TO-STR( )

VARIABLE-NAME = string_expression

Result type

STRING

Input parameters

VARIABLE-NAME = string_expression
Name of the S variable of type structure which is to be converted into a string. The variable name must be enclosed in apostrophes if it is specified as a literal (see the example below and the last example for IS-DECLARED()).

Result

The converted expression, in the form of a string

Error messages

```
SDP0475   VARIABLE MUST BE A STRUCTURE OR A LIST/ARRAY

SDP0476   RESULT STRING TOO LONG

SDP0477   INCORRECT SDF STRUCTURE

SDP1007   NO VARIABLE DECLARED
```

Example

```
/DECLARE-VARIABLE MYSTRUCT(TYPE=*STRUCTURE(DEF=*DYNAMIC))
/MYSTRUCT.OPERAND1.SYSSTRUC = ‘VALUE1‘
/MYSTRUCT.OPERAND1.OPERAND2 = ‘VALUE2‘
/MYSTRUCT.OPERAND1.OPERAND3#1 = ‘VALUE3‘
/WRITE-TEXT &(TO-C-LITERAL(VARIABLE-TO-STRING('MYSTRUCT')))
OPERAND1 = VALUE1(OPERAND2 =VALUE2, OPERAND3 = (VALUE3))
```

VERIFY( ) Verify strings

Domain: **String functions**

The VERIFY( ) function compares two strings and returns the position of the first character in the string (STRING) which is not contained in the search string (PATTERN). The number and the order of the characters in the two strings are ignored. The search direction can be freely selected; the returned position value is always with respect to the beginning of the first string.

Format

VERIFY( )
STRING = string_expression$_1$,PATTERN = string_expression$_2$,DIRECTION = <u>*FORWARD</u> / *REVERSE

Result type

INTEGER

Input parameters

STRING = string_expression$_1$
Designates the compare string to be searched for characters which do not exist in the search string specified for PATTERN.

PATTERN = string_expression$_2$
Designates the search string.

DIRECTION =
Designates the search direction.

DIRECTION = <u>*FORWARD</u>
The search starts at the beginning of string_expression$_1$, i.e. string_expression$_1$ is searched from left to right.

DIRECTION = *REVERSE
The search starts at the end of string_expression$_1$, i.e. string_expression$_1$ is searched from right to left.

Result

Positive integer which indicates the position of the first character in the compare string which is not contained in the search string.

0
All characters in the compare string are contained in the search string.

Error message

```
SDP0413   ILLEGAL LENGTH
```

Example 1

```
/A = '314!59'
/B = '0123456789'
/C = VERIFY(STRING = A, PATTERN = B)
/SHOW-VARIABLE C
C = 4
```

Example 2

A string is to be searched for the first non-blank character.

```
/A = 'xyz'
/B = VERIFY(STRING = A, PATTERN = '␣')
/SHOW-VARIABLE B
B = 1
```

WILDCARD( ) Search for pattern

Domain: **String functions/test functions**

The WILDCARD( ) function compares a string with a pattern string. The pattern string is subject to the general rules for patterns in BS2000 ("BS2000 wildcards" see the additional "with-wild" data type qualification on page 553), or the general rules for patterns in POSIX ("POSIX wildcards" see the additional "with-wild" data type qualification on page 554).

Format

WILDCARD( )
STRING = string_expression$_1$,PATTERN = string_expression$_2$,WILDCARD-MODE = <u>*BS2000</u> / *POSIX

Result type

BOOLEAN

Input parameters

STRING = string_expression$_1$
Designates the string to be checked.

PATTERN = string_expression$_2$
Designates the pattern.

WILDCARD-MODE = <u>*BS2000</u> / *POSIX
Specifies how wildcards are to be interpreted during replacement; either in the BS2000 wildcard syntax or in the POSIX wildcard syntax.

Result

TRUE
The checked string matches the pattern indicated in PATTERN.

FALSE
The checked string does not match the pattern indicated in PATTERN.

Error message

```
SDP0443   SYNTAX OF PATTERN IS NOT A CORRECT WILDCARD SYNTAX
```

Example

```
/A = 'This is the text to be checked. Search for umlauts.'
/B = 'ue'
/C = 'ae'
/D = WILDCARD(STRING = A, PATTERN = B)
/SHOW-VARIABLE D
D = FALSE

/B = '*ue*'
/D = WILDCARD(STRING = A, PATTERN = B)
SHOW-VARIABLE D
D = TRUE
/C = '*ae*'
/D = WILDCARD(STRING = A, PATTERN = C)
/SHOW-VARIABLE D
D = FALSE
```

X-LITERAL-TO-INTEGER( ) Convert string to integer

Domain: **Conversion functions**

The X-LITERAL-TO-INTEGER( ) function converts a string which is up to 4 bytes long to an integer. The input string can be specified as an X string or as a C string. An empty string is assigned the value 0.

If the input string consists of less than 4 characters, it is padded from left to right with X'00'.

X-LITERAL-TO-INTEGER( ) is the inverse function to INTEGER-TO-X-LITERAL( ).

Format

X-LITERAL-TO-INTEGER( )
X-LIT-TO-INT( )

STRING = string_expression

Result type

INTEGER

Input parameters

STRING = string_expression
Specifies the string up to 4 characters in length which is to be converted.

Result

Integer value

0
The string contains an empty string.

Error message

```
SDP0403   SPECIFIED STRING IS TOO LONG (MORE THAN 4 CHARACTERS)
```

Example

```
/DECLARE-VARIABLE A( TYPE= *INTEGER )
/A = X-LIT-TO-INT( X'F1F2F3F4' )
/SHOW-VARIABLE A
A = -235736076
/A = X-LIT-TO-INT( C'/ $*' )
/SHOW-VARIABLE A
A = 1631607644

/A = X-LIT-TO-INT( C'' )
/SHOW-VARIABLE A
A = 0
/A = X-LIT-TO-INT( X'' )
/SHOW-VARIABLE A
A = 0
/A = X-LIT-TO-INT( X'00' )
/SHOW-VARIABLE A
A = 0
```

15 SDF-P commands

This chapter contains descriptions of the SDF-P commands, arranged in alphabetical order. It also contains the CALL-PROCEDURE and ENTER-PROCEDURE commands for BS2000/OSD-BC V7.0 that belong to the BS2000/OSD basic configuration.

The command descriptions are preceded by three introductory sections: a description of the SDF syntax, a general description of the command return codes, and a brief description of privileges.

15.1 SDF syntax representation

The following example shows the representation of the syntax of a command in a manual. The command format consists of a field with the command name. All operands with their legal values are then listed. Operand values which introduce structures and the operands dependent on these operands are listed separately.

```
HELP-SDF                                                    Alias: HPSD

 GUIDANCE-MODE = *NO / *YES
,SDF-COMMANDS = *NO / *YES
,ABBREVIATION-RULES = *NO / *YES
,GUIDED-DIALOG = *YES (...)

   *YES(...)
     |    SCREEN-STEPS = *NO / *YES
     |   ,SPECIAL-FUNCTIONS = *NO / *YES
     |   ,FUNCTION-KEYS = *NO / *YES
     |   ,NEXT-FIELD = *NO / *YES

,UNGUIDED-DIALOG = *YES (...) / *NO

   *YES(...)
     |    SPECIAL-FUNCTIONS = *NO / *YES
     |
     |   ,FUNCTION-KEYS = *NO / *YES
```

This syntax description is valid for SDF V4.6A.The syntax of the SDF command/statement language is explained in the following three tables.

Table 2: Notational conventions

The meanings of the special characters and the notation used to describe command and statement formats are explained in table 2.

Table 3: Data types

Variable operand values are represented in SDF by data types. Each data type represents a specific set of values. The number of data types is limited to those described in table 3.

The description of the data types is valid for the entire set of commands/statements. Therefore only deviations (if any) from the attributes described here are explained in the relevant operand descriptions.

Table 4: Suffixes for data types

Data type suffixes define additional rules for data type input. They contain a length or interval specification and can be used to limit the set of values (suffix begins with *without*), extend it (suffix begins with *with*), or declare a particular task mandatory (suffix begins with *mandatory*). The following short forms are used in this manual for data type suffixes:

cat-id	cat
completion	compl
correction-state	corr
generation	gen
lower-case	low
manual-release	man
odd-possible	odd
path-completion	path-compl
separators	sep
temporary-file	temp-file
underscore	under
user-id	user
version	vers
wildcard-constr	wild-constr
wildcards	wild

The description of the ‘integer’ data type in table 4 contains a number of items in italics; the italics are not part of the syntax and are only used to make the table easier to read.
For special data types that are checked by the implementation, table 4 contains suffixes printed in italics (see the *special* suffix) which are not part of the syntax.

The description of the data type suffixes is valid for the entire set of commands/statements. Therefore only deviations (if any) from the attributes described here are explained in the relevant operand descriptions.

Metasyntax

Representation	Meaning	Examples
UPPERCASE LETTERS	Uppercase letters denote keywords (command, statement or operand names, keyword values) and constant operand values. Keyword values begin with *.	**HELP-SDF** **SCREEN-STEPS** = **<u>*NO</u>**
UPPERCASE LETTERS in boldface	Uppercase letters printed in boldface denote guaranteed or suggested abbreviations of keywords.	**GUID**ANCE-**MODE** = ***Y**ES
=	The equals sign connects an operand name with the associated operand values.	**GUID**ANCE-**MODE** = **<u>*NO</u>**
< >	Angle brackets denote variables whose range of values is described by data types and suffixes (see tables 3 and 4).	**SYNTAX-F**ILE = <filename 1..54>
<u>Underscoring</u>	Underscoring denotes the default value of an operand.	**GUID**ANCE-**MODE** = **<u>*NO</u>**
/	A slash serves to separate alternative operand values.	**NEXT-FIELD** = **<u>*NO</u>** / ***Y**ES
(…)	Parentheses denote operand values that initiate a structure.	,**UNGUID**ED-**DIA**LOG = **<u>*Y</u>**<u>ES</u> (...) / ***NO**
[ ]	Square brackets denote operand values which introduce a structure and are optional. The subsequent structure can be specified without the initiating operand value.	**SELECT** = [***BY-ATTR**IBUTES](...)

Indentation	Indentation indicates that the operand is dependent on a higher-ranking operand.	,**GUID**ED-**DIA**LOG = **<u>*Y</u>**<u>ES</u> (...)
		  <u>*Y</u><u>ES</u>(...)
		    | **SCREEN-STEPS** = **<u>*NO</u>** /
		    | ***Y**ES

Table 2: Metasyntax (part 1 of 2)

Representation	**Meaning**	**Examples**
|	A vertical bar identifies related operands within a structure. Its length marks the beginning and end of a structure. A structure may contain further structures. The number of vertical bars preceding an operand corresponds to the depth of the structure.	**SUP**PORT = ***TAPE**(...) ***TAPE(...)** | **VOL**UME = ***ANY**(...) | ***ANY**(...) | | ...
,	A comma precedes further operands at the same structure level.	**GUID**ANCE**-MODE** = ***NO** / ***Y**ES ,**SDF-COM**MANDS = ***NO** / ***Y**ES
list-poss(n):	The entry “list-poss” signifies that a list of operand values can be given at this point. If (n) is present, it means that the list must not have more than n elements. A list of more than one element must be enclosed in parentheses.	list-poss: ***SAM** / ***ISAM** list-poss(40): <structured-name 1..30> list-poss(256): ***OMF** / ***SYSLST**(...) / <filename 1..54>
Alias:	The name that follows represents a guaranteed alias (abbreviation) for the command or statement name.	**HELP-SDF** Alias: **HPSDF**

Table 2: Metasyntax (part 2 of 2)

Data types

Data type	Character set	Special rules
alphanum-name	A…Z 0…9 $, #, @	
cat-id	A…Z 0…9	Not more than 4 characters; must not begin with the string PUB
command-rest	freely selectable	
composed-name	A…Z 0…9 $, #, @ hyphen period catalog ID	Alphanumeric string that can be split into multiple substrings by means of a period or hyphen. If a file name can also be specified, the string may begin with a catalog ID in the form :cat: (see data type filename).
c-string	EBCDIC character	Must be enclosed within single quotes; the letter C may be prefixed; any single quotes occurring within the string must be entered twice.
date	0…9 Structure identifier: hyphen	Input format: yyyy-mm-dd yyyy: year; optionally 2 or 4 digits mm: month dd: day
device	A…Z 0…9 hyphen	Character string, max. 8 characters in length, corresponding to a device available in the system. In guided dialog, SDF displays the valid operand values. For notes on possible devices, see the relevant operand description.
fixed	+, - 0…9 period	Input format: [sign][digits].[digits] [sign]: + or - [digits]: 0...9 must contain at least one digit, but may contain up to 10 characters (0...9, period) apart from the sign.

Table 3: Data types (part 1 of 6)

Data type	**Character set**	**Special rules**
filename	A…Z 0…9 $, #, @ hyphen period	Input format: $[\text{:cat:}][\text{\$user.}]\left\{\begin{array}{l}\text{file}\\ \text{file(no)}\\ \text{group}\\ \\ \text{group}\left\{\begin{array}{l}\text{(*abs)}\\ \text{(+rel)}\\ \text{(-rel)}\end{array}\right\}\end{array}\right\}$:cat: optional entry of the catalog identifier; character set limited to A...Z and 0...9; maximum of 4 characters; must be enclosed in colons; default value is the catalog identifier assigned to the user ID, as specified in the user catalog. $user. optional entry of the user ID; character set is A…Z, 0…9, $, #, @; maximum of 8 characters; first character cannot be a digit; $ and period are mandatory; default value is the user's own ID. $. (special case) system default ID file file or job variable name; may be split into a number of partial names using a period as a delimiter: $\text{name}_1[.\text{name}_2[...]]$ name_i does not contain a period and must not begin or end with a hyphen; file can have a maximum length of 41 characters; it must not begin with a $ and must include at least one character from the range A...Z.

Table 3: Data types (part 2 of 6)

Data type	Character set	Special rules
filename (contd.)		#file (special case) @file (special case) # or @ used as the first character indicates temporary files or job variables, depending on system generation. file(no) tape file name no: version number; character set is A...Z, 0...9, $, #, @. Parentheses must be specified. group name of a file generation group (character set: as for “file”) $\text{group}\left\{\begin{array}{l}\text{(*abs)}\\\text{(+rel)}\\\text{(-rel)}\end{array}\right\}$ (*abs) absolute generation number (1-9999); * and parentheses must be specified. (+rel) (-rel) relative generation number (0-99); sign and parentheses must be specified.
integer	0…9, +, -	+ or -, if specified, must be the first character.
name	A…Z 0…9 $, #, @	Must not begin with 0...9.

Table 3: Data types (part 3 of 6)

Data type	Character set	Special rules
partial-filename	A…Z 0…9 $, #, @ hyphen period	Input format: [:cat:][$user.][partname.] :cat: see filename $user. see filename partname optional entry of the initial part of a name common to a number of files or file generation groups in the form: $name_1$.[$name_2$.[...]] $name_i$ (see filename). The final character of “partname” must be a period. At least one of the parts :cat:, $user. or partname must be specified.
posix-filename	A...Z 0...9 special characters	String with a length of up to 255 characters; consists of either one or two periods or of alpha-numeric characters and special characters. The special characters must be escaped with a preceding \ (backslash); the / is not allowed. Must be enclosed within single quotes if alter-native data types are permitted, separators are used, or the first character is a ?, ! or ^. A distinction is made between uppercase and lowercase.

posix-pathname	A...Z
	0...9
	special characters
	structure identifier:
	slash
		Input format: [/]$part_1$/.../$part_n$
		where $part_i$ is a posix-filename;
		max. 1023 characters;
		must be enclosed within single quotes if alter-native data types are permitted, separators are used, or the first character is a ?, ! or ^.

Table 3: Data types (part 4 of 6)

Data type	**Character set**	**Special rules**
product-version	A…Z 0…9 period single quote	Input format: [[C]’][V][m]m.naso[’] correction status release status where m, n, s and o are all digits and a is a letter. Whether the release and/or correction status may/must be specified depends on the suffixes to the data type (see suffixes without-corr, without-man, mandatory-man and mandatory-corr in table 4). product-version may be enclosed within single quotes (possibly with a preceding C). The specification of the version may begin with the letter V.
structured-name	A…Z 0…9 $, #, @ hyphen	Alphanumeric string which may comprise a number of substrings separated by a hyphen. First character: A...Z or $, #, @
text	freely selectable	For the input format, see the relevant operand descriptions.
time	0…9 structure identifier: colon	Time-of-day entry: Input format: $\left\{ \begin{array}{l} \text{hh:mm:ss} \\ \text{hh:mm} \\ \text{hh} \end{array} \right\}$ $\left. \begin{array}{ll} \text{hh:} & \text{hours} \\ \text{mm:} & \text{minutes} \\ \text{ss:} & \text{seconds} \end{array} \right\}$ Leading zeros may be omitted
vsn	a) A…Z 0…9 b) A…Z 0…9 $, #, @	a) Input format: pvsid.sequence-no max. 6 characters pvsid: 2-4 characters; PUB must not be entered sequence-no: 1-3 characters b) Max. 6 characters; PUB may be prefixed, but must not be followed by $, #, @.

Table 3: Data types (part 5 of 6)

Data type	**Character set**	**Special rules**
x-string	Hexadecimal: 00…FF	Must be enclosed in single quotes; must be prefixed by the letter X. There may be an odd number of characters.
x-text	Hexadecimal: 00…FF	Must not be enclosed in single quotes; the letter X must not be prefixed. There may be an odd number of characters.

Table 3: Data types (part 6 of 6)

Suffixes for data types

Suffix	Meaning
x..y *unit*	With data type “integer”: interval specification x minimum value permitted for “integer”. x is an (optionally signed) integer. y maximum value permitted for “integer”. y is an (optionally signed) integer. *unit* additional units. The following units may be specified: *days* *byte* *hours* *2Kbyte* *minutes* *4Kbyte* *seconds* *Mbyte* *milliseconds*
x..y *special*	With the other data types: length specification For data types catid, date, device, product-version, time and vsn the length specification is not displayed. x minimum length for the operand value; x is an integer. y maximum length for the operand value; y is an integer. x=y the length of the operand value must be precisely x. *special* Specification of a suffix for describing a special data type that is checked by the implementation. “special” can be preceded by other suffixes. The following specifications are used: *arithm-expr* arithmetic expression (SDF-P) *bool-expr* logical expression (SDF-P) *string-expr* string expression (SDF-P) *expr* freely selectable expression (SDF-P) *cond-expr* conditional expression (JV) *symbol* CSECT or entry name (BLS)
with	Extends the specification options for a data type.
-compl	When specifying the data type “date”, SDF expands two-digit year specifications in the form yy-mm-dd to: 20yy-mm-dd if yy < 60 19yy-mm-dd if yy ≥ 60
-low	Uppercase and lowercase letters are differentiated.
-path-compl	For specifications for the data type “filename”, SDF adds the catalog and/or user ID if these have not been specified.
-under	Permits underscores (_) for the data types “name” and “composed-name”.

Table 4: Data type suffixes (part 1 of 7)

Suffix	**Meaning**	
with (contd.)		
-wild(n)	Parts of names may be replaced by the following wildcards. n denotes the maximum input length when using wildcards. Due to the introduction of the data types posix-filename and posix-pathname, SDF now accepts wildcards from the UNIX world (referred to below as POSIX wildcards) in addition to the usual BS2000 wildcards. However, as not all commands support POSIX wildcards, their use for data types other than posix-filename and posix-pathname can lead to semantic errors. Only POSIX wildcards or only BS2000 wildcards should be used within a search pattern. Only POSIX wildcards are allowed for the data types posix-filename and posix-pathname. If a pattern can be matched more than once in a string, the first match is used.	
	BS2000 wildcards	Meaning
	*	Replaces an arbitrary (even empty) character string. If the string concerned starts with *, then the * must be entered twice in succession if it is followed by other characters and if the character string entered does not contain at least one other wildcard.
	Terminating period	Partially-qualified entry of a name. Corresponds implicitly to the string "./*", i.e. at least one other character follows the period.
	/	Replaces any single character.
	<s_x:s_y>	Replaces a string that meets the following conditions: – It is at least as long as the shortest string (s_x or s_y) – It is not longer than the longest string (s_x or s_y) – It lies between s_x and s_y in the alphabetic collating sequence; numbers are sorted after letters (A...Z, 0...9) – s_x can also be an empty string (which is in the first position in the alphabetic collating sequence) – s_y can also be an empty string, which in this position stands for the string with the highest possible code (contains only the characters X'FF')
	<s_1,…>	Replaces all strings that match any of the character combinations specified by s. s may also be an empty string. Any such string may also be a range specification "s_x:s_y" (see above).

Table 4: Data type suffixes (part 2 of 7)

Suffix	**Meaning**	
with-wild(n) (contd.)	-s	Replaces all strings that do not match the specified string s. The minus sign may only appear at the beginning of string s. Within the data types filename or partial-filename the negated string -s can be used exactly once, i.e. -s can replace one of the three name components: cat, user or file.
	Wildcards are not permitted in generation and version specifications for file names. Only system administration may use wildcards in user IDs. Wildcards cannot be used to replace the delimiters in name components cat (colon) and user ($ and period).	
	POSIX wildcards	Meaning
	*	Replaces any single string (including an empty string). An * appearing at the first position must be duplicated if it is followed by other characters and if the entered string does not include at least one further wildcard.
	?	Replaces any single character; not permitted as the first character outside single quotes.
	[c_x-c_y]	Replaces any single character from the range defined by c_x and c_y, including the limits of the range. c_x and c_y must be normal characters.
	[s]	Replaces exactly one character from string s. The expressions [c_x-c_y] and [s] can be combined into [s_1c_x-c_ys_2].
	[!c_x-c_y]	Replaces exactly one character not in the range defined by c_x and c_y, including the limits of the range. c_x and c_y must be normal characters. The expressions [!c_x-c_y] and [!s] can be combined into [!s_1c_x-c_ys_2].
	[!s]	Replaces exactly one character not contained in string s. The expressions [!s] and [!c_x-c_y] can be combined into [!s_1c_x-c_ys_2].

Table 4: Data type suffixes (part 3 of 7)

Suffix	**Meaning**	
with (contd.)		
wild-constr(n)	Specification of a constructor (string) that defines how new names are to be constructed from a previously specified selector (i.e. a selection string with wildcards). See also with-wild. n denotes the maximum input length when using wildcards. The constructor may consist of constant strings and patterns. A pattern (character) is replaced by the string that was selected by the corresponding pattern in the selector. The following wildcards may be used in constructors:	
	Wildcard	Meaning
	*	Corresponds to the string selected by the wildcard * in the selector.
	Terminating period	Corresponds to the partially-qualified specification of a name in the selector; corresponds to the string selected by the terminating period in the selector.
	/ or ?	Corresponds to the character selected by the / or ? wildcard in the selector.
	<n>	Corresponds to the string selected by the n-th wildcard in the selector, where n is an integer.
	Allocation of wildcards to corresponding wildcards in the selector: All wildcards in the selector are numbered from left to right in ascending order (global index). Identical wildcards in the selector are additionally numbered from left to right in ascending order (wildcard-specific index). Wildcards can be specified in the constructor by one of two mutually exclusive methods: 1. Wildcards can be specified via the global index: <n> 2. The same wildcard may be specified as in the selector; substitution occurs on the basis of the wildcard-specific index. For example: the second “/” corresponds to the string selected by the second “/” in the selector	

Table 4: Data type suffixes (part 4 of 7)

Suffix	**Meaning**
with-wild-constr(n) (contd.)	The following rules must be observed when specifying a constructor: – The constructor can only contain wildcards of the selector. – If the string selected by the wildcard <...> or [...] is to be used in the constructor, the index notation must be selected. – The index notation must be selected if the string identified by a wildcard in the selector is to be used more than once in the constructor. For example: if the selector “A/” is specified, the constructor “A<n><n>” must be specified instead of “A//”. – The wildcard * can also be an empty string. Note that if multiple asterisks appear in sequence (even with further wildcards), only the last asterisk can be a non-empty string, e.g. for “****” or “*//*”. – Valid names must be produced by the constructor. This must be taken into account when specifying both the constructor and the selector. – Depending on the constructor, identical names may be constructed from different names selected by the selector. For example: “A/*” selects the names “A1” and “A2”; the constructor “B*” generates the same new name “B” in both cases. To prevent this from occurring, all wildcards of the selector should be used at least once in the constructor. – If the constructor ends with a period, the selector must also end with a period. The string selected by the period at the end of the selector cannot be specified by the global index in the constructor specification.

Table 4: Data type suffixes (part 5 of 7)

Suffix	**Meaning**
with-wild-constr(n) (contd.)	Examples:

Selector	Selection	Constructor	New name
A//*	AB1 AB2 A.B.C	D<3><2>	D1 D2 D.CB
C.<A:C>/<D,F>	C.AAD C.ABD C.BAF C.BBF	G.<1>.<3>.XY<2>	G.A.D.XYA G.A.D.XYB G.B.F.XYA G.B.F.XYB
C.<A:C>/<D,F>	C.AAD C.ABD C.BAF C.BBF	G.<1>.<2>.XY<2>	G.A.A.XYA G.A.B.XYB G.B.A.XYA G.B.B.XYB
A//B	ACDB ACEB AC.B A.CB	G/XY/	GCXYD GCXYE GCXY. [1] G.XYC

[1] The period at the end of the name may violate naming conventions (e.g. for fully-qualified file names).

Suffix	**Meaning**
without	Restricts the specification options for a data type.
-cat	Specification of a catalog ID is not permitted.
-corr	Input format: [[C]'][V][m]m.na[''] Specifications for the data type product-version must not include the correction status.
-gen	Specification of a file generation or file generation group is not permitted.
-man	Input format: [[C]'][V][m]m.n[''] Specifications for the data type product-version must not include either release or correction status.
-odd	The data type x-text permits only an even number of characters.
-sep	With the data type “text”, specification of the following separators is not permitted: ; = ( ) < > ␣ (i.e. semicolon, equals sign, left and right parentheses, greater than, less than, and blank).
-temp-file	Specification of a temporary file is not permitted (see #file or @file under filename).

Table 4: Data type suffixes (part 6 of 7)

Suffix	**Meaning**
without (contd.)	
-user	Specification of a user ID is not permitted.
-vers	Specification of the version (see “file(no)”) is not permitted for tape files.
-wild	The file types posix-filename and posix-pathname must not contain a pattern (character).
mandatory	Certain specifications are necessary for a data type.
-corr	Input format: [[C]’][V][m]m.naso[’] Specifications for the data type product-version must include the correction status and therefore also the release status.
-man	Input format: [[C]’][V][m]m.na[so][’] Specifications for the data type product-version must include the release status. Specification of the correction status is optional if this is not prohibited by the use of the suffix without-corr.
-quotes	Specifications for the data types posix-filename and posix-pathname must be enclosed in single quotes.

Table 4: Data type suffixes (part 7 of 7)

15.2 Command return codes

All SDF-P commands supply return codes which provide the user with information on command execution. This command return code is comparable to the return code on the program level. The command return code allows users to respond to specific error situations.

The command return code consists of three parts:

- subcode1, which assigns the error situation to an error class indicating how serious the error is. The value of subcode1 is output in *decimal*. The following five error classes are defined in BS2000:
  - Class A: no error
    The value is zero. Processing can be continued normally.
  - Class B: syntax error
    The value is a number between 1 and 31. There is a syntactical error in the command input. This error should be corrected before the command is entered again.
  - Class C: internal error (system error)
    The value is 32. The input should be repeated only after the internal error has been rectified.
  - Class D: errors which belong to no other error class
    The value is a number between 64 and 127. The maincode should be evaluated to determine what should be done.
  - Class E: command cannot be executed at the moment
    The value is a number between 128 and 130. The input can be repeated without correction. The command can be executed after a waiting period. The length of the waiting period is defined as short-term, long-term and indefinite.
    Short-term is indicated by the value 128 and means that the waiting period is regarded as acceptable for interactive jobs.
    Long-term is indicated by the value 129 and means that the waiting period is regarded as acceptable for batch jobs.
    Indefinite is indicated by the value 130 and means that it is not clear whether the error can be rectified at all.
- subcode2, which can contain supplementary information on the error class.
- maincode, which corresponds to a message code and supplies specific error information. This message code can be used to output the appropriate error message via the predefined MESSAGE( ) function or using the SDF command HELP-MSG-INFORMATION (see the “Commands, Vol. 1-5” manuals [3] for information on HELP-MSG-INFORMATION).

The command return code can be requested with the predefined functions SUBCODE1( ), SUBCODE2 ( ) and MAINCODE( ).

There are separate return codes for each command. In addition to the special return codes for specific commands, there are some general return codes, which are listed below.

Notes

- Normally, execution of a command is terminated when an error is detected. If more than one error occurs, it is not possible to guarantee that the first error to occur will be the first error reported, since the order of operand checking is not guaranteed.
- The return codes and messages are guaranteed only for S procedures, not for non-S procedures or batch jobs.
- The following commands are partially or completely checked during the pre-analysis of an S procedure. Any error occurring during this pre-analysis is an error in the command during procedure preparation and the command is not executed. For this reason, there are very few error messages which actually appear during command execution. If one of these commands is generated by means of expression replacement, a context error occurs.
  The commands to which the above applies are:

BEGIN-BLOCK	EXIT-BLOCK
BEGIN-PARAMETER-DECLARATION	FOR
CYCLE	GOTO
DECLARE-PARAMETER	IF
ELSE	IF-BLOCK-ERROR
ELSE-IF	IF-CMD-ERROR
END-BLOCK	INCLUDE-BLOCK
END-FOR	REPEAT
END-IF	SET-PROCEDURE-OPTIONS
END-PARAMETER-DECLARATION	UNTIL
END-WHILE	WHILE

- If an error occurs during execution of a command which initiates a block, then this error can be processed only in an IF-BLOCK-ERROR block; this can be done only after completion of the block which started the invalid command.
- The defaults are CMD0001 for the maincode and 0 for subcode1 and subcode2.

The general return codes (i.e. the return codes which can occur for any command) are:

(SC2)	SC1	Maincode	Meaning[1]
	0	CMD0001	No error
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	130	SDP0099	No further address space available

[1] If the table also contains guaranteed messages, then "/ guaranteed messages" is added to the meaning column.

For all commands, statements and records in which expression replacement is carried out, the following return codes may appear if errors occur during expression replacement:

(SC2)	SC1	Maincode	Meaning
	1	SDP0140	Syntax error during replacement
	64	SDP0141	Semantic error during replacement

The following return code may appear for all data lines in a false context:

(SC2)	SC1	Maincode	Meaning
	64	SDP0091	Semantic error

The following return code may appear for all statements in a false context:

(SC2)	SC1	Maincode	Meaning
	64	SDP0091	Semantic error

15.3 Privileges

With a few exceptions, all commands can be called by users with any of the following privileges:

STD-PROCESSING
OPERATING
HARDWARE-MAINTENANCE
SECURITY-ADMINISTRATION
SAT-FILE-MANAGEMENT
SAT-FILE-EVALUATION

Exceptions:

– ENTER-PROCEDURE command
  Required privilege: STD-PROCESSING or HARDWARE-MAINTENANCE

– For users with the privilege SECURITY-ADMINISTRATION, SAT-FILE-MANAGEMENT or SAT-FILE-EVALUATION, use of the commands listed below is restricted to procedures:

  BEGIN-BLOCK
  BEGIN-PARAMETER-DECLARATION
  CYCLE
  DECLARE-PARAMETER
  ELSE
  ELSE-IF
  END-BLOCK
  END-FOR
  END-IF
  END-PARAMETER-DECLARATION
  END-WHILE
  EXECUTE-CMD
  EXIT-BLOCK
  FOR
  GOTO
  IF
  IF-BLOCK-ERROR
  IF-CMD-ERROR
  INCLUDE-BLOCK
  INCLUDE-PROCEDURE
  REPEAT
  SET-PROCEDURE-OPTIONS
  TRACE-PROCEDURE
  UNTIL
  WHILE

15.4 Commands

ASSIGN-STREAM
Assign S variable stream

Domain: **PROCEDURE**

Command description

The ASSIGN-STREAM command is used to assign an S variable stream for structured outputs to an (output) server that controls further processing of the variable stream.

(For further details see chapter “S variable streams” on page 187.)

Format

```
ASSIGN-STREAM

STREAM-NAME = SYSVAR / SYSMSG / SYSINF / <structured-name 1..20>
,TO = *STD / <structured-name 1..20> / *DUMMY / *SAME-AS-CALLING-PROC / *VARIABLE(...) /
        *SERVER(...)
   *VARIABLE(...)
     | VARIABLE-NAME = *NONE / <composed-name 1..255>(...)
     |    <composed-name 1.255>(...)
     |      | WRITE-MODE = *EXTEND / *PREFIX
     |,RETURN-VARIABLE-NAME = *NONE / <composed-name 1..255>(...)
     |    <composed-name 1.255>(...)
     |      | WRITE-MODE = *EXTEND / *PREFIX
     |,CONTROL-VAR-NAME = *NONE / <composed-name 1..255>(...)
     |    <composed-name 1.255>(...)
     |      | WRITE-MODE = *EXTEND / *PREFIX
     |,RET-CONTROL-VAR-NAME = *NONE / <composed-name 1..255>(...)
     |    <composed-name 1.255>(...)
     |      | WRITE-MODE = *EXTEND / *PREFIX
   *SERVER(...)
     | SERVER-NAME = <structured-name 1..30>
     |,SERVER-INFORMATION = *NONE / <c-string 1..1800>
```

Operands

STREAM-NAME = <structured-name 1..20> / SYSVAR / SYSMSG / SYSINF
Name of the S variable stream. The constant values SYSINF, SYSMSG and SYSVAR are reserved words. They must not be abbreviated.

SYSINF:	transmits structured outputs from commands and programs
SYSMSG:	transmits structured guaranteed messages
SYSVAR:	transmits structured outputs from commands, programs and structured guaranteed messages. However, it is still possible to process the different data items separately.

TO =
Specifies the server which is linked with the S variable stream.

TO = <u>*STD</u>
Default assignment.
The following table shows what values are adopted internally by the default value for TO, for the different combinations which can be formed with the STREAM-NAME operand.

STREAM-NAME=	**TO=*STD**	**Information received**
SYSINF	SYSVAR	Structured outputs from commands and programs
SYSMSG	SYSVAR	Structured guaranteed messages
SYSVAR	*DUMMY	Structured outputs from commands and programs, or structured guaranteed messages
<structured-name 1..20>	*DUMMY	User variable stream

TO = <structured-name 1..20>
Name of the user server.
Any loops in chains of S variable stream assignments will be rejected; e.g.

```
ASSIGN-STREAM S3,*DUMMY
ASSIGN-STREAM S2,S3
ASSIGN-STREAM S3,S2   →   SDP0511
```

TO = *DUMMY
No assignment.
Transmitted variables are lost. The client is informed of this by a warning.

TO = *SAME-AS-CALLING-PROC
Assigns the calling procedure’s server.
If there is no assignment in the calling procedure, the assignment is rejected and the S variable stream remains unaltered.

TO = *VARIABLE(...)
The server is SDF-P.
The transmitted variables are written into the specified S variable, or the return information is read from the specified S variables.

VARIABLE-NAME =
Specifies the S variable into which the transmitted S variable is written (for further details see also the description of TRANSMIT-BY-STREAM, page 783).

VARIABLE-NAME = *NONE
The transmitted output variable is ignored.

VARIABLE-NAME = <composed-name 1..255>(...)
Name of the S variable into which the transmitted S variable is written.
The specified S variable must be a list of structures.

WRITE-MODE =
Specifies how the assigned input list is processed.

WRITE-MODE = _*EXTEND_
The transmitted variables are appended to the assigned S variable as the last element. Variables from different transmissions can be accumulated.

WRITE-MODE = *PREFIX
The transmitted variables are added in to the assigned S variable as the first element. Variables from different transmissions can be accumulated.

RETURN-VARIABLE-NAME =
Specifies the S variable whose contents are transmitted to the remote return variable (for further details see also the description of TRANSMIT-BY-STREAM, page 783).

Note
An identical variable specification with the same WRITE-MODE for RETURN-VARIABLE-NAME and for VARIABLE-NAME is not corrected at transmission time. This means that the return variable is overwritten by the variable data.

RETURN-VARIABLE-NAME = _*NONE_
No transmission; both the local and the remote return variable remain unaltered.

RETURN-VARIABLE-NAME = <composed-name 1..255>(...)
Name of the S variable read by the transmitted return variable.
The specified S variable must be a list of structures.

WRITE-MODE =
Specifies how the return variable or the list of structures is processed.

WRITE-MODE = _*EXTEND_
The last element of the specified list is removed.
The next transmission will remove the last element left by this transmission. If the list is empty, it will be processed as for RETURN-VARIABLE-NAME = *NONE.

WRITE-MODE = *PREFIX
The first element of the specified list is removed.
The next transmission will remove the first element left by this transmission. If the list is empty, it will be processed as for RETURN-VARIABLE-NAME = *NONE.

CONTROL-VAR-NAME =
Specifies the S variable into which the transmitted control variable is written (for further details see also the description of TRANSMIT-BY-STREAM).

CONTROL-VAR-NAME = <u>*NONE</u>
The transmitted control variable is ignored.

CONTROL-VAR-NAME = <composed-name 1..255>(...)
Name of the S variable into which the transmitted control variable is written.
The specified S variable must be a list of structures.

WRITE-MODE =
Specifies how the control variable or the list of structures is processed.

WRITE-MODE = <u>*EXTEND</u>
The transmitted control variables are appended to the assigned S variable as the last element. Control variables from different transmissions can be accumulated.

WRITE-MODE = *PREFIX
The transmitted control variables are added in to the assigned S variable as the first element. Control variables from different transmissions can be accumulated.

RET-CONTROL-VAR-NAME =
Specifies the S variable from which the transmitted return control variable is read (for further details see also the description of TRANSMIT-BY-STREAM, page 783).

RET-CONTROL-VAR-NAME = <u>*NONE</u>
The return control variable is ignored, i.e. it remains unaltered.

RET-CONTROL-VAR-NAME = <composed-name 1..255>(...)
Name of the S variable from which the transmitted return control variable is read.
The specified S variable must be a list of structures.

WRITE-MODE =
Specifies how the return control variable or the list of structures is processed.

WRITE-MODE = <u>*EXTEND</u>
The last element of the specified list is removed.
The next transmission will remove the last element left by this transmission. If the list is empty, it will be processed as for RET-CONTROL-VAR-NAME = *NONE.

WRITE-MODE = *PREFIX
The first element of the specified list is removed.
The next transmission will remove the first element left by this transmission. If the list is empty, it will be processed as for RET-CONTROL-VAR-NAME = *NONE.

TO = *SERVER(...)
Links the S variable stream with the specified server.

SERVER-NAME = <structured-name 1..30>
Name of the server.

SERVER-INFORMATION =
Information which must be sent to the server: e.g. the name of the format library for FHS.

SERVER-INFORMATION = *NONE
No information must be sent to the server.

SERVER-INFORMATION = <c-string 1..1800>
Text of the message, in the form of a string.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
2	0	SDP0531	Warning returned by server; process continuing
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	CMD0216	Do not have required privilege
	64	SDP0091	Semantic error
	64	SDP0532	Server error; command rejected
	64	SDP0534	Internal server error; command terminated. Server link terminated following unexpected event or due to shortage or absence of system resources
	130	SDP0099	No further address space available

Example

See the SHOW-STREAM-ASSIGNMENT (page 750) and TRANSMIT-BY-STREAM (page 783) commands.

BEGIN-BLOCK
Initiate command block

Domain: **PROCEDURE**

Command description

Command blocks which are to be treated as a logical unit begin with the BEGIN-BLOCK command and end with the END-BLOCK command. These command blocks are also called BEGIN blocks (BEGIN block: see section “Creating the procedure body” on page 92).

The command block can be identified by a tag. This tag can also be used as a branch destination (tags: see chapter “The procedure concept in SDF-P” on page 49).

The PROGRAM-INPUT operand controls the handling of commands within inputs to programs (statements and data) as well as the handling of the return codes from program statements.

Format

```
BEGIN-BLOCK

PROGRAM-INPUT = *STD / *MIXED-WITH-CMD(...)

  *MIXED-WITH-CMD(...)
   |  PROPAGATE-STMT-RC = *STD / *TO-CMD-RC
```

Operands

PROGRAM-INPUT =
Determines whether inputs to programs (statements and data) may contain commands and controls the handling of the return codes from program statements. The setting is not valid for subsequent procedure calls.

PROGRAM-INPUT = *STD
Commands are handled as in the enclosing BEGIN block. In the first-level BEGIN block, or if there is no BEGIN block, the commands must be bracketed by a HOLD-PROGRAM and a RESUME-PROGRAM command. In other words, if the data lines are interrupted by a command (except HOLD-PROGRAM), an end-of-file condition (EOF) is generated.

PROGRAM-INPUT = *MIXED-WITH-CMD(...)
No distinction is made between statements/data records and commands, i.e. commands do not generate an end-of-file condition (EOF).

PROPAGATE-STMT-RC =
Determines whether return codes from program statements are to be interpreted as command return codes **and** whether these return codes are to trigger SDF-P error handling.

PROPAGATE-STMT-RC = <u>*STD</u>
The handling of return codes from statements and SDF-P error handling are determined by the enclosing BEGIN block. In the first-level BEGIN block, or if there is no BEGIN block, return codes from statements are ignored; error handling at command level requires the use of the predefined function STMT-SPINOFF().

PROPAGATE-STMT-RC = *TO-CMD-RC
The return codes from program statements are available as command return codes and control SDF-P error handling. Further processing does not differentiate between command return codes and return codes from program statements.
SDF-P error handling is triggered only if SUBCODE1 is not equal to zero.

Note
The predefined function STMT-SPINOFF() is of no use in this case, since it will never return the value 'YES'.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	1	SDP0223	Incorrect environment
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	130	SDP0099	No further address space available

Examples

Program statements separated from commands:

Program statements are bracketed by the commands HOLD-PROGRAM and RESUME-PROGRAM. A BEGIN block is not required.

```
/start-lms
/ hold-program
/ library = 'my-library'
/ write-text 'Elements of &library.'
/ resume-program
// show-element-attributes *library-element(&library.)
/ hold-program
/ library = 'my-second-library'
/ write-text 'Elements of &library.'
/ resume-program
// show-element-attributes *library-element(&library.)
//end
```

Program statements mixed with commands:

Commands and program statements are enclosed by a BEGIN block. Commands are automatically distinguished from program statements.

```
/begin-block program-input = *mixed-with-cmd
/ start-lms
/ library = 'my-library'
/ write-text 'Elements of &library.'
// show-element-attributes *library-element(&library.)
/ library = 'my-second-library'
/ write-text 'Elements of &library.'
// show-element-attributes *library-element(&library.)
// end
/end-block
```

Return codes from program statements

Return codes from program statements and command return codes are treated alike.

```
/begin-block program-input = *mixed-with-cmd(propagate-stmt-rc = *to-cmd-rc)
/ &* Start a program which generates statement return codes
/ start-executable-program my-new-program
// my-statement1
/ if-cmd-error &* Test statement return code
/ write-text 'Error during execution of my-statement'
/ write-text 'Maincode: &mc; Subcode1: &sc1; Subcode2: &sc2'
/ end-if
/ show-file-attributes &my-file.
/ if-cmd-error &* Test command return code
/ write-text 'Error &mc during access to my-file'
/ end-if
// my-statement2
/ save-returncode &* Save statement return code
/ if (maincode <> 'CMD0001')
/ write-text 'Warning &mc during execution of my-statement2'
/ end-if
// end
/end-block
```

Processing of the STEP statement:

The STEP statement resets the SDF-P error handling triggered by errors in statements. If the return code is needed, separate error handling must be performed in an SDF-P error block (/IF-BLOCK-ERROR or /IF-CMD-ERROR instead of //STEP).

```
/begin-block program-input = *mixed-with-cmd(propagate-stmt-rc = *to-cmd-rc)
/ &* The following program generates statement return codes
/ start-executable-program <name>
// <statement>
// step
/ &* SDF-P error handling is reset
// end
/end-block
```

Processing of the END statement:

The END statement (//END) terminates program execution as well as SDF-P error handling triggered by errors in statements, if any. If the return code is needed, error handling must be performed in an SDF-P error block that precedes the END statement (/IF-BLOCK-ERROR or /IF-CMD-ERROR).

```
/begin-block program-input = *mixed-with-cmd(propagate-stmt-rc = *to-cmd-rc)
/ &* Start a program which generates statement return codes
/ start-program <name>
// my-statement
// end
/ if-block-error
/ &* the //END statement has been processed,
/ &* the return code which is available is the command
/ &* return code of the /start-program and NOT the statement
/ &* return code of //my-statement
/ write-text 'Error &mc returned by /start-program'
/ end-if
/end-block
```

BEGIN-PARAMETER-DECLARATION
Declare procedure parameters

Domain: **PROCEDURE**

Command description

The procedure parameters are declared with the DECLARE-PARAMETER command in the procedure head. If the DECLARE-PARAMETER command is to be called several times, these calls are enclosed in a command block which begins with the BEGIN-PARAMETER-DECLARATION command and ends with the END-PARAMETER-DECLARATION command.

The BEGIN-PARAMETER-DECLARATION command is also required when the command OPEN-VARIABLE-CONTAINER is to be inserted one or more times in the procedure head. This is necessary if a procedure parameter is to be initialized by a permanent variable (for further details see section “Variable containers for permanent variables” on page 163).

Format

BEGIN-**PAR**AMETER-**DECL**ARATION

Command return codes

Return codes will be supplied by this command only when it is used outside the procedure head. Errors in the procedure head are recognized by SDF-P during pre-analysis and result in the procedure run being aborted.

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	130	SDP0099	No further address space available

BEGIN-STRUCTURE
Declare static structure

Domain: **PROCEDURE**

Command description

If a structure layout is declared, BEGIN-STRUCTURE identifies the beginning of the structure layout declaration. The structure layout must be declared before the static structures which are to correspond to it. The declaration of the structure layout is terminated with the END-STRUCTURE command.

If a static structure is declared with *BY-SYSCMD, the BEGIN-STRUCTURE command must directly follow the DECLARE-VARIABLE command in which the structure is declared. In this case, the command initiates the element declarations.

(See section “Variable declaration” on page 138 for a description of structure declarations.)

Format

```
BEGIN-STRUCTURE

NAME = *NONE / <structured-name 1..20>(...)

   <structured-name 1..20>(...)

   |  SCOPE = CURRENT / PROCEDURE / TASK(...)
   |
   |     TASK(...)
   |
   |        |  STATE = ANY / *NEW
```

Operands

NAME =
Identifies the beginning of a structure layout declaration or the beginning of the element declaration of a static structure.

NAME = *NONE
Identifies the beginning of an element declaration for a static structure which was initiated with TYPE = *STRUCTURE(*BY-SYSCMD) in the DECLARE-VARIABLE command.

NAME = <structured-name 1..20>(...)
Name of a structure layout.
NAME can be used in a DECLARE-VARIABLE command with TYPE = *STRUCTURE (DEFINITION = <structured-name 1..20>) to refer to the structure layout. The name specified there for a structure layout must match the name specified here in the NAME operand. In this manner, a structure layout can be unambiguously assigned to the structure.

SCOPE =
Defines the scope of the structure layout.

SCOPE = <u>*CURRENT</u>
In a call procedure, corresponds to the PROCEDURE entry.
In an INCLUDE procedure, CURRENT means that the structure layout is created in the current INCLUDE procedure. The structure layout is then visible in this INCLUDE procedure (and in all INCLUDE procedures on lower nesting levels).
The structure layout disappears at the (dynamic) end of the CALL or INCLUDE procedure.

SCOPE = *PROCEDURE
The structure layout is declared in the current CALL procedure. In an include procedure, the current procedure is always the calling CALL procedure.
The structure layout is visible in the current CALL procedure and in all procedures called with INCLUDE-PROCEDURE from the current CALL procedure. The layout is declared in the current CALL procedure. It is therefore retained until the end of this procedure, even if it was declared in an INCLUDE procedure called from the current CALL procedure.

SCOPE = *TASK(...)
The life of the structure layout is determined by the life of the task. The structure layout is visible in all the procedures in which no other structure with the same name and a different scope (i.e. *CURRENT or *PROCEDURE) has been declared.

STATE = <u>*ANY</u>
If a structure layout of this name already exists in the task, the existing structure layout is used. A new structure layout is not created. In such multiple declarations, the rule is that the structure layout declared here must match the existing layout. If a structure layout with this name does not yet exist in the task, a new structure layout is defined.

STATE = *NEW
The structure layout cannot be present in the task. A new structure layout is declared.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available

Example

See the DECLARE-ELEMENT command, page 594.

CALL-PROCEDURE
Start command sequence

Domain: **PROCEDURE**

This command is a component part of BS2000/OSD-BC (status of description: V7.0).
In contrast to the SDF-P commands, the caller requires the STD-PROCESSING or HARDWARE-MAINTENANCE privilege.

Command description

The CALL-PROCEDURE command starts a stored command sequence (procedure). During processing, symbolic parameters contained in the sequence are replaced by the values specified in the command call (PROCEDURE-PARAMETERS operand).

S procedures:

- Current parameters may be transferred as variables; these are also used by the procedure to return output values.
- Current parameters may be transferred as positional parameters or as keyword parameters. The sequence of positional parameters corresponds to the dynamic sequence of the DECLARE-PARAMETER commands; the names of keywords correspond to the names of formal procedure parameters. Keywords may be abbreviated as long as they remain unequivocal.
- Logging is set in the command call; the same applies to the specification as to whether an already loaded program may be unloaded or not.

Procedures can be stored as:

- a cataloged SAM or ISAM file (even a temporary one) with records of variable length
- a type J or SYSJ element in a PLAM library
- an S variable of the “list” type

Procedure formats:

- text procedure
  The S procedure is in its original text format. The full SDF-P functionality is available only if the chargeable SDF-P subsystem is loaded when the procedure is called. In libraries, element type J should be used for text procedures.
- object procedure
  An S procedure in text format has been translated to object format with the COMPILE-PROCEDURE command. An object procedure can utilize the full functionality of SDF-P (apart from the COMPILE-PROCEDURE command) regardless of whether the SDF-P subsystem is currently available or not. In libraries, element type SYSJ (the default for COMPILE-PROCEDURE) should be used for object procedures.

Format

```
CALL-PROCEDURE                                                             Alias: CL / CLP

FROM-FILE = <filename 1..54 without-gen> / *LIBRARY-ELEMENT(...) / *VARIABLE(...)
  *LIBRARY-ELEMENT(...)
     | LIBRARY = <filename 1..54 without-gen>
     | ,ELEMENT = <composed-name 1..64>(...)
     |    <composed-name 1..64>(...)
     |       | VERSION = *HIGHEST-EXISTING / <composed-name 1..24>
     | ,TYPE = *STD / *BY-LATEST-MODIFICATION / <alphanum-name 1..8>
  *VARIABLE(...)
     | VARIABLE-NAME = <composed-name 1..255>
,PROCEDURE-PARAMETERS = *NO / <text 0..1800 with-low>
,LOGGING = *PARAMETERS(...) / YES / *NO /
  *PARAMETERS(...)
     | CMD = *BY-PROC-TEST-OPTION / *YES / *NO
     | ,DATA = *BY-PROC-TEST-OPTION / *YES / *NO
,UNLOAD-ALLOWED = *YES / *NO
,EXECUTION = *YES / *NO
```

Operands

FROM-FILE = <filename 1..54 without-gen> / *LIBRARY-ELEMENT(...) / *VARIABLE(...)
Name of the procedure file.

FROM-FILE = *LIBRARY-ELEMENT(...)
The procedure is stored in a PLAM library element.

LIBRARY = <filename 1..54 without-gen>
Name of the PLAM library containing the procedure file as an element (type J or SYSJ: see the TYPE operand).

ELEMENT = <composed-name 1..64>(...)
Name of the element.

VERSION = *HIGHEST-EXISTING / <composed-name 1..24>
Version of the library element. The default value is HIGHEST-EXISTING, i.e. the procedure is taken from the element with the highest version.

TYPE = *STD / *BY-LATEST-MODIFICATION / <alphanum-name 1..8>
Designates the element type the procedure file is stored under in the PLAM library.

TYPE = *STD
The procedure file can be stored as an element of type SYSJ or J.
The specified element is first searched for among the type SYSJ elements.
If it is not found there, the search proceeds to the type J elements.

A non-S procedure can only be a type J element.
An S procedure may be either a text procedure (original text format) or an object procedure (compiled object format). To simplify maintenance of the two formats in a library, text procedures should be stored as type J elements, object procedures as type SYSJ elements. The COMPILE-PROCEDURE command (part of the chargeable SDF-P subsystem) by default generates an object procedure of type SYSJ (default) from a text procedure of type J.
If this convention is followed, specifying TYPE=*STD (the default value) ensures that object procedures will be given precedence over text procedures.

TYPE = *BY-LATEST-MODIFICATION
The procedure file can be stored as an element of type SYSJ or J.
If the specified element exists both as type SYSJ and as type J, the element most recently modified will be called. If the time stamp is identical, the type SYSJ element will be called.
Specifying TYPE=*BY-LATEST-MODIFICATION ensures that the most up-to-date element will be called, typically during the debugging phase when a procedure is being written or modified.

TYPE = <alphanum-name 1..8>
The procedure file will be searched among elements of the specified type only.

FROM-FILE = *VARIABLE(...)
The procedure is stored in an S variable of the “list” type.

VARIABLE-NAME = <composed-name 1..255>
Name of the S variable.

PROCEDURE-PARAMETERS = *NO / <text 0..1800 with-low>
Defines the current procedure parameters; the parameters must be enclosed in parentheses.
See section “Passing procedure parameters” on page 106 for more details about procedure parameters.

LOGGING = <u>*PARAMETERS</u>(...) / *YES / *NO
This controls logging of procedure execution.
The LOGGING operand is ignored when *non-S procedures* are called, since in this case logging can only be declared in the procedure head (see the LOGGING operand in the BEGIN-PROCEDURE command).
When an S procedure is logged, every procedure line that is processed is output with the line number and procedure level prefixed to it.

See section “Setting the logging” on page 84 for more details about logging.

LOGGING = <u>*PARAMETERS</u>(...)
Logging can be set separately for command/statement lines and for data lines.

CMD = <u>*BY-PROC-TEST-OPTION</u> / *YES / *NO
This specifies whether commands are to be logged. The default value is BY-PROC-TEST-OPTION, i.e. no logging (equivalent to *NO) or the value selected as the default by the user with the MODIFY-PROC-TEST-OPTIONS command (component of the chargeable SDF-P subsystem).

DATA = <u>*BY-PROC-TEST-OPTION</u> / *YES / *NO
This specifies whether data lines are to be logged. The default value is BY-PROC-TEST-OPTION, i.e. no logging (equivalent to *NO) or the value selected as the default by the user with the MODIFY-PROC-TEST-OPTIONS command (component of the chargeable SDF-P subsystem).

UNLOAD-ALLOWED = <u>*YES</u> / *NO
This specifies whether a program that is loaded when the procedure is called may be unloaded.
Protection against unloading is guaranteed *only* for unloading by means of the commands LOAD-/START-EXECUTABLE-PROGRAM (or LOAD-/START-PROGRAM) and CANCEL-PROGRAM.
The specification YES is ignored if the procedure is called from a procedure for which UNLOAD-ALLOWED=*NO was declared.

EXECUTION = <u>*YES</u> / *NO
This specifies whether the procedure is merely to be analyzed for test purposes or whether it is also to be executed.
Only EXECUTION=*YES may be specified for *non-S procedures*.
Testing is possible via the MODE operand of the MODIFY-SDF-OPTIONS command.

Command return codes

The following command return codes can only be returned if the called procedure does not supply any command return code itself (e.g. EXIT-PROCEDURE not executed due to an error).
Command return codes whose maincode begins with “SSM” can only be returned when a non-S procedure is called.
Command return codes whose maincode begins with “SDP” can only be returned when an S procedure is called.

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
2	0	SSM2058	Protocol type error
2	0	SSM2065	EOF on procedure file, /END-PROC simulated
	1	SSM2036	Incomplete operand
	1	SSM2054	Symbolic operand error
	1	SSM2055	Symbolic operand error in /BEGIN-PROC
	1	SDP0138	Error in pre-analysis of text procedure, or object procedure invalid
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0093	Non-S procedure can only be type J element
	64	SDP0144	Error on parameter transfer
	64	SSM2052	DMS error (Open error)
	64	SSM2053	Not a SAM/ISAM file or file does not begin with /BEGIN-PROC or /PROC
	64	SSM2056	/CALL-PROC and /BEGIN-PROC parameters incompatible
	64	SSM2061	Error on accessing library element
	64	SSM2064	Procedure file cannot be fetched by remote processor
	130	SDP0099	No further address space available
xx	xx	xxxxxxx	Other return codes from the called procedure

CLOSE-VARIABLE-CONTAINER
Close variable container

Domain: **PROCEDURE**

Command description

The CLOSE-VARIABLE-CONTAINER command closes the specified variable containers.

Note

If a variable container is closed before its scope ceases to exist, the variables remain declared but can no longer be accessed. Every access attempt is rejected with error message SDP1030.

Format

CLOSE-VARIABLE-CONTAINER
CONTAINER-NAME = <composed-name 1..64 with-wild(80)> / list-poss(2000): <composed-name 1..64>

Operands

CONTAINER-NAME =
Name of the variable container.

CONTAINER-NAME = list-poss(2000): <composed-name 1..64>
List of the names of the variable containers.

CONTAINER-NAME = <composed-name 1..64 with-wild(80)>
Variable container which matches a specified search pattern.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	CMD0216	Do not have required privilege
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available

Example

See the SHOW-VARIABLE-CONTAINER-ATTR command, page 776.

COMPILE-PROCEDURE
Compile procedure

Domain: **PROCEDURE**

Command description

The COMPILE-PROCEDURE command converts an S procedure into a compiled procedure, i.e. into an intermediate format that can be used in environments in which the SDF-P subsystem is not available.

The full functional scope of SDF-P can be used in compiled procedures. This also holds true if these procedures are started in an environment containing only SDF-P-BASYS.

Notes

- The COMPILE-PROCEDURE command is part of the SDF-P subsystem. It is rejected if the subsystem has not been loaded (even if it is included in a compiled procedure).
- It is not possible to specify wildcards for the FROM-FILE and TO-FILE operands. It is, however, possible to specify both files and library elements.
- Inconsistent specifications for input/output are considered semantic errors, i.e. the appropriate error handling routine will be initiated.
- It is only possible to send error messages to SYSOUT. These messages are identical to those for a procedure called with CALL-PROCEDURE with the exception of the following additional compiler-specific messages:
  ```
  SDP1300 PROCEDURE COMPILER VERSION ‘(&00)’ STARTED
  SDP1301 PROCEDURE COMPILER TERMINATED NORMALLY
  SDP1302 PROCEDURE COMPILER TERMINATED ABNORMALLY
  ```
- COMPILE-PROCEDURE can also be used to convert S procedures which do not contain any chargeable SDF-P functions, or only include a few such functions, and which run on installations without the SDF-P subsystem. These installations must, however, include V2.0B of SDF-P-BASYS.

Format

```
COMPILE-PROCEDURE

FROM-FILE = <filename 1..54 without-gen> / *LIBRARY-ELEMENT(...)
  *LIBRARY-ELEMENT(...)
   |  LIBRARY = <filename 1..54 without-gen>
   | ,ELEMENT = <composed-name 1..64>(...)
   |    <composed-name 1..64>(...)
   |     |  VERSION = *HIGHEST-EXISTING / *UPPER-LIMIT / <composed-name 1..24>
   | ,TYPE = J / <alphanum-name 1..8>
,TO-FILE = <filename 1..54 without-gen> / *LIBRARY-ELEMENT(...) / *DUMMY
  *LIBRARY-ELEMENT(...)
   |  LIBRARY = *SAME / <filename 1..54 without-gen>
   | ,ELEMENT =  *SAME(...) / <composed-name 1..64>(...)
   |    *SAME(...)
   |     |  VERSION = *SAME / *UPPER-LIMIT / *INCREMENT / *HIGHEST-EXISTING /
   |     | <composed-name 1..24>
   |    <composed-name 1..64>(...)
   |     |  VERSION = *SAME / *UPPER-LIMIT / *INCREMENT / *HIGHEST-EXISTING /
   |     | <composed-name 1..24>
   | ,TYPE = SYSJ / <alphanum-name 1..8>
```

Operands

FROM-FILE =
Designates the source procedure.

FROM-FILE = <filename 1..54 without-gen>
Name of the procedure file.

FROM-FILE = *LIBRARY-ELEMENT(...)
The procedure is stored in a PLAM library.

LIBRARY = <filename 1..54 without-gen>
Name of the PLAM library containing the procedure.

ELEMENT = <composed-name 1..64>(...)
Name of the element.

VERSION =
Specifies the version number of the element.

VERSION = *HIGHEST-EXISTING
Selects the highest existing version number.

VERSION = *UPPER-LIMIT
Selects the highest possible version number.

VERSION = <composed-name 1..24>
Selects the specified version number.

TYPE= J / <alphanum-name 1..8>
Element type. The default element type is J.

TO-FILE =
Specifies where the compiled procedure is to be stored.

TO-FILE =<filename 1..54 without-gen>
Name of the file in which the compiled procedure is to be stored.

TO-FILE = *DUMMY
No procedure need be compiled. The command merely checks the procedure (see CALL-PROCEDURE EXECUTION = *NO).

TO-FILE = *LIBRARY-ELEMENT(...)
The compiled procedure is stored in a PLAM library.

LIBRARY = *SAME / <filename 1..54 without-gen>
Name of the PLAM library in which the compiled procedure is to be stored. The default value is the library for the source procedure.

ELEMENT =
Name of the element for the compiled procedure.

ELEMENT = *SAME(...)
The name of the element is the same as in the source procedure.

VERSION =
Specifies the version number of the element (only for S procedures).

VERSION = *SAME
Selects the same version number as in the source procedure.

VERSION = *UPPER-LIMIT
Selects the highest possible version number.

VERSION = *INCREMENT
The version number of the element is increased.

VERSION = *HIGHEST-EXISTING
Selects the highest existing version number.

VERSION = <composed-name 1..24>
Selects the specified version number.

ELEMENT = <composed-name 1..64>(...)
Name of the element.

VERSION =
Specifies the version number of the element (only for S procedures).

VERSION = *SAME
Selects the same version number as in the source procedure.

VERSION = *UPPER-LIMIT
Selects the highest possible version number.

VERSION = *INCREMENT
The version number of the element is increased.

VERSION = *HIGHEST-EXISTING
Selects the highest existing version number.

VERSION = <composed-name 1..24>
Selects the specified version number.

TYPE= SYSJ / <alphanum-name 1..8>
Element type. The default element type is SYSJ.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	1	SDP0138	Error during procedure preanalysis Guaranteed message: SDP0138
	1	SDP0223	Incorrect environment
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available

Example

Installation A: SDF-P has already been started

```
/ASSIGN-SYSLST TO-FILE=THE-RESULT-LISTING
/MODIFY-JOB-OPTIONS LISTING=*YES
/
/"AN ERRORED PROCEDURE IS COMPILED"
/COMPILE-PROCEDURE *LIB(LIB=THE-PROC-LIB,EL=THE-ERRONEOUS-PROC),-
/                  TO-FILE=*LIB-ELEMENT
SDP1300 PROCEDURE COMPILER VERSION 'V2.4A20' STARTED
SDP0201 INVALID BLOCK-CLOSING COMMAND USED
:
SDP1302 PROCEDURE COMPILER TERMINATED ABNORMALLY
/
/MODIFY-JOB-OPTIONS LISTING=*NO
/ASSIGN-SYSLST TO-FILE=*PRIMARY
/"LISTING: THE-RESULT-LISTING"
/
/...

/"AFTER CORRECTION"
/COMPILE-PROCEDURE *LIB(LIB=THE-PROC-LIB,EL=THE-CORRECT-PROC),-
/                  TO-FILE=*LIB-ELEMENT
SDP1300 PROCEDURE COMPILER VERSION 'V2.4A20' STARTED
SDP1301 PROCEDURE COMPILER TERMINATED NORMALLY
```

Installation B: SDF-P has not been started:

```
/CALL-PROCEDURE *LIB(LIB=THE-PROC-LIB,EL=THE-CORRECT-PROC)
%          1  1 /SET-PROCEDURE-OPTIONS
%          2  1 /DECLARE-VARIABLE A(TYPE=INTEGER,INITIAL-VALUE=1)
%          3  1 /WHILE (A < 3)
%          4  1 /IF (NOT IS-CAT-FILE ('MYFILE.1'))
%          4  1 /END-IF
%          5  1 /A=A+1
%          6  1 /END-WHILE
%          4  1 /IF (NOT IS-CAT-FILE ('MYFILE.2'))
%          4  1 /CREATE-FILE MYFILE.2
%          4  1 /END-IF
%          5  1 /A=A+1
%          6  1 /END-WHILE
%             1 /EXIT-PROCEDURE ERROR=*NO
```

CYCLE
Terminate loop pass

Domain: **PROCEDURE**

Command description

The CYCLE command can be called in loop blocks (FOR, WHILE, REPEAT block). It then terminates the current loop pass and resumes procedure execution by executing the terminating command in the loop block (END-FOR, END-WHILE, UNTIL). The loop condition is then rechecked and, if necessary, the next loop pass started (see also section “Branch to end of loop” on page 103).

Execution of the CYCLE command can be made subject to a condition.

Format

CYCLE
BLOCK = ***LAST** / ***ALL** / <structured-name 1..255> ,**COND**ITION = ***NONE** / <text 1..1800 with-low *bool-expr*>

Operands

BLOCK =
Designates the loop or loop block.

BLOCK = *LAST
Designates the next higher loop block; procedure execution is continued with the next loop terminating command.

BLOCK = *ALL
In the event of nested loops, designates the outermost loop block; procedure execution is continued with the last loop terminating command.

BLOCK = <structured-name 1..255>
Name of the loop to be terminated; the loop name is equivalent to the tag in the command call for the loop starting command.

CONDITION =
Defines a condition for execution of the CYCLE command.

CONDITION = *NONE
Command execution is not subject to any condition.

CONDITION = <text 1..1800 with-low *bool-expr*>
The CYCLE command is not executed unless the specified Boolean expression is “TRUE”.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	1	SDP0223	Incorrect environment
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error (incorrect expression)
	130	SDP0099	No further address space available

Example

Example 1

```
/LOOP: WHILE (COND < 9)
:
:
/IF (INP='*SKIP')
/WR-TEXT 'Element is skipped'
/CYCLE BLOCK=LOOP
/END-IF
:
:
/END-WHILE       "This command is executed after CYCLE"
```

Example 2

```
/J=0
/FOR I=(’line 1’,’line 2’,’line 3’,’line 4)
/J=J+1
/CYCLE BLOCK=*LAST,CONDITION=(J=3)
/SHOW-VARIABLE I
/END-FOR
```

Output:
```
I = line 1
I = line 2
I = line 4
```

The third list element (I=3) is not evaluated, i.e. execution of the loop is terminated and execution of the procedure continues at END-FOR.

DECLARE-CONSTANT
Declare variable with constant value

Domain: **PROCEDURE**

Command description

The DECLARE-CONSTANT command is used to declare one or more variables and to assign them a constant value, thus protecting these values from being overwritten. Variables with a constant value are treated in much the same way as ordinary variables. You cannot, however, modify their value using SET-VARIABLE or remove the value using FREE-VARIABLE.

Format

```
DECLARE-CONSTANT
--------------------------------------------------------------------------
 VARIABLE-NAME = list-poss(2000): <structured-name 1..20>(...)

   <structured-name 1..20>(...)

     |  VALUE = <text 0..1800 with-low expr>
     |
     | ,TYPE = *ANY / *STRING / *INTEGER / *BOOLEAN

,SCOPE = *CURRENT(...) / *PROCEDURE(...) / *TASK(...)

   *CURRENT(...)

     |  IMPORT-ALLOWED = *NO / *YES

   *PROCEDURE(...)

     |  IMPORT-ALLOWED = *NO / *YES

   *TASK(...)

     |  STATE = *ANY / *NEW / *OLD

,CONTAINER = *STD / <composed-name 1..64> / *VARIABLE(...)

   *VARIABLE(...)

     |  VARIABLE-NAME = <structured-name 1..20>
     |
     | ,SCOPE = *VISIBLE / *TASK
```

Operands

VARIABLE-NAME = list-poss (2000): <structured-name 1..20>(...)
Name of the constant variable to be declared. The variable must be a simple variable; it must not be a complex variable or a variable element.

VALUE = <text 0..1800 with-low *expr*>
Assigns a variable a constant value; the value must be compatible with the data type of the variable and may be specified as an expression.

TYPE =
Assigns the data type to the variable.

TYPE = *ANY
The data type STRING, INTEGER or BOOLEAN may be assigned to the variable. The data type cannot be changed once a constant variable has been declared.

TYPE = *STRING
Assigns the data type STRING to the variable.
Value range: any character string.

TYPE = *INTEGER
Assigns the data type INTEGER to the variable.
Value range: integer between -2^{31} and $2^{31}-1$.

TYPE = *BOOLEAN
Assigns the data type BOOLEAN to the variable.
Value range: TRUE, FALSE, YES, NO, ON, OFF.

SCOPE =
Defines the variable scope.

SCOPE = *CURRENT(...)
The variable is a procedure-local variable.
In call procedures, this corresponds to the entry PROCEDURE.
In include procedures, CURRENT means that the variable is declared in the current include procedure. It is then visible in this include procedure and in all include procedures on lower nesting levels (= scope: include).

IMPORT-ALLOWED =
Specifies whether the variable can be imported with IMPORT-VARIABLE in a called procedure.

IMPORT-ALLOWED = *NO
The variable cannot be imported with IMPORT-VARIABLE in a called procedure.

IMPORT-ALLOWED = *YES
The variable can be imported with IMPORT-VARIABLE in a called procedure.

SCOPE = *PROCEDURE
The variable is a procedure-local variable with the scope procedure.
The variable is declared in the current procedure.
In include procedures, the current procedure is always the higher-ranking call procedure from which the include procedure was called.
The variable is visible in this procedure and in all include procedures on lower nesting levels.

IMPORT-ALLOWED =
Specifies whether the variable can be imported with IMPORT-VARIABLE in a called procedure.

IMPORT-ALLOWED = *NO
The variable cannot be imported with IMPORT-VARIABLE in a called procedure.

IMPORT-ALLOWED = *YES
The variable can be imported with IMPORT-VARIABLE in a called procedure.

SCOPE = *TASK(...)
The variable is a task-global variable.
If it is declared in an include procedure, it is also visible in the higher-ranking call procedure from which the include procedure was called and in all include procedures on lower nesting levels.

STATE = *ANY
If a variable with the specified name already exists in the task, this variable is used; otherwise a new variable is declared.

STATE = *NEW
The task should not contain any variables with the specified name.

STATE = *OLD
The task must contain a variable with the specified name. The current variable declaration must then match the declaration of the existing variable.

CONTAINER = *STD
The variables cannot be assigned variable containers. The value of the variable is stored in class 5 memory.

CONTAINER = <composed-name 1..64>
Links the currently declared variable with the variable container specified here.
This variable container must already be open. “STD” must not be specified here, because “STD” is not interpreted as a permanently existing variable container.

CONTAINER = *VARIABLE(...)
Links the currently declared variable to another variable already defined in this procedure via a link mechanism. This variable is then known as the variable container.
Structure elements cannot be specified as variable containers.

VARIABLE-NAME = <structured-name 1..20>
Name of a variable already defined in the procedure. The variable attributes used as variable containers and the currently declared variable must be compatible with each other. This variable must also be declared with a constant value (the same constant value as is used in the VARIABLE-NAME operand) and with a constant data type.

SCOPE =
Scope of the container variable.

SCOPE = *VISIBLE
The variable is visible.

SCOPE = *TASK
Task variable.

Command return codes

(SC2)	SC1	Maincode	Meaning/Guaranteed messages
	0	CMD0001	No error
1	0	CMD0001	Warning; element already declared
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	CMD0216	Do not have required privilege
	64	SDP0091	Semantic error Guaranteed messages: SDP1018, SDP1030
	130	SDP0099	No further address space available

Example

```
/DECLARE-CONSTANT KBYTE(TYPE=*INTEGER,VALUE=1024)
/DECLARE-CONSTANT MBYTE(TYPE=*INTEGER,VALUE=KBYTE*KBYTE)
/DECLARE-CONSTANT PAMPAGE(TYPE=*INTEGER,VALUE=2*KBYTE)

/DECLARE-VARIABLE FILE(TYPE=*STRUCTURE)
/DECLARE-VARIABLE FILES(TYPE=*STRUCTURE),MULTIPLE-ELEMENTS=*LIST

/EXECUTE-CMD (SHOW-FILE-ATTRIBUTES *ALL),STRUCTURE-OUTPUT=FILES,-
/            TEXT-OUTPUT=*NO
/
/FOR FILE=*LIST(FILES)
/   IF (FILE.F-SIZE * PAMPAGE >= 5 * MBYTE)
/       WRITE-TEXT 'VERY HUGE FILE &(FILE.SHORT-F-NAME)'
/   ELSE-IF (FILE.F-SIZE * PAMPAGE >= 100 * KBYTE)
/       WRITE-TEXT 'HUGE FILE &(FILE.SHORT-F-NAME)'
/   END-IF
/END-FOR
```

This procedure declares three variables: ’KBYTE’, ’MBYTE’ and ’PAMPAGE’. They are declared with a constant value to ensure that they are correct throughout the entire procedure. Their values cannot be changed.

These variables are needed to test the size of the files of the current user ID.

DECLARE-ELEMENT
Declare structure element

Domain: **PROCEDURE**

Command description

Structure elements can be simple or complex variables (arrays, structures, lists). The following names are therefore used in the operand description below: “simple variable” (if the structure element is a simple variable), “complex variable” (if the structure element itself is a complex variable) and “variable” (if the statement applies to both simple and complex variables).

Variable attributes which cannot be defined in the DECLARE-ELEMENT command are taken from the superordinate structure (e.g. the SCOPE attribute of BEGIN-STRUCTURE or DECLARE-VARIABLE).

If the structure element is a complex variable, its elements must be initialized individually. Complex variables cannot be initialized in their entirety. This command can also be used to declare elements of dynamic structures.

Format

```
DECLARE-ELEMENT

NAME = list-poss(2000): <composed-name 1..255>(...)
   <composed-name 1..255>(...)
     | INITIAL-VALUE = *NONE / <text 0..1800 with-low expr>
     | ,TYPE = ANY / STRING / INTEGER / BOOLEAN / STRUCTURE(...)
     |    *STRUCTURE(...)
     |       | DEFINITION = *DYNAMIC / *BY-SYSCMD / <structured-name 1..20>
,MULTIPLE-ELEMENTS = NO / ARRAY(...) / LIST(...)
   ARRAY(...)
     | LOWER-BOUND = 0 / *NONE / <integer -2147483648..2147483647>
     | ,UPPER-BOUND = *NONE / <integer -2147483648..2147483647>
   LIST(...)
     | LIMIT = *NONE / <integer 1..2147483647>
```

Operands

NAME = list-poss (2000): <composed-name 1..255>(...)
Declares the variable name.

INITIAL-VALUE = *NONE
The variable is not initialized and is not assigned an initial value.
This means that the contents of an already initialized variable remain unchanged; a new variable does not contain a value, which means that a read access will produce an error.

INITIAL-VALUE = <text 0..1800 with-low *expr*>
Assigns an initial value to a new simple variable. The value must match the data type of the variable.
The operand can also be specified as an expression.
The entry is ignored for existing simple variables.
Complex variables cannot be initialized in their entirety.
Elements of structure layouts cannot be initialized.

TYPE =
Determines the data type of the variable.

TYPE = *ANY
Any values for the data types STRING, INTEGER and BOOLEAN can be assigned to the variable later on.

TYPE = *STRING
Assigns the data type STRING to the variable.
Value range: any character string.
Length: 0 to 4096 bytes (exception: if the variable is linked to a job variable, it must not be more than 256 bytes long.)

TYPE = *INTEGER
Assigns the data type INTEGER to the variable.
Value range: Integer between -2^{31} and $+2^{31}-1$

TYPE = *BOOLEAN
Assigns the data type BOOLEAN to the variable.
Value range: TRUE, FALSE, ON, OFF, YES, NO

TYPE = *STRUCTURE(...)
Stipulates that the structure element is a complex variable having the type “structure”.

DEFINITION = *DYNAMIC
Dynamically extendable structure.

DEFINITION = *BY-SYSCMD
Static structure whose elements are subsequently declared by commands in the SYSCMD stream.

DEFINITION = <structured-name 1..20>
Name of the structure layout through which the static structure is defined.

MULTIPLE-ELEMENTS = *NO
Determines that the structure elements is not an array or a list.

MULTIPLE-ELEMENTS = *ARRAY(...)
Declares an array, i.e. the structure element is declared as a complex variable having the type “array”.
An array cannot be initialized in its entirety.

LOWER-BOUND = 0 / <integer -2147483648..2147483647>
Lower limit for the array index.

LOWER-BOUND = *NONE
A lower limit is not defined for the array index.

UPPER-BOUND = *NONE
An upper limit is not defined for the array index.

UPPER-BOUND = <integer -2147483648..2147483647>
Upper limit for the array index.

MULTIPLE-ELEMENTS = *LIST(...)
Declares a list, i.e. the structure element is declared as a complex variable having the type “list”.

LIMIT = *NONE
The number of list elements is unlimited.

LIMIT = <integer 1..2147483647>
Defines the maximum number of list elements.

Command return codes

(SC2)	SC1	Maincode	Meaning/Guaranteed messages
	0	CMD0001	No error
1	0	CMD0001	Warning; element already declared
2	0	CMD0001	Warning; INITIAL-VALUE operand was ignored
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	CMD0216	Do not have required privilege
	64	SDP0091	Semantic error Guaranteed messages: SDP1018
	130	SDP0099	No further address space available

Example 1

```
/BEGIN-STRUCTURE NAME = BANK-CONNECT(SCOPE = *TASK)
/DECLARE-ELEMENT BANK-CODE(TYPE = *INTEGER)
/DECLARE-ELEMENT ACCT(TYPE = *INTEGER),-
/MULTIPLE-ELEMENTS = *ARRAY(LOWER-BOUND = 1, UPPER-BOUND =3)
/END-STRUCTURE
/DECLARE-VARIABLE PERSON(TYPE = *STRUCTURE(DEFINITION = *BY-SYSCMD))
/BEGIN-STRUCTURE

/DECLARE-ELEMENT SURNAME(TYPE = *STRING)
/DECLARE-ELEMENT FORENAME(TYPE = *STRING)
/DECLARE-ELEMENT BANK-CONNECT(TYPE = *STRUCTURE-
/(DEFINITION = BANK-CONNECT))
/END-STRUCTURE
```

Generates the procedure-local structure PERSON:

```
PERSON.SURNAME
PERSON.FORENAME
PERSON.BANK-CONNECT.BANK-CODE
PERSON.BANK-CONNECT.ACCT
```

The variables PERSON.SURNAME and PERSON.FORENAME were declared with TYPE = *STRING and can therefore be assigned only strings.
The variable PERSON.BANK-CONNECT.BANK-CODE was declared with TYPE = *INTEGER and can therefore be assigned only integers.
PERSON.BANK-CONNECT.ACCT is an array containing three elements which can be assigned only INTEGER values. The elements in this array are not yet generated in the variable declaration, but only after the first assignment is made.

For example, the following independent assignments can be made:

```
PERSON.BANK-CONNECT.BANK-CODE = 70010080
PERSON.BANK-CONNECT.ACCT#1 = 6001023
```

Example 2

```
/DECLARE-VARIABLE VARIABLE-NAME = TREE(TYPE = *STRUCTURE(*BY-SYSCMD)),-
/MULTIPLE-ELEMENTS = *ARRAY(LOWER-BOUND = 1,UPPER-BOUND = 10),-
/SCOPE = *TASK
/BEGIN-STRUCTURE
/DECLARE-ELEMENT AST,MULTIPLE-ELEMENTS = *ARRAY
/END-STRUCTURE
```

The following assignments can now be made:

```
/TREE#1.BRCH#1 =...;  /TREE#1.BRCH#2 =...
/TREE#2.BRCH#1 =...;  /TREE#2.BRCH#2 =...
........
/TREE#10.BRCH#1 =...;  /TREE#10.BRCH#2 =...
```

Example 3

```
/BEGIN-STRUCTURE NAME = HOUSING-UNIT
/   DECLARE-ELEMENT NMBR-ROOMS
/   DECLARE-ELEMENT ROOM-SIZE, MULTIPLE-ELEM = *ARRAY
/   DECLARE-ELEMENT TENANT-NAME
/END-STRUCTURE
/DECLARE-VARIABLE BLDG(TYPE = *STRUCTURE(*BY-SYS)),-
/       MULTIPLE-ELEMENT = *ARRAY
/BEGIN-STRUCTURE
/DECLARE-ELEMENT APT(TYPE = *STRUCTURE(DEF = HOUSING-UNIT)),-
/       MULTIPLE-ELEMENT = *ARRAY
/DECLARE-ELEMENT OWNER(TYPE = *STRUCTURE(DEF = HOUSING-UNIT)),-
/       MULTIPLE-ELEMENT = *ARRAY
/DECLARE-ELEMENT ADDRESS(TYPE = *STRUCTURE(DEF = HOUSING-UNIT)),-
/       MULTIPLE-ELEMENT = *ARRAY
/END-STRUCTURE
```

In assignments, the variables are addressed as follows:

```
BLDG#1.APT#1.NMBR-ROOMS =
BLDG#1.APT#2.NMBR-ROOMS =
BLDG#1.APT#3.NMBR-ROOMS =
...
BLDG#5.APT#8.NMBR-ROOMS =
...
```

Example 4

The declaration of a structure layout begins with the BEGIN-STRUCTURE command and ends with the END-STRUCTURE command.

```
/BEGIN-STRUCTURE NAME = AA
/DECLARE-ELEMENT Z
/END-STRUCTURE
/BEGIN-STRUCTURE NAME = BB
/DECLARE-ELEMENT X
/DECLARE-ELEMENT Y
/END-STRUCTURE
/DECLARE-VARIABLE V
/DECLARE-VARIABLE W
/...
/IF (V = W)
/DECLARE-VARIABLE A(TYPE = *STRUCTURE(DEF = AA))
/ELSE
/DECLARE-VARIABLE B(TYPE = *STRUCTURE(DEF = BB))
/END-IF
```

If V = W applies, structure A is declared (consisting of variables A.Z); otherwise, structure B is declared (consisting of variables B.X, B.Y). Control flow commands and procedure calls containing INCLUDE-PROCEDURE can be placed between BEGIN-STRUCTURE and END-STRUCTURE. However, elements of the calling procedure cannot be declared in the called include procedure. The elements of a structure must be declared in the same procedure as their starting BEGIN-STRUCTURE and ending END-STRUCTURE.
A reference in the include procedure to a structure which has been incompletely declared in this manner will produce an error.

Example 5

```
/DECLARE-VARIABLE DYN-STRUC(TYPE=*STRUCTURE(DEFINITION=*DYNAMIC))
/DECLARE-ELEMENT DYN-STRUC.SUB.NUMBER(TYPE=*INTEGER)
/DECLARE-ELEMENT DYN-STRUC.LIST,MULTIPLE-ELEMENTS=*LIST
/DYN-STRUC.SUB.STRING='DYNAMICALLY CREATED ELEMENT WITH DATA TYPE *ANY'
/DYN-STRUC.SUB.NUMBER=1234
/DYN-STRUC.LIST#1=1
/DYN-STRUC.LIST#2=2
```

Notes

– When elements are being declared for a structure layout, the operand specification NAME=list-poss(2000): <structured-name 1..20>(...) defines only the name of the element, it does not determine the name of the layout.
– When elements are being declared for a variable which has been defined as a dynamic structure, the operand specification NAME=list-poss(2000): <structured-name 1..255>(...) defines the complete name of the element, including the name of the variable as a whole.

DECLARE-PARAMETER
Declare procedure parameters

Domain: **PROCEDURE**

Command description

The procedure parameters needing an actual value during the procedure run are declared with the DECLARE-PARAMETER command. Furthermore, the way the parameter values are passed to the procedure is defined (initial value, prompting, ...). Procedure parameters may only be declared in the procedure header.

Procedure parameters are variables local to the procedure in SDF-P: When defined in the procedure header, they implicitly have SCOPE = *CURRENT.

The names of the procedure parameters are also keywords for the procedure parameters in the PROCEDURE-PARAMETERS operand of the CALL-PROCEDURE, ENTER-PROCEDURE and INCLUDE-PROCEDURE commands.

Format

```
DECLARE-PARAMETER

 NAME = list-poss(2000): <structured-name 1..20>(...)

   <structured-name 1..20>(...)
     |  INITIAL-VALUE = *NONE / *PROMPT(...) / <text 0..1800 with-low expr>
     |     *PROMPT(...)
     |        |  PROMPT-STRING = *STD / <text 0..1800 with-low string-expr>
     |        | ,DEFAULT-VALUE = *NONE / <text 0..1800 with-low expr>
     |        | ,SECRET-INPUT  = *NO / *YES
     | ,TYPE = *ANY / *STRING / *INTEGER / *BOOLEAN
     | ,TRANSFER-TYPE = *BY-VALUE / *BY-REFERENCE
```

Operands

NAME = list-poss (2000): <structured-name 1..20>(...)
Determines the name of the procedure parameter.

INITIAL-VALUE =
Determines the initial value of the procedure parameter.

INITIAL-VALUE = *NONE
No initial value is declared; the procedure parameter is not initialized. A value must be assigned to the procedure parameter when the procedure is called (see section "Passing procedure parameters" on page 106).

INITIAL-VALUE = *PROMPT(...)
If the procedure parameter does not yet contain a value the first time a read access is carried out, the value is requested in dialog. If this happens, the value is always converted into uppercase letters. If the value is enclosed between apostrophes, these are removed. If a dialog request is not possible, error message SDP0219 is output.

PROMPT-STRING =
Defines a string to be output as the prompt string (to prompt for input). The text specified in `DEFAULT-VALUE = ...` is added to the prompt string. The prompt always ends with a colon. The following then appears as the prompt:
<prompt-string>␣(DEFAULT = <default-value>)␣:

PROMPT-STRING = *STD
The parameter name (variable name) specified in `NAME=...` is output by default.

PROMPT-STRING = <text 0..1800 with-low *string-expr*>
Defines the string to be output as the prompt string.

DEFAULT-VALUE =
Defines an initial value in case nothing was input in the dialog (i.e. only [DUE]) or when the procedure runs in the background. The value is output as part of the prompt (for informational purposes).

DEFAULT-VALUE = *NONE
No (default) string is defined.

DEFAULT-VALUE = <text 0..1800 with-low *expr*>
Expression to be used as the default for the initial value. The expression specified must match the type of the parameter.

SECRET-INPUT = *NO / *YES
You can define if the input is to be entered secretly (i.e. will not be displayed) in the dialog. The input is also not logged in this case.

INITIAL-VALUE = <text 0..1800 with-low *expr*>
Determines an initial value. The specified expression must match the data type of the procedure parameter. This initial value applies unless some other value is passed when the procedure is called.

TYPE =
Determines the data type of the procedure parameter.

TYPE = <u>*ANY</u>
Stipulates that the procedure parameter can be assigned any STRING, INTEGER or BOOLEAN value.

TYPE = *STRING
Assigns the data type STRING to the procedure parameter.
Value range: any character string.

TYPE = *INTEGER
Assigns the data type INTEGER to the procedure parameter.
Value range: Integer between -2^{31} and $-2^{31}-1$

TYPE = *BOOLEAN
Assigns the data type BOOLEAN to the procedure parameter.
Value range: TRUE, FALSE, YES, NO, ON, OFF

TRANSFER-TYPE =
Declares whether the entered character string is to be interpreted as a value or a variable name.

TRANSFER-TYPE = <u>*BY-VALUE</u>
The specified character string is a value.
A procedure-local variable which accepts this value is declared. Nothing is returned to the calling procedure. The procedure parameter is used only as an input parameter. This is the same as the transfer mechanism in non-S procedures. The value of the entered argument overwrites the initial value in the DECLARE-PARAMETER command. If a value is not transferred to the procedure parameter, the initial value in the DECLARE-PARAMETER command applies. If INITIAL-VALUE = *NONE is defined, a value must be entered for this procedure parameter.

The entered string must be convertible to the type of the formal procedure parameter. A formal procedure parameter with TYPE = *ANY is always assigned the current type STRING.

Since procedure parameters are variables for the purposes of SDF-P, their values can be changed during procedure execution.

TRANSFER-TYPE = *BY-REFERENCE
The specified character string is the name of a variable containing the value of the procedure parameter. Each access to the procedure parameter in the called procedure is an access to the same variable in the calling procedure.

Command return codes

If DECLARE-PARAMETER is used in the procedure head of an S procedure, it is completely evaluated during procedure analysis. Any error is thus an error during procedure preparation; the procedure has not been executed when the error occurs. (For further details, see the command return codes for CALL-PROCEDURE or INCLUDE-PROCEDURE.) The following return codes can thus appear only if DECLARE-PARAMETER is used in another (i.e. wrong) context.

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	130	SDP0099	No further address space available

Example 1: Demonstration of prompting

In the procedure PROC.PROMPT, a procedure parameter is declared such that the system prompts the user for a value when the procedure is called. In this example, the parameter is the German name of a color, which is then translated into English.

```
/SET-PROCEDURE-OPTIONS
/DECLARE-PARAMETER NAME(INITIAL-VALUE=*PROMPT-
/    (PROMPT-STRING='ENTER THE NAME OF THE COLOR TO BE TRANSLATED',-
/    DEFAULT-VALUE='ROT')
/COLOR=TRANSLATE(STRING=NAME-,
/,WHEN1='ROT',    THEN1='RED'-,
/,WHEN2='GRUEN',  THEN2='GREEN'-,
/,WHEN3='BLAU',   THEN3='BLUE'-,
/,WHEN4='GELB',   THEN4='YELLOW'-,
/,WHEN5='SCHWARZ',THEN5='BLACK'-,
/,WHEN6='WEISS',  THEN6='WHITE'-,
/,ELSE='UNKNOWN')
/SHOW-VAR NAME
/SHOW-VAR COLOR
```

When the procedure is called, the first occurrence of the procedure parameter NAME, for which the system is to prompt the user for a value, causes the prompt message to be displayed. When the user enters a color (in German), this color is then translated into English.

```
(IN) /CALL-PROC PROC.PROMPT
(OUT) %PLEASE ENTER THE NAME OF THE COLOR TO BE TRANSLATED (DEFAULT = ROT):
(IN) ROT
(OUT) NAME = ROT
(OUT) COLOR = RED
```

Example 2: Effect of TRANSFER-TYPE

In Procedure P, procedure parameter PAR1 is declared with TRANSFER-TYPE = *BY-VALUE.

```
/DECLARE-PARAMETER PAR1(TYPE = *STRING,TRANSFER-TYPE = *BY-VALUE)
```

The type of parameter passing to an S procedure during the procedure call is the same as the type of the procedure transfer for non-S procedures.

The procedure could be called as follows:

```
/CALL-PROCEDURE P,PROCEDURE-PARAMETER=(PAR1 = ABC)
/CALL-PROCEDURE P,PROCEDURE-PARAMETER=(PAR1 = 'ABC')
```

In both instances, the string ’ABC’ is entered. Note that a write access to PAR1 can also be carried out within P; however, this has no effect on the environment of the caller. If the value of variable X is transferred to PAR1, an expression replacement operation must be used (with the valid escape character):

```
/CALL-PROCEDURE P,PROCEDURE-PARAMETER=(PAR1 = &X)
```

&X means that the value of variable X is entered. Changes to PAR1 have no effect for the calling procedure.

Procedure parameter PAR2 is declared in procedure P with `TRANSFER-TYPE = *BY-REFERENCE`

```
/DECLARE-PARAMETER PAR2(TYPE = *STRING, TRANSFER-TYPE = *BY-REFERENCE)
```

Procedure P could be called as follows:

```
/CALL-PROCEDURE P,PROCEDURE-PARAMETER=(PAR2 = ABC)
/CALL-PROCEDURE P,PROCEDURE-PARAMETER=(PAR2 = 'ABC')
```

In both instances, each access to variable PAR2 is actually an access to variable ABC in the calling procedure.

Example 3

In dialog, variable ABC is declared and assigned the value LEVEL0.
Procedure P is subsequently called.

```
/ABC = 'LEVEL0'
/CALL-PROCEDURE P,(ABC)
```

Procedure parameter PAR3 is declared in procedure P:

```
/DECLARE-PARAMETER PAR3(TYPE = *STRING, TRANSFER-TYPE = *BY-REFERENCE)
```

Procedure P contains the following command sequence:

```
/ABC = PAR3
/PAR3 = 'LEVEL1'
/SHOW-VARIABLE ABC
/EXIT-PROCEDURE
```

The assignment ABC = PAR3 means that variable ABC is implicitly declared in procedure P and is assigned the contents of procedure parameter PAR3, i.e. the contents of the variable ABC initialized in dialog (’LEVEL0’).
Procedure parameter PAR3 is then assigned the value ’LEVEL1’ and thus simultaneously assigned the variable ABC on the interactive level.
The SHOW-VARIABLE command is used to output the contents of variable ABC, which is visible in procedure P, i.e. the variable implicitly declared in procedure P during the first assignment (contents: LEVEL0).
Procedure P is terminated with EXIT-PROCEDURE.
The following command is now called in dialog:

```
/SHOW-VARIABLE ABC
```

This command then accesses the variable ABC declared in dialog. Since the contents of this variable are influenced by procedure parameter PAR3 in procedure P, SHOW-VARIABLE shows LEVEL1 as the variable contents (in procedure P, procedure parameter PAR3 was assigned this value).

Example 4

```
PROC.1

/SET-PROCEDURE-OPTIONS
/DECLARE-VARIABLE GARDEN(TYPE = *STRUCTURE(DEFINITION=*DYNAMIC))
/   GARDEN.CHAIR = 4
/   GARDEN.TABLE = 1
/   GARDEN.FURNIT = 0
/CALL-PROCEDURE PROC.2,(,&(GARDEN.TABLE),-
/       &(GARDEN.CHAIR),GARDEN.FURNIT)
/SHOW-VARIABLE GARDEN.FURNIT
GARDEN.FURNIT = 5

PROC.2

/SET-PROCEDURE-OPTIONS
/BEGIN-PARAMETER-DECLARATION
/DECLARE-PARAMETER ART('WINTERGARDEN',TRANSFER-TYPE=*BY-VALUE)
/DECLARE-PARAMETER TABLE(*NONE,TYPE=*INTEGER,TRANSFER-TYPE=*BY-VALUE)
/DECLARE-PARAMETER CHAIRS(0,TYPE=*INTEGER,TRANSFER-TYPE=*BY-VALUE)
/DECLARE-PARAMETER TOTAL(0,TRANSFER-TYPE=*BY-REFERENCE)
/END-PARAMETER-DECLARATION
/TOTAL=(TABLE)+(CHAIRS)
/SHOW-VARIABLE ART
/SHOW-VARIABLE TABLE
/SHOW-VARIABLE CHAIRS
/SHOW-VARIABLE TOTAL
ART = WINTERGARDEN
TABLE = 1
CHAIRS = 4
TOTAL = 5
```

The formal procedure parameter ART is assigned the current procedure parameter, which is an empty procedure parameter. ART is initialized with ’WINTERGARDEN’.
The second current procedure parameter cannot be a blank procedure parameter. This is made mandatory by INIT-VALUE = *NONE. The value of GARDEN.TABLE is transferred to the formal procedure parameter TABLE.
CHAIRS is assigned the value of GARDEN.CHAIR. The initial value 0 is overwritten by 4.

DECLARE-VARIABLE
Declare variable

Domain: **PROCEDURE**

Command description

DECLARE-VARIABLE is used to define the attributes of this variable and possibly an initial value as well.

Job variables can be integrated in SDF-P via the CONTAINER operand.

It is possible to make what would normally be a procedure-local S variable in a local procedure accessible for a procedure called with CALL-PROCEDURE. To do this, DECLARE-VARIABLE VARIABLE-NAME=..., SCOPE = CURRENT or SCOPE = PROCEDURE (IMPORT-ALLOWED=*YES) must be specified in the local procedure and IMPORT-VARIABLE VARIABLE-NAME= ..., FROM=*SCOPE(SCOPE=*CALLING-PROCEDURES) in the called procedure.

Format

(part 1 of 2)

```
DECLARE-VARIABLE                                                          Alias: DCV

VARIABLE-NAME = list-poss(2000): <structured-name 1..20>(...)

   <structured-name 1..20>(...)

      |  INITIAL-VALUE = *NONE / <text 0..1800 with-low expr>
      |
      | ,TYPE = *ANY / *STRING / *INTEGER / *BOOLEAN / *STRUCTURE(...)
      |
      |     *STRUCTURE(...)
      |
      |        |  DEFINITION = *DYNAMIC / *BY-SYSCMD / <structured-name 1..20>

,MULTIPLE-ELEMENTS = *NO / *ARRAY(...) / *LIST(...)

   *ARRAY(...)

      |  LOWER-BOUND = 0 / *NONE / <integer -2147483648..2147483647>
      |
      | ,UPPER-BOUND = *NONE / <integer -2147483648..2147483647>

   *LIST(...)

      |  LIMIT = *NONE / <integer 1..2147483647>
```

Continued ➠

(part 2 of 2)

```
,SCOPE = *CURRENT(...) / *PROCEDURE(...) / *TASK(...)

   *CURRENT(...)

     | IMPORT-ALLOWED = *NO / *YES

   *PROCEDURE(...)

     | IMPORT-ALLOWED = *NO / *YES

   *TASK(...)

     | STATE = *ANY / *NEW / *OLD

,CONTAINER = *STD / <composed-name 1..64> / *VARIABLE(...) / *JV(...)

   *VARIABLE(...)

     | VARIABLE-NAME = <structured-name 1..20>
     |
     | ,SCOPE = *VISIBLE / *TASK

   *JV(...)

     | JV-NAME = <filename 1..54>
     |
     | ,STATE = *ANY / *NEW / *OLD
```

Operands

VARIABLE-NAME = list-poss (2000): <structured-name 1..20>(...)
Declares the variable name, i.e. the name of a simple variable, which is not an element in a complex variable, or the name of a complex variable.

INITIAL-VALUE = *NONE
The variable is not initialized.
For a new variable, this means that the variable does not contain an initial value. A read access would produce an error.
If the variable is already present, its contents remain unchanged; it is not assigned a new initial value.

INITIAL-VALUE = <text 0..1800 with-low *expr*>
Assigns an initial value to a new variable; the value must match the data type of the variable and can also be specified as an expression.
The entry is ignored for existing variables; they are not assigned a new initial value.
Complex variables cannot be initialized in their entirety, i.e. INITIAL-VALUE cannot be used to assign a new initial value to these variables.

TYPE =
Assigns the data type to the variable.

TYPE = <u>*ANY</u>
The variable can be assigned any value of data types STRING, INTEGER and BOOLEAN.

TYPE = *STRING
Assigns the data type STRING to the variable.
Value range: any character string.

TYPE = *INTEGER
Assigns the data type INTEGER to the variable.
Value range: integer between -2^{31} and $+2^{31}-1$.

TYPE = *BOOLEAN
Assigns the data type BOOLEAN to the variable.
Value range: TRUE, FALSE, YES, NO, ON, OFF.

TYPE = *STRUCTURE(...)
Declares a complex variable of type “structure”.

DEFINITION = <u>*DYNAMIC</u>
Dynamically extendable structure.

DEFINITION = *BY-SYSCMD
Static structure whose elements are declared exclusively by commands in the SYSCMD stream.

DEFINITION = <structured-name 1..20>
Name of the structure layout.

MULTIPLE-ELEMENTS = <u>*NO</u>
Determines that the variable is not an array or a list.

MULTIPLE-ELEMENTS = *ARRAY(...)
Declares a complex variable of type “array”.

LOWER-BOUND = <u>0</u> / <integer - 2147483648..2147483647>
Lower limit for the array index.

LOWER-BOUND = *NONE
No lower limit is defined for the array index.

UPPER-BOUND = <u>*NONE</u>
No upper limit is defined for the array index.

UPPER-BOUND = <integer -2147483648..2147483647>
Upper limit for the array index. The specified value must be greater than or equal to the value for LOWER-BOUND.

MULTIPLE-ELEMENTS = *LIST(...)
Declares a complex variable of type “list”.

LIMIT = <u>*NONE</u>
The number of list elements is unlimited.

LIMIT = <integer 1..2147483647>
Maximum number of list elements.

SCOPE =
Defines the variable scope.

SCOPE = <u>*CURRENT</u>
The variable is a procedure-local variable.
In call procedures, this corresponds to the entry PROCEDURE.
In include procedures, CURRENT means that the variable is declared in the current include procedure. It is then visible in this include procedure and in all include procedures on lower nesting levels (= scope: include).

IMPORT-ALLOWED =
Specifies whether the variable can be imported with IMPORT-VARIABLE in a called procedure.

IMPORT-ALLOWED = <u>*NO</u>
The variable cannot be imported with IMPORT-VARIABLE in a called procedure.

IMPORT-ALLOWED = *YES
The variable can be imported with IMPORT-VARIABLE in a called procedure.

SCOPE = *PROCEDURE
The variable is a procedure-local variable with the scope procedure.
The variable is declared in the current procedure.
In include procedures, the current procedure is always the superordinate call procedure from which the include procedure was called.
The variable is visible in this procedure and in all include procedures on lower nesting levels.

IMPORT-ALLOWED =
Specifies whether the variable can be imported with IMPORT-VARIABLE in a called procedure.

IMPORT-ALLOWED = <u>*NO</u>
The variable cannot be imported with IMPORT-VARIABLE in a called procedure.

IMPORT-ALLOWED = *YES
The variable can be imported with IMPORT-VARIABLE in a called procedure.

SCOPE = *TASK(...)
The variable is a task-global variable.
If it is declared in an include procedure, it is also visible in the superordinate call procedure from which the include procedure was called and in all include procedures on lower nesting levels.

STATE = <u>*ANY</u>
If a variable with the specified name already exists in the task, this variable is used; otherwise a new variable is declared.

STATE = *NEW
The task should not contain any variables with the specified name.

STATE = *OLD
The task must contain a variable with the specified name. The current variable declaration must then match the declaration of the existing variable.

CONTAINER = <u>*STD</u>
The variable is assigned no variable container. The value of the variable is stored in a class 5 memory.

CONTAINER = <composed-name 1..64>
Links the currently declared variable with the variable container specified here.
This variable container must already be open. “*STD” must not be specified here, because it is not interpreted as a variable container with a permanent existence.

CONTAINER = *VARIABLE(...)
Links the currently declared variable to another variable already defined in this procedure via a link mechanism. This variable is then known as the variable container.
Structure elements cannot be specified as variable containers.

VARIABLE-NAME = <structured-name 1..20>
Name of a variable already defined in the procedure. The variable attributes used as variable containers and the currently declared variable must be compatible with each other.

SCOPE =
Scope of the container variable.

SCOPE = <u>*VISIBLE</u>
The variable is visible.

SCOPE = *TASK
Task variable.

CONTAINER = *JV(...)
Defines a job variable as a variable container: the currently declared variable is linked to a job variable, i.e. the value of the variable is stored in the job variable.
Variable containers having the type JV can be linked only to simple variables declared with TYPE = *STRING, and the string can be no more than 256 bytes long.
Complex variables cannot be linked to variable containers having the type JV.
If a task-global variable (SCOPE = *TASK) is linked to a job variable, the STATE entries in TASK and JV operands must be logically compatible. It must be ensured that the redeclaration of an existing task-global variable does not generate a new job variable.

Job variables are part of the chargeable software product “Job Variables”. They are available only if the JV subsystem is loaded. See the “Job Variables” manual [5] for more information on job variables.

JV-NAME = <filename 1..54>
Name of the job variable.

STATE = *ANY
If a job variable with this name already exists, this job variable is used; otherwise a new job variable is declared.

STATE = *OLD
The job variable must already exist.

STATE = *NEW
A new job variable is declared; the job variable should not already exist.

Command return code

(SC2)	SC1	Maincode	Meaning/Guaranteed messages
	0	CMD0001	No error
1	0	CMD0001	Nothing executed; element already declared
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	CMD0216	Do not have required privilege
	64	SDP0091	Semantic error Guaranteed messages: SDP1018, SDP1030
	130	SDP0099	No further address space available

Example 1

```
/DECLARE-VARIABLE A, SCOPE = *TASK
```

Variable A is declared as a task-global variable with TYPE = *ANY.

Example 2

```
/DECLARE-VARIABLE DATA(C'ANTON',*ANY)
```

The procedure-local variable DATA having the type *ANY is initialized with the string ’ANTON’.

Example 3

```
/DECLARE-VARIABLE LOGO(TRUE, *BOOLEAN)
```

The local Boolean variable LOGO is assigned the Boolean value TRUE.

Example 4

```
/DECLARE-VARIABLE LEN, SCOPE = *TASK
```

The variable LEN is declared.

Example 5

```
/DECLARE-VARIABLE BEG, TYPE = *STRING, SCOPE = *TASK, CONTAINER =
*JV(BEGIN.DAT)
```

The system searches for the job variable BEGIN.DAT. If it is not located, a job variable with this name is cataloged. The value of BEG is always stored in the job variable BEGIN.DAT.

Example 6

A task-global variable with the name A is declared for the ID:

```
/DECLARE-VARIABLE A, SCOPE = *TASK
```

A procedure is created which declares a variable having the name A and SCOPE = *PROCEDURE:

```
/SET-PROCEDURE-OPTIONS
/DECLARE-VARIABLE A, SCOPE = *PROCEDURE
 ...
/A = FILE1
/DELETE-FILE &A
...
....
/CREATE-FILE &A, ...
/...
```

Each time &A is used within the procedure, the contents of the local variable are accessed.

Example 7

A task-global variable VAR-A exists for the ID:

```
/DECLARE-VARIABLE VAR-A, SCOPE = *TASK
/VAR-A = 'TASK VARIABLE OF USER ID'
```

The procedure PROCEDUR.1 is called:

```
PROCEDUR.1

/SET-PROCEDURE-OPTIONS
/DECLARE-VARIABLE VAR-A, SCOPE = *PROCEDURE
/VAR-A = 'LOCAL VARIABLE FROM PROCEDURE 1'
/CALL-PROCEDURE PROCEDUR.2
/SHOW-VARIABLE VAR-A
```

'LOCAL VARIABLE FROM PROCEDURE 1' is output; in this procedure, VAR-A is declared as a local variable.

```
PROCEDUR.2

/SET-PROCEDURE-OPTIONS
/DECLARE-VARIABLE VAR-A, SCOPE = *TASK
/SHOW-VARIABLE VAR-A
```

'TASK VARIABLE OF USER ID' is output, since the global variable is accessed.

Example 8

Example of an extendable structure, even if implicit declarations are forbidden:

```
/DECLARE-VARIABLE A(TYPE = *STRUCTURE(*DYNAMIC))
/SET-VARIABLE  A.B = 7
/SET-VARIABLE  A.X = TRUE
/SET-VARIABLE  A#1 = 0        "Error: array element name"
```

The structure now contains the elements A.B and A.X.

Example 9

Elements of dynamic structures can also be declared explicitly:

```
/DECLARE-VARIABLE A(TYPE = *STRUCT(*DYNAMIC))
/DECLARE-ELEMENT A.B(7, *INTEGER)
/DECLARE-ELEMENT A.X(TRUE, *BOOLEAN)
/SET-VARIABLE  A#1 = 0        "Error: array element name"
```

The structure now contains the elements A.B and A.X.

Example 10

When using variable containers, the structure layouts must be visible before the variable is declared. These layouts must be compatible (in relation to the SET-VARIABLE command) with the structure of the variables which are saved in the variable container.

```
/BEGIN-STRUCTURE MYSTRUCT
/  DECLARE-ELEMENT ELEM1
/  DECLARE-ELEMENT ELEM2
/END-STRUCTURE
/OPEN-VARIABLE-CONTAINER MYCONT,FROM-FILE=*LIB-ELEM(MY-LIBRARY), -
/                                         AUTOMATIC-DECLARE=*NONE
/DECLARE-VARIABLE MYVAR(TYPE=*STRUCTURE(MYSTRUCT)), CONTAINER=MYCONT -
/ “MYVAR IS CREATED IN THE VARIABLE CONTAINER”
/MYVAR.ELEM1 = 'FIRST VALUE'
/MYVAR.ELEM2 = 'SECOND VALUE'
/SAVE-VARIABLE-CONTAINER MYCONT
/
/“THE LAYOUT, MYSTRUCT, IS NOW CHANGED (ONE ELEMENT IS SUPPRESSED,”
/“A DIFFERENT ONE IS APPENDED AS A NEW ELEMENT)”
/


/BEGIN-STRUCTURE MYSTRUCT
/  DECLARE-ELEMENT ELEM2
/  DECLARE-ELEMENT NEW-ELEM
/END-STRUCTURE
/OPEN-VARIABLE-CONTAINER MYCONT,FROM-FILE=*LIB-ELEM(MY-LIBRARY), -
/                                         AUTOMATIC-DECLARE=*NONE
/DECLARE-VARIABLE MYVAR(TYPE=*STRUCTURE(MYSTRUCT)), CONTAINER=MYCONT -
/ “MYVAR IS RETRIEVED FROM THE VARIABLE CONTAINER”
/SHOW-VARIABLE MYVAR
```

```
MYVAR.ELEM2 = SECOND VALUE
```

Only MYVAR.ELEM2 is output, because MYVAR.ELEM1 has been suppressed and MYVAR.NEW-ELEM has not yet been initialized.

See IMPORT-VARIABLE (page 678) for another example.

DELETE-STREAM
Delete S variable stream

Domain: **PROCEDURE**

Command description

The DELETE-STREAM command deletes S variable streams. Their assignments will no longer be displayed by SHOW-STREAM-ASSIGNMENT.

The deletion is restricted to those streams which were created at the highest procedure level (in dialog mode). Any subsequent transmissions via the deleted stream will be rejected.

Variable streams in procedures are implicitly deleted on exiting from the procedure unless the setting in SET-PROCEDURE-OPTIONS is SYSTEM-FILE-CONTEXT=*SAME-AS-CALLER.

Variable streams with reserved names (SYSINF, SYSMSG, ...) can never be deleted.

Format

DELETE-STREAM
STREAM-NAME = <composed-name 1..20 with-wild(40)> / list-poss(100): <structured-name 1..20>

Operands

STREAM-NAME =
Name of the S variable stream to be deleted.

STREAM-NAME = <structured-name 1..20 with-wild(40)>
All S variable streams which match this search pattern are assigned to *DUMMY, and are suppressed.

STREAM-NAME = list-poss(100): <structured-name 1..20>
List of S variable stream names which are to be assigned to *DUMMY and suppressed.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
2	0	SDP0516	Specified variable stream name does not exist, or the search pattern was not found; process continues
1	0	SDP0518	No match for wildcard. process continues
2	0	SDP0535	Warning from the server during deletion of a specified S variable steam; process continues
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	CMD0216	Do not have required privilege
	64	SDP0091	Semantic error
	64	SDP0515	Specified variable stream not created at the highest level
	64	SDP0536	Server error; deletion of the specified S variable stream has been rejected
	64	SDP0537	Internal server error; deletion of the specified S variable stream has been rejected. Server link terminated following unexpected event or due to system resource shortage or error

DELETE-VARIABLE
Delete variable

Domain: **PROCEDURE**

Command description

DELETE-VARIABLE deletes the declaration of an S variable within the current scope, i.e. including the declarations of imported task variables.

The name of the S variable can no longer be used, and its value is deleted.

Either simple or complex variables can be deleted, but not individual elements of complex variables.

The following variable declarations cannot be deleted using DELETE-VARIABLE:

– procedure parameters
– elements of complex variables
– system variables (e.g. SYSWITCH)
– container JVs
– non-permanent container variables
– structure layouts

Format

DELETE-**VAR**IABLE
VARIABLE-**NAME** = <structured-name 1..20 with-wild(40)> / list-poss(2000): <structured-name 1..20>

Operands

VARIABLE-NAME =
Name of the S variable to be deleted.

VARIABLE-NAME = <structured-name 1..20 with-wild(40)>
All the S variables which match this search pattern are deleted.

VARIABLE-NAME = list-poss(2000):<structured-name 1..20>
List of S variables to be deleted.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
1	0	CMD0001	Warning; nothing executed
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	CMD0216	Do not have required privilege
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available

Note

Error SDP1098 does not appear if variable names are specified with wildcards.

Example 1

Input

```
/DECLARE-VARIABLE TRIAL
/SET-VARIABLE TRIAL=15
/SHOW-VARIABLE TRIAL
```

Output

```
TRIAL = 15
```

Input

```
/DELETE-VARIABLE TRIAL
/SHOW-VARIABLE TRIAL
```

Output

```
SDP1008 VARIABLE/LAYOUT 'TRIAL' DOES NOT EXIST
SDP0234 OPERAND 'NAME' IS INCOMPLETE
```

Example 2

Input

```
/DELETE-VARIABLE SYS*  “No error message is returned”
/DELETE-VARIABLE SYSSWITCH
```

Output

```
SDP1098 DELETE VARIABLE NOT ALLOWED FOR THE VARIABLE 'SYSSWITCH'
```

ELSE
Initiate ELSE branch in IF block

Domain: **PROCEDURE**

Command description

In the IF block, the ELSE command initiates the last branch. The commands between the ELSE command and the terminating END-IF command are executed if none of the conditions previously checked in the IF or ELSE-IF commands applies (see section “Defining conditional branches” on page 93).

In the IF-BLOCK-ERROR and IF-CMD-ERROR blocks, the ELSE branch is executed if no errors occur (see section “Error handling” on page 69).

Format

ELSE

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	1	SDP0223	Incorrect environment
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	130	SDP0099	No further address space available

Example

See the IF command, page 672.

ELSE-IF
Initiate alternative branch in IF block

Domain: **PROCEDURE**

Command description

The ELSE-IF branch is executed if the condition specified in the ELSE-IF command applies; otherwise the system searches for the next branch in the IF block or an END-IF command. The ELSE-IF branch contains all commands positioned between the current ELSE-IF command and the next ELSE-IF, ELSE or END-IF command. (Full details are contained in section “Creating the procedure body” on page 92.)

Format

ELSE-IF
CONDITION = <test 0..1800 with-low *bool-expr*>

Operands

CONDITION = <test 0..1800 with-low *bool-expr*>
Logical expression
Defines the condition which must be met in order for the commands in the current ELSE-IF branch to be executed (see chapter “Expressions” on page 249 for information on logical expressions).

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	1	SDP0223	Incorrect environment
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	130	SDP0099	No further address space available

Example

See the IF command, page 672.

END-BLOCK
Terminate command block

Domain: **PROCEDURE**

Command description

END-BLOCK terminates a BEGIN block, i.e. a command block which was initiated with the BEGIN-BLOCK command.

Format

END-BLOCK
BLOCK = ***LAST** / <structured-name 1..255>

Operands

BLOCK =
Designates the BEGIN block to be terminated.

BLOCK = *LAST
Reference to the BEGIN block last opened.

BLOCK = <structured-name 1..255>
Reference to the tag in the BEGIN block last opened; specifying another block tag produces an error message.

Command return code

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	1	SDP0223	Incorrect environment
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	130	SDP0099	No further address space available

Example

See the BEGIN-BLOCK command, page 568.

END-FOR
Terminate FOR block

Domain: **PROCEDURE**

Command description

END-FOR terminates a FOR block, i.e. a FOR loop which was initiated with the FOR command.

When the END-FOR command is executed, the loop variable in the FOR command is assigned the next element in the value list. Execution then continues with the first command after the FOR command. Once the value list has been completely processed, the loop is terminated: procedure execution resumes with the command following the END-FOR command. (Full details are contained in section “Defining loops” on page 96.)

Format

END-FOR
BLOCK = ***LAST** / <structured-name 1..255>

Operands

BLOCK =
Designates the FOR block to be terminated.

BLOCK = *LAST
Reference to the FOR-BLOCK last opened.

BLOCK = <structured-name 1..255>
Reference to the tag of the FOR block last opened; specifying another block tag produces an error message.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	1	SDP0139	ack branch limit reached
	1	SDP0223	Incorrect environment
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available

Example

See the FOR command, page 659.

END-IF
Terminate IF block

Domain: **PROCEDURE**

Command description

END-IF terminates blocks with conditional command sequences, i.e.:

– IF block
– IF-BLOCK-ERROR block
– IF-CMD-ERROR block

Procedure execution then resumes with the command following END-IF. (Full details are contained in section “Defining conditional branches” on page 93.)

Format

END-IF
BLOCK = ***LAST** / <structured-name 1..255>

Operands

BLOCK =
Designates the IF, IF-BLOCK-ERROR or IF-CMD-ERROR block to be terminated.

BLOCK = *LAST
Reference to the IF, IF-BLOCK-ERROR or IF-CMD-ERROR block last opened.

BLOCK = <structured-name 1..255>
Reference to the tag of the IF, IF-BLOCK-ERROR or IF-CMD-ERROR block last opened; specifying another block tag produces an error message.

Command return codes

(SC2) SC1	Maincode	Meaning
0	CMD0001	No error
1	CMD0202	Syntax error
1	SDP0118	Command in false context
1	SDP0223	Incorrect environment
3	CMD2203	Incorrect syntax file
32	CMD0221	System error (internal error)
130	SDP0099	No further address space available

Example

See the IF command, page 672.

END-PARAMETER-DECLARATION
Terminate procedure parameter declaration

Domain: **PROCEDURE**

Command description

END-PARAMETER-DECLARATION terminates the command block which was initiated with the BEGIN-PARAMETER-DECLARATION command; the procedure parameters are declared in this block (see section “Declaring the procedure parameters” on page 89).

Format

END-PARAMETER-**DECL**ARATION

Command return codes

If END-PARAMETER-DECLARATION is used at the end of the procedure head of an S procedure, it is completely evaluated during preparation of the procedure and executed. Any error during execution of the command is thus an error during procedure preparation; the procedure has not been executed when the error occurs. If the command is executed correctly, then parameter transfer is also executed correctly. The following return codes can thus appear only if END-PARAMETER-DECLARATION is used in another (i.e. wrong) context.

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	130	SDP0099	No further address space available

END-STRUCTURE
Identify end of structure declaration

Domain: **PROCEDURE**

Command description

END-STRUCTURE terminates a structure declaration block which was initiated with BEGIN-STRUCTURE.

Format

END-STRUCTURE
NAME = ***LAST** / <structured-name 1..20>

Operands

NAME =
Designates the names of the structure to be terminated.

NAME = *LAST
Reference to the last structure declaration block initiated with BEGIN-STRUCTURE.

NAME = <structured-name 1..20>
Reference to the name of the structure declaration block last opened.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
2	0	CMD0001	Warning; structure is empty
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available

Example

See the DECLARE-ELEMENT command, page 594.

END-WHILE
Terminate WHILE block

Domain: **PROCEDURE**

Command description

END-WHILE terminates a WHILE block, i.e. a loop which was initiated with the WHILE command.
The loop condition in the WHILE command is checked during execution of the END-WHILE command. If the condition is met (TRUE), the first command in the WHILE block is used to start the next loop pass. Otherwise, the loop is terminated. Procedure execution resumes with the first command following END-WHILE (for further details see "WHILE block" on page 98).

Format

END-WHILE
BLOCK = ***LAST** / <structured-name 1..255>

Operands

BLOCK =
Designates the WHILE block to be terminated.

BLOCK = *LAST
Reference to the WHILE block last opened.

BLOCK = <structured-name 1..255>
Reference to the tag in the WHILE block last opened; specifying another block tag produces an error message.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	1	SDP0139	Back branch limit reached
	1	SDP0223	Incorrect environment
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available

Example

See the WHILE command, page 791.

ENTER-PROCEDURE
Start procedure in background as batch job

Domain: **PROCEDURE, JOB**

The command is part of the BS2000/OSD configuration (as of BS2000/OSD V7.0).
As an exception from the other commands, the use of ENTER-PROCEDURE requires either the STD-PROCESSING privileges or the HARDWARE-MAINTENANCE privileges.

Command description

Using the ENTER-PROCEDURE command, the user can start a procedure as a batch job. In contrast to the ENTER-JOB command, the user does not have to create a separate ENTER file. The procedure parameters are consequently variable at every asynchronous procedure execution (background procedure). ENTER files can only be started with the ENTER-JOB command.

Method of operation

1. The procedure file is created as a copy under the name `S.PROC.tsn.date.time`, where *date* has the format yyyy-mm-dd and *time* has the format hh.mm.ss.
2. An ENTER file with the name `S.E.tsn.date.time` and the following contents is created:

   ```
   /SET-LOGON-PARAMETERS
     .
     .
   /CALL-PROCEDURE FROM-FILE=S.PROC.tsn.date.time, -
   /               PROCEDURE-PARAMETERS=(parameter)
     .
     .
   /EXIT-JOB SYSTEM-OUTPUT=option
   ```

   The value of *parameter* corresponds to the entry in the PROCEDURE-PARAMETERS operand. After procedure execution, the copy of the procedure file is deleted.
   The value of *option* corresponds to the entry in the SYSTEM-OUTPUT operand.
3. The ENTER file is started with ENTER-JOB. Entries for the operands PROCESSING-ADMISSION, JOB-CLASS, JOB-NAME, MONJV, JV-PASSWORD, JOB-PRIORITY, RERUN-AFTER-CRASH, FLUSH-AFTER-SHUTDOWN, SCHEDULING-TIME, START, REPEAT-JOB, LIMIT, RESOURCES, LISTING and JOB-PARAMETER are transferred to the ENTER-JOB command.

Format

(part 1 of 2)

```
ENTER-PROCEDURE                                                              Alias: ENP

FROM-FILE = <filename 1..54 without-gen> / *LIBRARY-ELEMENT(...)
  *LIBRARY-ELEMENT(...)
  |  LIBRARY = <filename 1..51 without-gen>
  | ,ELEMENT = <composed-name 1..38>
,PROCEDURE-PARAMETERS = *NO / <text 0..1800 with-low>
,PROCESSING-ADMISSION = *SAME / *PARAMETERS(...)
  *PARAMETERS(...)
  |  USER-IDENTIFICATION = *NONE / <name 1..8>
  | ,ACCOUNT = *NONE / <alphanum-name 1..8>
  | ,PASSWORD = *NONE / <c-string 1..8> / <c-string 9..32> / <x-string 1..16> / *SECRET
,PROCEDURE-PASSWORD = *NONE / <x-string 1..8> / <c-string 1..4> /
                                  <integer -2147483648..2147483647> / *SECRET
,CRYPTO-PASSWORD = *NONE / <c-string 1..8> / <x-string 1..16> / *SECRET
,HOST = *STD / <c-string 1..8> / *ANY
,JOB-CLASS = *STD / <name 1..8>
,JOB-NAME = *NO / <name 1..8>
,MONJV = *NONE / <filename 1..54 without-gen-vers>
,JV-PASSWORD = *NONE / <c-string 1..4> / <x-string 1..8> / *SECRET /
                       <integer -2147483648..2147483647>
,JOB-PRIORITY = *STD / <integer 1..9>
,RERUN-AFTER-CRASH = *NO / *YES
,FLUSH-AFTER-SHUTDOWN = *NO / *YES
,SCHEDULING-TIME = *STD / *PARAMETERS(...) / *BY-CALENDAR(...)
  *PARAMETERS(...)
  |  START = *STD / *SOON / *IMMEDIATELY / *AT-STREAM-STARTUP / *WITHIN(...) / *AT(...) /
  |             *EARLIEST(...) / *LATEST(...)
  |     *WITHIN(...)
  |     |  HOURS = 0 / <integer 0..23 hours>
  |     | ,MINUTES = 0 / <integer 0..59 minutes>
  |     *AT(...)
  |     |  DATE = *TODAY / <date>
  |     | ,TIME = <time>
```

continued ➠

(part 2 of 2)

```
        |     *EARLIEST(...)
        |        |  DATE = *TODAY / <date>
        |        | ,TIME = <time>
        |
        |     *LATEST(...)
        |        |  DATE = *TODAY / <date>
        |        | ,TIME = <time>
        |
        | ,REPEAT-JOB = *STD / *NO / *DAILY / *WEEKLY / *AT-STREAM-STARTUP / *PERIOD(...)
        |
        |     *PERIOD(...)
        |        |  HOURS = 0 / <integer 0..23 hours>
        |        | ,MINUTES = 0 / <integer 0..59 minutes>

    *BY-CALENDAR(...)
        |  CALENDAR-NAME = <filename 1..54 without-gen-vers>
        |
        | ,SYMBOLIC-DATE = <filename 1..20 without-cat-user-vers> /
        |                  <partial-filename 2..20 without-cat-user>

,LIMIT = *STD / <integer 1..32767> / *BY-DATE(...)

    *BY-DATE(...)
        |  DATE = <date>
        |
        | ,TIME = <time>

,RESOURCES = *PARAMETERS (...)

    *PARAMETERS(...)
        |  RUN-PRIORITY = *STD / <integer 30..255>
        |
        | ,CPU-LIMIT = *STD / *NO / <integer 1..32767 seconds>
        |
        | ,SYSLST-LIMIT = *STD / *NO / <integer 0..999999>
        |
        | ,SYSOPT-LIMIT = *STD / *NO / <integer 0..999999>

,LOGGING = *STD / *YES / *NO

,LISTING = *NO / *YES

,JOB-PARAMETER = *NO / <c-string 1..127>

,SYSTEM-OUTPUT = *STD / *PRINT / *DELETE

,ASSIGN-SYSTEM-FILES = *STD / *PARAMETERS(...)

    *PARAMETERS(...)
        |  SYSLST = *STD / *PRIMARY / *DUMMY / <filename 1..54>
        |
        | ,SYSOUT = *STD / *PRIMARY / *DUMMY / <filename 1..54>

,PROTECTION = *NONE / *CANCEL
```

Operands

FROM-FILE = *LIBRARY-ELEMENT(...) / <filename 1..54 without-gen>
Name of the file or PLAM library element which contains the procedure.
The procedure must not begin with the SET-LOGON-PARAMETERS or LOGON command, i.e. it must not be an ENTER file.
If the job submitter is not the file owner (differing user IDs), the file must be accessible (see the operand PROTECTION=PARAMETERS in the CREATE-FILE and MODIFY-FILE-ATTRIBUTES commands).
The job submitter must in any case have at least execution privileges if the file is protected by a basic ACL or GUARDS.
If the file has an execute password, the password must be specified in the PROCEDURE-PASSWORD operand.

FROM-FILE = *LIBRARY-ELEMENT(...)
The procedure is stored in a PLAM library.

LIBRARY = <filename 1..51 without-gen>
Name of the library containing the procedure as an element.

ELEMENT = <composed-name 1..38>
Name of the element.
The sum of the lengths of the library name (excluding the catalog ID and user ID) and the element name must not exceed 39 characters in the case of single-digit catalog IDs. In the case of multi-digit catalog IDs, the maximum number of characters decreases accordingly.

PROCEDURE-PARAMETERS = *NO / <text 0..1800 with-low>
Parameter values which are to be set instead of the appropriate symbolic parameters. Parameter values must be enclosed in parentheses. Input is carried out as described in the CALL-PROCEDURE command.

PROCESSING-ADMISSION =
Specifies the user ID under which the batch job is to run.

PROCESSING-ADMISSION = *SAME
The batch job should run under the current user ID (i.e. the one under which ENTER-PROCEDURE was specified).

PROCESSING-ADMISSION = *PARAMETERS(...)
Parameters defining the LOGON authorization of the destination user ID.

USER-IDENTIFICATION = *NONE / <name 1..8>
User ID under which the batch job should run.

ACCOUNT = *NONE / <alphanum-name 1..8>
Account number of the user ID.

PASSWORD = *NONE / <c-string 1..8> / <c-string 9..32> / <x-string 1..16> / *SECRET
Password of the user ID.
The long password mechanism is supported (<c-string 9..32>). A hash algorithm converts the long password to the internal 8-byte representation. See the MODIFY-USER-PROTECTION command description for details of the long password mechanism.

The operand has the following special characteristics:
- The password entered is not logged.
- The input field is automatically blanked out in the guided dialog.
- In unguided dialog and foreground procedures, the entry *SECRET or ^, SDF provides a blanked out input field for inputting the password .

PROCEDURE-PASSWORD = *NONE / <c-string 1..4> / <x-string 1..8> / <integer -2147483648..2147483647> / *SECRET
Password protecting the procedure file from being executed.
The operand has the following special characteristics:
- The password entered is not logged.
- The input field is automatically blanked out in the guided dialog.
- In unguided dialog and foreground procedures, the entry *SECRET or ^, SDF provides a blanked out input field for inputting the password .

CRYPTO-PASSWORD = *NONE / <c-string 1..8> / <x-string 1..16> / *SECRET
Password used when encrypting the procedure file. The copy of the procedure file (S.PROC file) is decrypted with the aid of the crypto password.
The operand has the following special characteristics:
- The password entered is not logged.
- The input field is automatically blanked out in the guided dialog.
- In unguided dialog and foreground procedures, the entry *SECRET or ^, SDF provides a blanked out input field for inputting the password .

HOST =
Specifies the host the job is to run on.
Operand values other than *STD are available only to users who have the HIPLEX MSCF (multiprocessor systems) software product.

HOST = *STD
The job is to run on the local host.

HOST = <c-string 1..8>
Host name of the host the ENTER job is to run on.

HOST = *ANY
Allowed only on an XCS network. For details see the “HIPLEX MSCF” manual [9].

JOB-CLASS = *STD / <name 1..8>
Job class in which the job is to be placed. The SHOW-USER-ATTRIBUTES or SHOW-JOB-CLASS command can be used to query authorization for the various job classes.

JOB-NAME = *NO / <name 1..8>
Name for the ENTER job. The ENTER job can be addressed using this name (e.g. with SHOW-JOB-STATUS). The name is also printed on the header page of the printer listing. The default is *NO, which means that the ENTER job is given the name of the job issuing the command.

MONJV = *NONE / <filename 1..54 without-gen-vers>
Only possible if the chargeable JV subsystem is loaded
Name of the job variable (JV) that is to monitor the batch job. The user can address the batch job via this JV.

The system sets the JV to appropriate values while the batch job is being processed:

$S	Job in queue
$R	Job running
$T	Job terminated normally
$A	Job terminated abnormally
$M	Job exported with MOVE-JOBS

JV-PASSWORD = *NONE / <c-string 1..4> / <x-string 1..8> / <integer -2147483648..2147483647> / *SECRET
Password of the JV.
The operand has the following special characteristics:
- The password entered is not logged.
- The input field is automatically blanked out in the guided dialog.
- In unguided dialog and foreground procedures, the entry *SECRET or ^, SDF provides a blanked out input field for inputting the password .

JOB-PRIORITY = *STD / <integer 1..9>
Job priority to be given to the batch job.
The lower the value, the higher the priority. The maximum permissible value is defined in the job class definition and may be queried using the SHOW-JOB-CLASS command.

JOB-PRIORITY = *STD
The standard priority specified for the job class applies.

RERUN-AFTER-CRASH = *NO / *YES
Specifies whether the batch job is to be restarted in the next session if processing is aborted on account of a system error or shutdown.

Note

The operand is not evaluated if job repetition is enabled in the REPEAT operand.

FLUSH-AFTER-SHUTDOWN = *NO / *YES
Specifies whether the batch job is to be removed from the job queue if it has not been processed until shutdown.

Note

The operand is not evaluated if job repetition is enabled in the REPEAT operand. FLUSH-AFTER-SHUTDOWN=*YES is rejected for calendar jobs.

SCHEDULING-TIME = *STD / *PARAMETERS(...) / *BY-CALENDAR(...)
Defines how scheduling times are specified for the batch job.

SCHEDULING-TIME = *STD
The default settings for START and REPEAT-JOB scheduling time specifications for the selected job class apply (see the operands of the SCHEDULING-TIME= *PARAMETERS(...) structure).
A batch job run at the operator terminal (console) by the operator is assigned the START and REPEAT-JOB values defined by the operands of the same name in the SET-LOGON-PARAMETERS command in the ENTER file. If there are no values specified there, the default values specified for the job class apply.
Tasks with the OPERATING privilege can configure this default mechanism in the DEFAULT-FROM-FILE operand.

SCHEDULING-TIME = *PARAMETERS(...)
Defines a scheduling time (start time) for the batch job. It is also possible to define repeat jobs.

START =
Starting time for the batch job. Values other than *STD are ignored unless they are permitted by virtue of the job class definition (see SHOW-JOB-CLASS command).

START = *STD
The default value for the selected job class applies.

START = *SOON
The job is to be started as soon as possible, in accordance with its priority.

START = *IMMEDIATELY
The job is to be started immediately.

START = *AT-STREAM-STARTUP
The job is to be started after the next startup of the job scheduler.

START = *WITHIN(...)
The job is to be started within the specified time period.

HOURS = 0 / <integer 0..23 *hours*>
Number of hours.

MINUTES = 0 / <integer 0..59 *minutes*>
Number of minutes.

START = *AT(...)
The job is to be started at exactly the time specified.

DATE = *TODAY / <date>
Date. This can be specified in the form [yy]yy-mm-dd. Only the last two digits of the year are evaluated, which means that the century is ignored in four-digit year specifications. 20 is automatically prefixed to two-digit year specifications < 80, 19 to two-digit year specifications ≥ 80.

TIME = <time>
Time of day in the format hh:mm, where hh = hours and mm = minutes.
A seconds arguments is ignored.

START = *EARLIEST(...)
The job is to be started no sooner than the time specified.

DATE = *TODAY / <date>
Date. This can be specified in the form [yy]yy-mm-dd. Only the last two digits of the year are evaluated, which means that the century is ignored in four-digit year specifications. 20 is automatically prefixed to two-digit year specifications < 80, 19 to two-digit year specifications ≥ 80.

TIME = <time>
Time of day in the format hh:mm, where hh = hours and mm = minutes.
A seconds arguments is ignored.

START = *LATEST(...)
The job is to be started no later than the time specified.

DATE = *TODAY / <date>
Date. This can be specified in the form [yy]yy-mm-dd. Only the last two digits of the year are evaluated, which means that the century is ignored in four-digit year specifications. 20 is automatically prefixed to two-digit year specifications < 80, 19 to two-digit year specifications ≥ 80.

TIME = <time>
Time of day in the format hh:mm, where hh = hours and mm = minutes.
A seconds arguments is ignored.

REPEAT-JOB =
Time interval at which the batch job is to be repeated. Values other than *STD are ignored unless they are permitted by virtue of the job class definition (see the SHOW-JOB-CLASS command). The time base for repetitions depends on the specification in the START operand (see below, “Combinations of the START and REPEAT-JOB operands”). For the repetitions, the following applies:

– The i-th repetition (i ≥ 1) of a job is not started until the (i-1)th repetition has ended.

- Cancellation of the currently executing job (i) has no effect on the start of (i+1); ($i \geq 0$).
- Cancellation of the entire job: both the currently executing job (i) and the subsequent job (i+1) must be canceled, ($i \geq 0$);
(CANCEL-JOB command, or make job (i) the last job in the series using the command MODIFY-JOB...,REPEAT-JOB=*NO).

REPEAT-JOB = <u>*STD</u>
The default value for the selected job class applies.

REPEAT-JOB = *NO
The batch job is not repeated.

REPEAT-JOB = *DAILY
Daily repetition at the time specified with START.

REPEAT-JOB = *WEEKLY
Weekly repetition at the time specified with START.

REPEAT-JOB = *AT-STREAM-STARTUP
Repetition following each startup of the job scheduler.

REPEAT-JOB = *PERIOD(...)
Repetition after the specified time interval.

HOURS = <u>0</u> / <integer 0..23 *hours*>
Number of hours.

MINUTES = <u>0</u> / <integer 0..59 *minutes*>
Number of minutes.

SCHEDULING-TIME = *BY-CALENDAR(...)
The batch job scheduling time and any repeat jobs are specified in the form of a symbolic date defined in a calendar file (calendar job).
The entries in a calendar file can be listed with the SHOW-CALENDAR command. Creation of calendar files with the CALENDAR utility is described in the “Calendar” manual [23].

CALENDAR-NAME = <filename 1..54 without-cat-user-gen-vers>
Name of the calendar file.

SYMBOLIC-DATE = <filename 1..20 without-cat-user-vers> /
<partial-filename 2..20 without-cat-user>
Symbolic date which defines the scheduling time and any repetition cycles within the calendar file. The symbolic date may also be entered as a partially qualified value. In this way, several scheduling times can be defined for one calendar day, if the SYMDATs are defined accordingly.

Example: Definition of SYMDATs in the calendar file:
- WORK.DAY.1 (every other day at 06:00 hrs)
- WORK.DAY.2 (every other day at 18:00 hrs)

– WORK.WEEK.1 (each Friday at 21:00 hrs)
`SYMBOLIC-DATE=WORK.` starts a calendar job that will start all three scheduling times.

LIMIT = *STD / <integer 1..32767> / *BY-DATE(...)
Governs how long a calendar job remains in existence. This limit applies in addition to the limits set by the calendar file.

LIMIT = *STD
The duration of the calendar job depends entirely on the symbolic date entry in the calendar.

LIMIT = <integer 1..32767>
Only allowed for calendar jobs.
Maximum number of repeats for the calendar job.
On termination, a check is performed to determine whether the run counter has reached or exceeded the maximum value. If this is the case, the entire calendar job is terminated. Otherwise the run counter is incremented by one.

LIMIT = *BY-DATE(...)
Only allowed for calendar jobs.
No repeats of the calendar job will be started once the specified date arrives. A repeat which is currently in progress will abort when the date arrives.
The specified date relates only to the calculated starting date for repeat jobs. Overshoots due to rescheduling of postponed repeats or to delays in the job scheduler are allowed.

The date specification consists of the day and the time:

DATE = <date>
Date. This can be specified in the form [yy]yy-mm-dd. Only the last two digits of the year are evaluated, which means that the century is ignored in four-digit year specifications. 20 is automatically prefixed to two-digit year specifications < 80, 19 to two-digit year specifications ≥ 80.

TIME = <time>
Time of day.

RESOURCES = *PARAMETERS(...)
Specifications regarding run priority, CPU time, maximum number of SYSLST and SYSOPT records.

RUN-PRIORITY = *STD / <integer 30..255>
Run priority the batch job is to be given. The lower the value, the higher the priority. The maximum permissible priority value is the lesser of the two values (i.e. the more favorable of the values) from the user catalog and the job class definition.
If no maximum value is defined for the job class, then the following rules apply:

- If the value specified explicitly is smaller than the value in the user entry, message JMS0045 is issued. The batch job is assigned the greater of the two values (i.e. the less favorable value) from a direct comparison of the run priority in the user entry and the standard run priority of the job class.
- If no value is specified or if *STD or *BY-JOB-CLASS is explicitly specified, the batch job is assigned the standard run priority of the job class.

The values can be queried with the SHOW-USER-ATTRIBUTES and SHOW-JOB-CLASS commands.

RUN-PRIORITY = <u>*STD</u>
The standard priority specified for the job class applies.

CPU-LIMIT = <u>*STD</u> / *NO / <integer 1..32767 *seconds*>
Maximum CPU time, in seconds, that the batch job may consume. The maximum time permitted depends on the job class specified. See also section “Time limits in BS2000” in the “Commands, Vol. 1-5” manual [3].

CPU-LIMIT = <u>*STD</u>
The default value for the selected job class applies.

CPU-LIMIT = *NO
The ENTER job is to run with no time limit (NTL). This operand value is permitted only if the corresponding authorization exists (user entry).

SYSLST-LIMIT = <u>*STD</u> / *NO / <integer 0..999999>
Designates the maximum number of records output by the job to the system files SYSLST, SYSLST01, SYSLST02, ..., SYSLST99. Data records in the system file SYSOUT that are simultaneously written to SYSLST are not taken into account.
This value must not exceed the limit defined in the job class definition. This limit may be queried using the SHOW-JOB-CLASS command.

SYSLST-LIMIT = <u>*STD</u>
Default value for the selected job class. If the specified number is exceeded:
- in batch mode, the job is terminated abnormally;
- in interactive mode, the user may specify whether the job is to be continued or terminated. If continued, output is limited by “number” again.

SYSLST-LIMIT = *NO
The number of records is not limited.

SYSOPT-LIMIT = <u>*STD</u> / *NO / <integer 0..999999>
Designates the maximum number of records to be output by the job to the system file SYSOPT.
This value must not exceed the limit defined in the job class definition. This limit may be queried using the SHOW-JOB-CLASS command.

SYSOPT-LIMIT = <u>*STD</u>
Default value of the selected job class. If the specified number is exceeded:

- in batch mode, the job is terminated abnormally;
- in interactive mode, the user may specify whether the job is to be continued or terminated. If continued, output is limited by “number” again.

SYSOPT-LIMIT = *NO
The number of records is not limited.

LISTING = *NO / *YES
Specifies whether job execution is also to be logged on SYSLST.

LOGGING =
Controls logging of job execution. For non-S procedures the LOGGING operand is ignored because logging is defined in the BEGIN-PROCEDURE in that instance.

LOGGING = *STD
Logging only takes place if the procedure file is not read-protected.

LOGGING = *YES
Job execution is logged. Execution of a file can only be logged if the correct password has been entered in the password table of the job originator by means of the ADD-PASSWORD command or is specified in the PROCEDURE-PASSWORD operand.

LOGGING = *NO
Job execution is not logged.

JOB-PARAMETER =
Specification of additional attributes for the selected job class, if such attributes have been defined and made known by the system administrator.

JOB-PARAMETER = *NO
No additional attributes.

JOB-PARAMETER = <c-string 1..127>
Arbitrary sequence of characters defined by the system administrator to identify additional job class attributes.

SYSTEM-OUTPUT =
Controls output of the system files SYSLST and SYSOUT on job termination (see also the EXIT-JOB command, SYSTEM-OUTPUT operand).

SYSTEM-OUTPUT = *STD
Outputs the system files SYSLST and SYSOUT to printer in the event of errored execution.

SYSTEM-OUTPUT = *PRINT
Outputs the system files SYSLST and SYSOUT to printer.

SYSTEM-OUTPUT = *DELETE
Output of the system files SYSLST and SYSOUT is suppressed.

ASSIGN-SYSTEM-FILES =
Specifies which allocation should apply to the system files SYSLST and SYSOUT at the start of the batch job.
The operand supports the assignment of cataloged files to the two system files during an asynchronous procedure run. Consequently the procedure file can, for instance, continue output to SYSOUT in interactive mode while directing output to a cataloged file in batch mode. The assignment to dummy files (*DUMMY) is also supported (see also FILE-NAME=*DUMMY in the ADD-FILE-LINK command).

ASSIGN-SYSTEM-FILES = <u>*STD</u>
The primary allocation for SYSLST and SYSOUT is preset, i.e. output on printer at the end of the task (dependent on SYSTEM-OUTPUT), if there has been no change in allocation within the procedure.

ASSIGN-SYSTEM-FILES = *PARAMETERS(...)
Specifies which allocation should apply to SYSLST and SYSOUT at the start of the batch job.

SYSLST = *STD / *PRIMARY / *DUMMY / <filename 1..54>
Output destination to which the system file SYSLST is to be assigned. The default value is *STD, i.e. the existing assignment is unchanged.

SYSOUT = *STD / *PRIMARY / *DUMMY / <filename 1..54>
Output destination to which the system file SYSOUT is to be assigned. The default value is *STD, i.e. the existing assignment is unchanged.

PROTECTION = <u>*NONE</u> / *CANCEL
Specifies whether the batch job is to be protected against accidental termination with the CANCEL-JOB command.

PROTECTION = <u>*NONE</u>
The batch job is not protected against accidental termination.

PROTECTION = *CANCEL
The batch job is not protected against accidental termination. In interactive jobs that wish to terminate this batch job with the CANCEL-JOB command, the system additionally requests confirmation. Accidental termination of the batch job due to incorrect specification of the job number should thus be prevented.

Return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	Command executed
2	0	CMD0002	Command executed with warning e.g. DELETE=*YES ignored for repeat job or when a library element is specified
	1	CMD0202	Syntax error in command
	32	CMD0221	System error
	64	JMS0630	Semantic or privileges error e.g. processor, catalog ID, job class unknown; MONJV not accessible
	64	JMS0640	Error on accessing the procedure file e.g. not a SAM or ISAM file, file empty, missing access right
	64	JMS0670	Error in a remote job
	130	JMS0620	No further storage space or TSN available; or specified MONJV is already monitoring a job
	130	JMS0650	MSCF not available, or no connection with the specified processor

Notes

– Combinations of the START and REPEAT-JOB operands:

	REPEAT-JOB		
START	**AT-STREAM-STARTUP**	**DAILY or WEEKLY**	**PERIOD**
IMMEDIATELY or SOON	a)	c)	c)
AT bzw. EARLIEST	a)	d)	f)
LATEST or WITHIN	a)	c)	g)
AT-STREAM-STARTUP	b)	e)	h)

Table 5: Combinations of the START and REPEAT-JOB operands in the ENTER-PROCEDURE command

a) The first and all subsequent starts of the job take place as specified.

b) The first start of the job is effected with START=*AT-STREAM-STARTUP. All further starts take place after the startup of the job scheduler with START=*SOON.

c) The time base for the repetition cycle is the time of job acceptance.

d) The specified point in time (START=...., TIME=....) is the time base for the repetition cycle.

e) The first start of the job follows startup of the job scheduler. This start time is the time base for the repetition cycle. All further starts take place with START=*SOON.

f) The specified point in time (START=...., TIME=....) is the time base for the repetition cycle. The second and all further starts take place with START=*SOON.

g) The time base for the repetition cycle is the time of job acceptance. All further starts take place with START=*SOON.

h) The time base for the repetition cycle is the first start time. The first start of the job follows startup of the job scheduler. All further starts take place with START=*SOON.

– The following applies to the *WITHIN, *AT and *LATEST entries in the START operand: After the specified point in time or period of time, jobs that have not been started are treated in the same way as jobs started with START=*SOON and highest job priority.

  Example

  A job with START=*LATEST(...) could not be started by the desired time because the job scheduler was not active. It will then be started (within the same session) as soon as possible after the next startup of the job scheduler.

– Determining the scheduling time of a calendar job:

  – In the first version of the calendar job, the symbolic date (SYMDAT) specified in the SYMBOLIC-DATE operand is passed on to the CALENDAR component in the evaluation of the job attributes. The CALENDAR component returns the next point in time (date and time), with regard to the current point in time, specified by the SYMDATs defined in the calendar file.
    In the case of partially qualified SYMDATs, a scheduling time is returned for each SYMDAT beginning with the character string, and the relevant number of jobs is started.
  – The scheduling times of the following versions are determined according to the same procedure while the previous jobs are processed.

  As a consequence, any modifications made to the calendar file only take effect on calendar job versions whose scheduling time is determined after the update of the calendar file.
  In particular, the number of calendar job versions started by means of a partially qualified SYMDAT can be extended (by defining new SYMDATs) or reduced (by deleting SYMDATs).

– Job variables (JVs) are available only to users who have the software product JV (see also the “Job Variables” manual [5]).

– A batch job intended to run on a remote host is accessible via a monitoring JV only if the MRSCAT of each host contains the catalog ID of the home pubset of the other host.

– Access to procedure files via RFA is not possible.

– So as to ensure availability of the procedure file at execution time (start of the batch job), a copy of the file with the prefix S.PROC. is always created.

– The remaining operands determine execution of the ENTER file. They are identical with those of the ENTER-JOB command. The following operands of the ENTER-JOB command are not supported:

FILE-PASSWORD	The ENTER file created receives a random password which is used to supply the FILE-PASSWORD operand.
DELETE	DELETE=*YES always applies.

– The ENTER file created is always started with DELETE=*YES.
The ENTER file and the copy of the procedure file (S.PROC file) are not automatically deleted if the job is to be repeated (REPEAT operand).

– The copy of the procedure file (S.PROC file) is password-protected, but may be deleted by the owner without entry of the password. The user can thus delete S.PROC files that were not deleted on account of a system error. The user may not delete S.PROC files for repeat jobs.
The same applies to password protection on the S.E file.

– The S.PROC and S.E files are set up depending on the class 2 system parameter DESTLEV with DESTROY-BY-DELETE=*YES.

– Procedure files can be protected by means of read, write and execute passwords. The execute password or a higher-level password must have been entered in the password table of the calling job, either through specification in the PROCEDURE-PASSWORD operand or previously through an ADD-PASSWORD command.
A read password must be specified if job execution is to be logged.
The passwords are validated on processing the ENTER-PROCEDURE command. If the passwords are subsequently changed, the successful validation during ENTER-PROCEDURE processing remains in force and the procedure file can be executed.
If the procedure to be executed is a PLAM library element, a password specified in the PROCEDURE-PASSWORD operand is interpreted as a password only for accessing the PLAM library.

– S.IN files are always unencrypted. It must be possible to decrypt the copy of the procedure file during command processing, either by specifying the CRYPTO-PASSWORD operand or using the crypto password entered in the crypto password table (see the ADD-CRYPTO-PASSWORD command).
If a batch job is to run on a remote computer, the crypto password can only be specified via the operand. The operand has no meaning on a remote computer with BS2000/OSD < V6.0

– The S.PROC and the S.E files are protected by a file lock for the duration of the batch job.
Note the following: the file locks are set when the pubset is imported on which the files are located. The file locks only consider files to which batch jobs from the current job pool (on the home pubset) refer. If the files are located on a shared pubset, the file locks are coordinated from the master computer.

Example

Procedure call:

```
/ENTER-PROCEDURE P, (PAR1 = 'ABC', PAR2 = 'XYZ')
```

A file called S.E.tsn.date.time is created with the following contents:

```
/SET-LOGON-PARAMETERS
...
/CALL-PROCEDURE S.PROC.tsn.date.time, (PAR1 = ’ABC’,PAR2 = ’XYZ’),LOGGING =
*YES
...
/EXIT-JOB
```

The background procedure is started (SET-LOGON-PARAMETERS). This procedure calls the procedure S.PROC.tsn.date.time (CALL-PROCEDURE S.PROC.tsn.date.time). After execution of the called procedure, the background procedure is terminated (EXIT-JOB).

EXECUTE-CMD
Execute command and structured output

Domain: **PROCEDURE**

Command description

The EXECUTE-CMD command passes the command specified with the operand CMD=... on for execution and writes the command output (messages, output information) to a specified variable. Both structured and unstructured output can be generated, depending on the capabilities of the command server. The reaction to errors occurring in command execution can be controlled by evaluation of the return code.

Most SHOW commands supply structured outputs. The output structures are described in in the manual “Commands, Volume 6” [4] or in the relevant product manual.

Guaranteed messages can be rerouted to variables as structured output. Refer to the manual “MSGMAKER” [21] for a description of the structure and the operation.

Notes

– Commands such as LOAD-/START-EXECUTABLE-PROGRAM (or LOAD-/START-PROGRAM), and similar ones, should not be specified with EXECUTE-CMD. If such commands are nevertheless specified, then all the operands of /EXECUTE-CMD will be ignored, and error message SDP0229 will be returned. The commands will then be executed as though outside EXECUTE-CMD.
– EXECUTE-CMD cannot be used to execute AID commands.

Format

```
EXECUTE-CMD

 CMD = <text 0..1800 with-low>
,TEXT-OUTPUT = *SYSOUT / *NONE / <composed-name 1..255>(...)
    <composed-name 1..255>(...)
      |  WRITE-MODE = *REPLACE / *EXTEND
,STRUCTURE-OUTPUT = *NONE / <composed-name 1..255>(...)
    <composed-name 1..255>(...)
      |  WRITE-MODE = *REPLACE / *EXTEND
,MSG-STRUCTURE-OUTPUT = *NONE / <composed-name 1..255>(...)
    <composed-name 1..255>(...)
      |  WRITE-MODE = *REPLACE / *EXTEND /
,RETURNCODE = *STD / *NONE / *VARIABLE(...)
    *VARIABLE(...)
      |  SUBCODE1 = *NONE / <composed-name 1..255>
      | ,SUBCODE2 = *NONE / <composed-name 1..255>
      | ,MAINCODE = *NONE / <composed-name 1..255>
```

Operands

CMD = <text 0...1800 with-low>
Command to be executed (enclosed in parentheses and without a slash).

TEXT-OUTPUT =
Output in the form of a text string. If the command to be executed is implemented by a TU (Task Unprivileged) program or by a procedure, it is not possible to ignore the SYSOUT outputs or to reroute them into a variable.

TEXT-OUTPUT = *SYSOUT
Output to SYSOUT.

TEXT-OUTPUT = *NONE
Unstructured output is not to take place.

TEXT-OUTPUT = <composed-name 1..255> (...)
Output to a list variable. Designates the list variable to which output is to take place; the list variable must be declared with MULTIPLE-ELEMENTS = *LIST in the DECLARE-VARIABLE command. An output line of the SYSOUT layout thus corresponds to a list element. The SYSOUT layout is version-dependent and can therefore not be guaranteed.

WRITE-MODE = *REPLACE / *EXTEND
Specifies whether the list is to be overwritten or extended.
*REPLACE means that the list is overwritten.

STRUCTURE-OUTPUT =
Structured output with messages.
No error message is returned if STRUCTURE-OUTPUT is specified for a command that cannot produce structured output, with the exception of the commands SHOW-USER-STATUS and SHOW-SYSTEM-STATUS.

STRUCTURE-OUTPUT = *NONE
No structured output is to be generated. This also excludes structured output to an S variable stream for the specified command, irrespective of any assignment made with the ASSIGN-STREAM command.

STRUCTURE-OUTPUT = <composed-name 1.255> (...)
Name of the variable into which the structured output is to be made. This variable must be declared as a list variable (MULTIPLE-ELEMENTS = *LIST in the DECLARE-VARIABLE command).

WRITE-MODE = *REPLACE / *EXTEND
Specifies whether the list is to be overwritten or extended.
*REPLACE means that the list is overwritten.

MSG-STRUCTURE-OUTPUT =
Structured output of the messages.

MSG-STRUCTURE-OUTPUT = *NONE
No structured output is to be generated. This also excludes structured output to an S variable stream for the specified command, irrespective of any assignment made with the ASSIGN-STREAM command.

MSG-STRUCTURE-OUTPUT = <composed-name 1..255>(...)
Name of the variable into which the structured message output is to be made. This variable must be declared as a list variable (MULTIPLE-ELEMENTS = *LIST in the DECLARE-VARIABLE command).
For further details, see section “Structured output in S variables” on page 195.

WRITE-MODE = *REPLACE / *EXTEND
Specifies whether the list is overwritten or extended.
If *REPLACE is specified, the list is overwritten.

RETURNCODE =
Defines the behavior if the command is executed incorrectly.

RETURNCODE = *STD
If the command is executed incorrectly, the system searches for the next IF-BLOCK-ERROR command, and the error handling procedure defined in that command is executed.

RETURNCODE = *NONE
No error information is output and no error handling performed.

RETURNCODE = *VARIABLE(...)
The command return codes are output in variables; error handling is not performed automatically.

SUBCODE1 = *NONE / <composed-name 1..255>
Designates a variable in which subcode1 is output
(subcode1 = error class).
The variable must be declared with TYPE = *ANY or TYPE = *INTEGER.

SUBCODE2 = *NONE / <composed-name 1..255>
Designates a variable in which subcode2 is output
(subcode2 = additional information on subcode1).
The variable must be declared with TYPE = *ANY or TYPE = *INTEGER.

MAINCODE = *NONE / <composed-name 1..255>
Designates a variable in which the error code is output
(maincode = error code).
The variable must be declared with TYPE = *ANY or TYPE = *STRING.

Note

The default value for STRUCTURE-OUTPUT and MSG-STRUCTURE-OUTPUT is in each case *NONE. This means that if these operands are not explicitly specified in an EXECUTE-CMD command, or if this value is assigned to them, no variables will be written to SYSINF and/or SYSMSG, even if an ASSIGN-STREAM command was used to assign a destination to these streams before EXECUTE-CMD was issued. This is because any such assignment is overwritten by “*NONE” (i.e. with a *DUMMY value) as part of the EXECUTE-CMD command.

Command return codes

The return codes depend on the setting of the RETURNCODE operand.

*The following return codes are possible if RETURNCODE = *STD:*

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available
xx	xx	xxxxxxx	other return codes from the executed commands

*The following return codes are possible if RETURNCODE = *NONE / *VARIABLE(...):*

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error (but only in EXECUTE-CMD)
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available

EXIT-BLOCK Terminate processing of command block

Domain: **PROCEDURE**

Command description

EXIT-BLOCK terminates processing of a command block (BEGIN, IF, WHILE, REPEAT block, etc.) and resumes procedure execution at the command following the block termination command. For a BEGIN-Block that was called with a INCLUDE-BLOCK command, execution of the procedure continues at the command that follows the INCLUDE-BLOCK command.

Execution of the EXIT-BLOCK command can be made to depend on a condition.

Notes
- From dialog blocks, you can return only to the dialog level.
- Expression replacement (&...) is not permissible for this command.

Format

EXIT-BLOCK
BLOCK = **<u>*LAST</u>** / ***ALL** / <structured-name 1..255>
,**CONDITION** = **<u>*NONE</u>** / <text 1..1800 with-low *bool-expr*>

Operands

BLOCK =
Designates the block to be terminated.

BLOCK = <u>*LAST</u>
Reference to the next higher block; this block is terminated.

BLOCK = *ALL
Processing of all surrounding blocks is terminated; from an interactive block, the system returns to the interactive level.

BLOCK = <structured-name 1..255>
Reference to the tag of the surrounding block to be terminated.

CONDITION =
Defines the condition for execution of the EXIT-BLOCK command.

CONDITION = *NONE
Command execution is not subject to any condition.

CONDITION = <text 1..1800 with-low *bool-expr*>
The EXIT-BLOCK command is not executed unless the specified Boolean expression is "TRUE".

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	1	SDP0223	Incorrect environment
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available

Example 1

```
/LOOP: WHILE (BED < 9)
:
:
/IF (INP='*END')
/WRITE-TEXT 'Processing stops'
/EXIT-BLOCK LOOP
/END-IF
:
:
/END-WHILE
/"Resume processing here after executing EXIT-BLOCK"
```

Example 2

```
/J=0
/FOR I=('line 1','line 2','line 3','line 4)
/J=J+1
/EXIT-BLOCK BLOCK=*LAST,CONDITION=(J>3)
/SHOW-VARIABLE I
/END-FOR
```

Output:

```
I = line 1
I = line 2
I = line 3
```

The fourth list element is not evaluated.

EXIT-PROCEDURE
Terminate procedure

Domain: **PROCEDURE**

Command description

EXIT-PROCEDURE terminates the current procedure and returns job control to the calling level.

Even if the procedure is terminated normally, i.e. without errors, error information can be passed on. This must be stipulated in the ERROR operand. The specification ERROR=*YES initiates error handling if SUBCODE1 is not zero.

The RESUME-PROGRAM operand can be used to stipulate that an already loaded program is resumed after termination of the procedure.

If the EXIT-PROCEDURE command is missing at the end of the procedure, the procedure is automatically terminated once execution reaches the end of the procedure. If an error occurs, the error code is also returned to the caller.

Format

```
EXIT-PROCEDURE

ERROR = *NO(...) / *YES(...)
   *NO(...)
      |  SUBCODE2 = 0 / <  integer 0..255>
      | ,MAINCODE = CMD0001 / <alphanum-name 7..7>
   *YES(...)
      |  SUBCODE1 = 64 / <integer 0..255>
      | ,SUBCODE2 = 0 / <integer 0..255>
      | ,MAINCODE = SDP0018 / <alphanum-name 7..7>
,RESUME-PROGRAM = *NO / *YES
```

Operands

ERROR =
Return code which is returned to the caller.

ERROR = *NO(...)
The return code with error class “NO-ERROR” is returned. The operands SUBCODE2 and MAINCODE can be used to pass further error information.

SUBCODE2 = 0 / <integer 0..255>
Additional information on the error class. The default is zero, i.e. there is no additional information.

MAINCODE = CMD0001 / <alphanum-name 7..7>
Passes a message key whose significance the caller can determine with the SDF command HELP-MSG-INFORMATION. The default is CMD0001, i.e the procedure was executed without errors.

ERROR = *YES(...)
The caller receives an return code which indicates an error. The error class and additional information can be passed with the operands SUBCODE1, SUBCODE2 and MAINCODE. If SUBCODE1 is not zero, error handling will be initiated in the calling procedure.

SUBCODE1 = 64 / <integer 0..255>
Number indicating the error class.
Error class 64: “SEMANTIC ERROR”.

SUBCODE2 = 0 / <integer 0..255>
Additional information on the error class. The value 0 means that there is no additional information.

MAINCODE = SDP 0018
Default value for the maincode.

MAINCODE = <alphanum-name 7..7>
Passes a message key whose significance the caller can determine with the SDF command HELP-MSG-INFORMATION (see also the manual “System Messages” [15]).

RESUME-PROGRAM =
Defines whether a program just being loaded is resumed at the end of the procedure.

RESUME-PROGRAM = *NO
The program is not resumed at the end of the procedure.

RESUME-PROGRAM = *YES
The program is resumed at the end of the procedure.

Command return codes

With EXIT-PROCEDURE ERROR = *YES, the command can report any given return code to the caller. From the caller's point of view, this is the return code of the CALL-PROCEDURE or INCLUDE-PROCEDURE command. If, however, execution of the EXIT-PROCEDURE command itself results in an error, control is not returned to the caller; instead, one of the following return codes is passed and the error handling within the procedure is initiated.

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	130	SDP0099	No further address space available

FOR
Initiate FOR block

Domain: **PROCEDURE**

Command description

The FOR command initiates a FOR block, which subsequently ends with the /END-FOR command. The entire construct is a FOR loop. The user defines a control variable <composed-name 1..255> =; the number of loop passes and the value of the control variable depend on the assignment to the right of the equal sign (expression, list variable (*LIST), counter):

- *LIST(...)
  The number of loop passes is determined by the number of variable elements. With each loop pass, the control variable is assigned the value of the next list element, the elements of the list variable being processed in ascending order.
- *COUNTER(...)
  The number of loop passes is determined by the initial value (FROM=), the final value (TO=) and the increment (INCREMENT=). The control value always contains the current loop value.
- <text 0..1800 with-low>
  The number of loop passes is determined by the number of elements in the value list. With each loop pass, the control variable is assigned the next expression (string, arithmetic, Boolean, ... expression), working from left to right.

If a mixed list containing <text 0..1800 with-low>, *LIST(...) and *COUNTER(...) values is specified, it will be processed from left to right.

Expression replacement (&...) in any of the FOR operands takes place only upon entering the FOR loop and not with each loop pass.

Note

The operands of the FOR command are evaluated by SDF-P and are to be entered as shown in the following. The SDF alias rules apply to the operands. SDF functions that output information on possible operand values or correction dialogs are not available at the operand level. SDF only provides one input field with “# =” in interactive dialogs.

See also section “Defining loops” on page 96 for more information on FOR blocks.

Format

```
FOR
--------------------------------------------------------------------------------------------
<composed-name 1..255> = list-poss(2000): *LIST(...) / *COUNTER (...) / <text 0..1800 with-low expr>

    *LIST(...)
      |  LIST-NAME = <composed-name 1..255>

    *COUNTER(...)
      |  FROM = <text 0..1800 arithm-expr>
      |  ,TO = *UNLIMITED / <text 0..1800 arithm-expr>
      |  ,INCREMENT = 1 / <text 0..1800 arithm-expr>

,CONDITION = *NONE / <text 0..1800 with-low bool-expr>
```

Operands

<composed-name 1..255> =
Designates the control variable. The control variable must have the same type as the elements to the right of the equals sign, or it must be convertible.

<composed-name 1..255> = *LIST(...)
A list variable is assigned to the control variable. The number of loop passes is determined by the number of variable elements. With each loop pass, the control variable is assigned the value of the next list element, the elements of the list variable being processed in ascending order.
If the control variable and the list variable are of type “structure” and contain elements with identical names, then the control variable elements are overwritten by the list variable elements with the same names (see the operand WRITE-MODE = *REPLACE of the SET-VARIABLE command (page 743), section “Lists” on page 140, and chapter “Expressions” on page 249).

LIST-NAME=<composed-name 1..255>
Name of the list variable.

<composed-name 1..255> = *COUNTER (...)
The number of loop passes is determined by the initial value (FROM=), the final value (TO=) and the increment (INCREMENT=). These values cannot be modified once execution has started. The control value always contains the current loop value. The value of the control variable can be modified by direct assignment during the loop pass. The control variable must have the type INTEGER, or it must be convertible. See page 265 for more information about arithmetic expressions.

FROM = <text 0..1800 *arithm-expr*>
Initial value of the control variable (numeric or arithmetic expression).

TO= *UNLIMITED / <text 0..1800 *arithm-expr*>
Final value of the control variable (numeric or arithmetic expression).
*UNLIMITED means: the final value is 2^{31}-1 if a value > 0 was specified for the increment, or -2^{31} for an increment value < 0. Exceeding the limit values results in an error (branch to error handling).

INCREMENT = 1 / <text 0..1800 *arithm-expr*>
Increment by which the current loop value is increased with each loop pass (numeric or arithmetic expression). The (positive/negative) sign determines whether the increment is positive or negative. Incrementing must, however, lead up to the final value, otherwise the loop pass will not be executed.
If the increment changes its sign during the loop pass, the loop pass is aborted.

Note
INCREMENT = 0 will cause loop passes to be executed until the specified condition (Operand CONDITION=...) is no longer met or until EXIT-BLOCK is issued.

<composed-name 1..255> = <text 0..1800 with-low *expr*>
Any type of expression is permitted here (string, arithmetic, Boolean expression, ...).
The number of loop passes is determined by the number of elements in the value list. With each loop pass, the control variable is assigned the next expression, working from left to right.

CONDITION = *NONE / <text 0..1800 *bool-expr*>
Defines the condition checked at the beginning of each loop pass. The FOR loop is executed if the condition is “TRUE”. The FOR loop is aborted if the condition is “FALSE”.
*NONE means: no condition is evaluated, the loop is always executed. See section “Boolean constants” on page 251 for more information about Boolean expressions.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	1	SDP0223	Incorrect environment
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available

Example 1

```
/DECLARE-VARIABLE A,MULTIPLE-ELEMENTS=*LIST
/DECLARE-VARIABLE I
/SET-VARIABLE A=1436,WRITE-MODE=*EXTEND
/A=1455,WRITE-MODE=*EXTEND
/A=1577,WRITE-MODE=*EXTEND
/FOR I=*LIST(A)
/CANCEL-JOB JOB-ID=*TSN(&I)
/END-FOR
```

The FOR loop issues the following commands:

```
/CANCEL-JOB JOB-ID=*TSN(1436)
/CANCEL-JOB JOB-ID=*TSN(1455)
/CANCEL-JOB JOB-ID=*TSN(1577)
```

Example 2

```
/FOR I=(5,7,12,2,3),CONDITION=(I<10)
/SHOW-VARIABLE I
/END-FOR
```

Output:

```
I = 5
I = 7
```

List elements 12, 2, 3 are not evaluated.

Example 3

Calculate and output the prime numbers from the range 2 up to the specified number.

```
/       DECL-VAR N(TYPE=*INTEGER)
/       DECL-VAR PRIMARY-NUMBERS(TYPE=*INTEGER),MULT-ELEM=*LIST
/       WR-TEXT 'Please enter integer >= 2!'
/       READ-VAR N,INPUT=*TERMINAL
/       PRIMARY-NUMBERS# = 2
/LOOP1: FOR I = *COUNTER(FROM=3,TO=N,INCREMENT=2)
/LOOP2:     FOR J = *LIST(LIST-NAME=PRIMARY-NUMBERS)
/               CYCLE LOOP1,COND=(I MOD J == 0)
/           END-FOR LOOP2
/           PRIMARY-NUMBERS = I ,MODE=*EXTEND
/       END-FOR LOOP1
/       WR-TEXT 'Prime numbers from the range 2 to &(N):'
/       SHOW-VAR PRIMARY-NUMBERS,INF=*PAR(NAME=*NONE)
/       EXIT-PROC
```

Example 4

Processing a list variable and including an index.

```
/DECL-VAR LANGUAGES,MULT-ELEM=*LIST
/LANGUAGES = 'German', WR-MODE=*EXTEND
/LANGUAGES = 'English', WR-MODE=*EXTEND
/LANGUAGES = 'French', WR-MODE=*EXTEND
/LANGUAGES = 'Spanish', WR-MODE=*EXTEND
/
/FOR I = *COUNTER(FROM=1,TO=SIZE('LANGUAGES'))
/    WR-TEXT '&(LANGUAGES#I) is the &(I)th language'
/END-FOR
```

Output

```
German is the 1st language
English is the 2nd language
French is the 3rd language
Spanish is the 4th language
```

FREE-VARIABLE
Delete contents of variable

Domain: **PROCEDURE**

Command description

The FREE-VARIABLE command deletes the contents of one or more S variables.

Read access to the variables is impossible after calling FREE-VARIABLE since the variables do not have valid contents any more. The variable declarations are not deleted, however, by this command. The only exception to this are variables with a dynamic structure.

The FREE-VARIABLE command can be used on simple and composed variables. One or more elements can be deleted from list variables with this command. For variables with a dynamic structure, not only the contents are deleted, but also the element itself.

If the S variable is linked to a variable container, then the variable container can also be deleted. However, write access to the variable is not possible anymore.

Format

```
FREE-VARIABLE

 VARIABLE-NAME = <structured-name 1..20 with-wild(40)> / *LIST(...) /
                     list-poss(2000): <composed-name 1..255>

*LIST(...)
      |  LIST-NAME = <composed-name 1..255>
      | ,FROM-INDEX = *FIRST / *LAST / <integer 1..214783647>
      | ,NUMBER-OF-ELEMENTS = 1 / *REST / <integer 1..214783647>

,DESTROY-CONTAINER-JV = *NO / *YES
```

Operands

VARIABLE-NAME =
Specifies the variable whose content is to be deleted.

VARIABLE-NAME = <structured-name 1..20 with-wild(40)>
Deletes the content of the variables whose names match the specified search pattern.

VARIABLE-NAME = *LIST(...)
Deletes the contents of elements in a list variable.

LIST-NAME = <composed-name 1..255>
Name of the list variable.

FROM-INDEX = *FIRST / *LAST / <integer 1..2147483647>
Index of the element of the list variable starting at which the specified number of list elements are to be deleted.
*FIRST: The deletion process starts with the first element of the list (default).
Specifying *LAST causes the last element in the list to be deleted. In this case the NUMBER-OF-ELEMENTS operand is ignored.

NUMBER-OF-ELEMENTS = 1 / *REST / <integer 1..2147483647>
Number of list elements to be deleted.
Default value: One element will be deleted.
Specifying *REST causes all elements to be deleted from the specified start element (FROM-INDEX operand) to the last element in the list.

VARIABLE-NAME = list-poss(2000): <composed-name 1..255>
Deletes the contents of simple or composed variables which are contained in the specified names list.

DESTROY-CONTAINER-JV = *NO / *YES
You can define whether or not the variable container linked to the variable is to be deleted or not. (For example, if an S variable is linked to a job variable as a variable container, then the job variable will be deleted from the file catalog if DESTROY-CONTAINER-JV = *YES. The S variable cannot be write-accessed anymore).

If the variable is not linked to a variable container, then the operand is ignored.

Rules for deleting list variables and other composed variables

– If a name designates a composed variable, then the contents of all variable elements are deleted. The attributes of the composed variables (TYPE, SCOPE,...) are still valid, however. Newly created elements of this variable must correspond to those of the original declaration.

Example

```
/DECLARE-VARIABLE L(TYPE=*STRING),MULTIPLE-ELEMENT=*LIST
/L='ELEM1',WRITE-MODE=*EXTEND
/L='ELEM2',WRITE-MODE=*EXTEND
/FREE-VARIABLE L     &* all elements deleted but declaration remains
/SHOW-VARIABLE L     &* Nothing displayed
/L=3,WRITE-MODE=*EXTEND
%  SDP1036 VARIABLE TYPE AND VALUE TYPE DO NOT MATCH
```

– If a name designates a dynamic structure, then all element declarations are deleted and removed.

Example
```
/DECLARE-VARIABLE S(TYPE=*STRUCTURE(DEFINITION=*DYNAMIC))
/S.ELEM1=1; S.ELEM2='ELEM2'; S.ELEM3 =TRUE;
/FREE-VARIABLE S     &* all element declarations are deleted
/S.ELEM1='ELEM1'     &* no type retained, so modification possible
```

– If the elements of a composed variable are dynamic structures, then the information “element is of TYPE = *STRUCTURE(*DYNAMIC)” is retained (the elements of the dynamic structure itself are deleted and removed).
If the elements are statically structured, then the declaration of the structure layout remains valid. The same is also true for lists.

Example
```
/DCV L1(TYPE=*STRUCTURE(*DYNAMIC)),MULTIPLE-ELEMENT=*LIST
/L1#.ELEM1=1; L1#.ELEM2='ELEM2'; L1#.ELEM3=TRUE
/FREE-VARIABLE L1#
/L1#.ELEM1='ELEM1'    &* no type retained, so modification possible
/
/BEGIN-STRUCTURE S
/DECLARE-ELEMENT ELEM1(TYPE=*STRING)
/DECLARE-ELEMENT ELEM2(TYPE=*INTEGER)
/END-STRUCTURE
/DCV L2(TYPE=*STRUCTURE(S)),MULTIPLE-ELEMENT=*LIST
/L2#.ELEM1='ELEM1';L2#.ELEM2=2
/FREE-VARIABLE L2#     &* Elements deleted but declaration remains
/L2#.ELEM1=1
%  SDP1036 VARIABLE TYPE AND VALUE TYPE DO NOT MATCH
```

– If the composed variable that the FREE-VARIABLE command is applied to is an element of a dynamic structure, then all its elements and the composed variable itself are deleted. This means that explicitly declared arrays, list variables and static structures can also be deleted. If such composed variables with the same properties are to be reproduced, then they must be redeclared.

Example
```
/DCV S(TYPE=*STRUCTURE(*DYNAMIC))
/S.L#.ELEM1=1; S.L#.ELEM2='ELEM2'; S.L#.ELEM3=TRUE;
/FREE-VARIABLE S.L
/SHOW-VARIABLE S.L
```

– List variables are renumbered after elements are deleted so that the numbering is continuos starting with 1.

Example
```
/DCV L1(TYPE=*INTEGER),MULTIPLE-ELEMENT=*LIST
/L1=1,W-M=*EXT; L1=2,W-M=*EXT; L1=3,W-M=*EXT; L1=4,W-M=*EXT;
/FREE-VARIABLE L1#2
/SHOW-VARIABLE L1,LIST-INDEX-NUMBER=*YES
L1#1 = 1
L1#2 = 3
L1#3 = 4

/DCV L2(TYPE=*INTEGER),MULTIPLE-ELEMENT=*LIST
/L2=*STRING-TO-VARIABLE('(1,2,3,4)')
/SHOW-VARIABLE L2,LIST-INDEX-NUMBER=*YES
L2#1 = 1
L2#2 = 2
L2#3 = 3
L2#4 = 4
/FREE-VARIABLE (L2#1,L2#2)&* Use *LIST syntax to free elem 1 / 2
/SHOW-VARIABLE L2,LIST-INDEX-NUMBER=*YES
L2#1 = 2
L2#2 = 4
```

Command return codes

(SC2)	SC1	Maincode	Meaning/Guaranteed messages
	0	CMD0001	No error
1	0	CMD0001	Warning: no elements deleted; variable has already been deleted
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error Guaranteed messages: SDP1008
	130	SDP0099	No further address space available

Examples

```
/DECLARE-VARIABLE B(100,*INTEGER), SCOPE=*PROCEDURE
/FREE-VARIABLE B
/B = 33
/FREE-VARIABLE B ————————————————————————————————————————————————————  (1)
/SHOW-VARIABLE B ————————————————————————————————————————————————————  (2)
```

(1) The FREE-VARIABLE command deletes the value. After that, variable B can only be write-accessed.

(2) No output since the variable does not have a valid value anymore.

Deletion with wildcards specified

```
/TSDP-023-003-0001-1 = ' ALPHA '
/TSDP-023-003-0001-2 = ' BETA  '
/TSDP-023-003-0001-3 = ' GAMMA '
/FREE-VAR T<R:T><A:E><N:Q>-02/-*-<1,3>
/SHOW-VARIABLE
TSDP-023-003-0001-2 =  BETA
*END-OF-CMD
```

Deletion with list elements

```
/DECLARE-VARIABLE T-LIST (TYPE= *STRING), MULTIPLE-ELEMENTS= *LIST
/                 T-LIST#1    = 'FIRST'
/                 T-LIST#2    = 'SECOND'
/                 T-LIST#3    = 'THIRD'
/                 T-LIST#4    = 'FOURTH'
/                 T-LIST#5    = 'ANTEPENULTIMATE'
/                 T-LIST#6    = 'PENULTIMATE'
/                 T-LIST#7    = 'LAST'
/FREE-VARIABLE *LIST (LIST-NAME=T-LIST,FROM-INDEX=*LAST) ——————————————  (1)
/SHOW-VARIABLE T-LIST ————————————————————————————————————————————————  (2)
T-LIST(*LIST) = FIRST
T-LIST(*LIST) = SECOND
T-LIST(*LIST) = THIRD
T-LIST(*LIST) = FOURTH
T-LIST(*LIST) = ANTEPENULTIMATE
T-LIST(*LIST) = PENULTIMATE
/FREE-VARIABLE *LIST(LIST-NAME=T-LIST,FROM-INDEX=4,NUMBER-OF-ELEM= 2 )    (3)
/SHOW-VARIABLE T-LIST ————————————————————————————————————————————————  (4)
T-LIST(*LIST) = FIRST
T-LIST(*LIST) = SECOND
T-LIST(*LIST) = THIRD
T-LIST(*LIST) = PENULTIMATE
```

(1) FROM-INDEX=*LAST in the FREE-VARIABLE command deletes the last element in the list T-LIST.

(2) The SHOW-VARIABLE command displays the 6 elements still remaining in the list.

(3) 2 elements in the list are to be deleted starting at the 4th element.

(4) The SHOW-VARIABLE command displays the 4 elements still remaining in the list. Elements 4 and 5 which previously existed are no longer present.

GOTO
Branch to tag

Domain: **PROCEDURE**

Command description

GOTO branches to the specified tag and continues procedure execution that location. The tag must be defined in the same or a surrounding block (see Example 2). In addition, the tag must be unique to the procedure. However, it should not be placed before a block. (See also the section “Random branch destinations” on page 104 for more on GOTO)

Format

GOTO
LABEL = <structured-name 1..255>

Operands

LABEL = <structured-name 1..255>
Name of the tag where procedure execution is to resume; the name is specified without the terminating colon.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	1	SDP0139	Back branch limit reached
	1	SDP0223	Incorrect environment
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	130	SDP0099	No further address space available

Example 1

```
.
.
.
/GOTO TAG10
.
.
.
/TAG10: CREATE-FILE ...
```

Example 2

```
/LOOP1: WHILE (A < B)
/ADD1:      X = X + A
/LOOP2:     WHILE (X < Y)
/ADD2:          A = A + 1
/               GOTO ADD1                "Permitted, because outer loop"
/           END-WHILE LOOP2
/           GOTO ADD2                    "Forbidden, because inner loop"
/       END-WHILE LOOP1
```

IF
Initiate IF block

Domain: **PROCEDURE**

Command description

The IF command initiates an IF block, i.e. a conditional command sequence: If the condition in the IF command is met, the command sequence following the IF command is executed. Otherwise, the system searches for other ELSE-IF or ELSE commands in the current IF block. If the current IF block does not contain any ELSE-IF or ELSE commands, procedure execution resumes with the command following the appropriate END-IF (see section “Defining conditional branches” on page 93).

Format

IF
CONDITION = <text 0..1800 with-low *bool-expr*>

Operands

CONDITION = <text 0..1800 with-low *bool-expr*>
Logical expression as the condition for executing the command sequence between the IF and ELSE-IF or ELSE command (see chapter “Expressions” on page 249 for information on logical expression).
If the logical expression includes a single ’=’ sign, then this must be enclosed in parentheses.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	1	SDP0223	Incorrect environment
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available

Example

```
/A = 2
/B = 3
/IF (A = B)
/WRITE-TEXT 'A AND B ARE CORRECTLY INITIALIZED'
/ELSE-IF (A > B)
/WRITE-TEXT 'A IS TOO LARGE'
/ELSE
/WRITE-TEXT 'B IS TOO LARGE'
/END-IF
B IS TOO LARGE
```

Note on the parentheses in the logical expressions:

(1) The parentheses can be omitted when the “=” sign is duplicated: `/IF  A==B`

(2) The parentheses can be omitted: `/ELSE-IF  A > B`

IF-BLOCK-ERROR
Initiate block error handling

Domain: **PROCEDURE**

Command description

IF-BLOCK-ERROR initiates a command sequence which is executed in the following instances:

- if an error occurred in the current block
- if an error occurred in a block nested in the current block and was not intercepted before leaving this nested block.

An ELSE branch can be defined in the IF-BLOCK-ERROR block, using the ELSE command. The IF-BLOCK-ERROR block is terminated with the END-IF command.

If the IF-BLOCK-ERROR command is called even though an error did not occur, the ELSE branch is executed - if present - or command execution resumed after the related END-IF command. If the command return code is to be evaluated when no error has occurred, either in the ELSE branch or after the END-IF command, a SAVE-RETURNCODE command must be entered before the IF-BLOCK-ERROR command.

Format

IF-BLOCK-ERROR

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	1	SDP0223	Incorrect environment
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	130	SDP0099	No further address space available

Example

```
/BL1: BEGIN-BLOCK
/...
/    BL2: BEGIN-BLOCK
/    ...
/    “Error 1 occurs here”
/    ...
/    END-BLOCK BLOCK = BL2
/...
/“Error 2 occurs here”
/...
/IF-BLOCK-ERROR ... “The error is intercepted here”
/...
/END-IF
/END-BLOCK BLOCK = BL1
```

Block BL2 is nested in block BL1. Error handling is not carried out in block BL2.
Errors that occur in block BL2 are intercepted by block BL1 in the IF-BLOCK-ERROR block just like errors that occur in Block BL1.

IF-CMD-ERROR
Initiate command error handling

Domain: **PROCEDURE**

Command description

IF-CMD-ERROR initiates a command sequence which is executed when an error occurs in the directly preceding command. This permits specific error handling for this command, thus avoiding block error handling.

IF-CMD-ERROR is ignored after the following commands:

- IF / END-IF
- FOR
- WHILE
- REPEAT
- BEGIN-BLOCK
- GOTO
- CYCLE
- EXIT-BLOCK

The ELSE command can be used in the IF-CMD-ERROR block to define an ELSE branch. In addition, a SAVE-RETURNCODE command is executed implicitly in the ELSE branch, which means that the latest return code from the command is available even if the command was executed without errors. The IF-CMD-ERROR block is terminated with the END-IF command.

If IF-CMD-ERROR is called even though an error did not occur, the ELSE branch is executed - if present - or command execution resumed after the corresponding END-IF.

Format

IF-CMD-ERROR

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	1	SDP0223	Incorrect environment
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	130	SDP0099	No further address space available

Example

See section “Error handling” on page 69.

Note

The command preceding IF-CMD-ERROR and the IF-CMD-ERROR block effectively form a BEGIN block.

IMPORT-VARIABLE
Import variable

Domain: **PROCEDURE**

Command description

The IMPORT-VARIABLE command is used to import a previously declared variable into the called procedure. It is equivalent in many ways to DECLARE-VARIABLE, but eliminates the need to repeatedly list all assigned attributes.

Format

```
IMPORT-VARIABLE
-------------------------------------------------------------------------------------------
 VARIABLE-NAME = <structured-name 1..20 with-wild(40)> / list-poss(2000): <structured-name 1..20>
,FROM = *SCOPE(...)
    *SCOPE(...)
     |   SCOPE = *TASK / *CALLING-PROCEDURES
```

Operands

VARIABLE-NAME =
Name of a variable outside the procedure which is to be imported into the current procedure.

VARIABLE-NAME = <structured-name 1..20 with-wild(40)>
Name of a variable outside the procedure which is to be imported into the current procedure.
If the name contains wildcards, all variables are imported whose names match the search pattern specified. If a wildcard string matches no variable name, mesage SPD0519 is issued.

VARIABLE-NAME = list-poss(2000): <structured-name 1..20>
One or more names of variables which are to be imported into the current procedure. When specified as a list, the variables are imported in the specified order.

FROM= *SCOPE(...)
Specifies the scope of the variable to be imported.

SCOPE = *TASK
The search for the variable to be imported is carried out within the entire task.

SCOPE = *CALLING-PROCEDURES
The search for the variable to be imported is carried out within the called procedure. The variable must have been declared in the procedure with IMPORT-ALLOWED=*YES. The search starts in the called procedure and can be continued upwards to the first procedure (in the event of a background procedure) or to the dialog level (in the event of a foreground procedure).
(If the variable to be imported already exists in the procedure (i.e. is visible), no action is performed; in this case it is irrelevant whether the variable was declared with IMPORT-ALLOWED = *YES or *NO).

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
1	0	CMD0001	Warning: element already declared
2	0	SDP2000	Warning: not all elements of the input list could be processed successfully. Guaranteed message: SDP2000
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error Guaranteed messages: SDP1008, SDP1018
	64	SDP2001	None of the elements could be imported
	130	SDP0099	No further address space available

Examples

```
/DECLARE-VARIABLE VERBOSE-MODE(TYPE=*BOOLEAN, INITIAL-VALUE= YES),-
/                  SCOPE=*CURRENT(IMPORT-ALLOWED=*YES)
/LEVEL=1
/CALL-PROCEDURE MY-PROCEDURE
/           --->IMPORT-VARIABLE VERBOSE-MODE, -
/               FROM=*SCOPE(*CALLING-PROCEDURES)
/
/               LEVEL=2
/               IF (VERBOSE-MODE)
/                  WRITE-TEXT 'CURRENT PROCEDURE AT LEVEL &(LEVEL).'
/               END-IF
/           <---EXIT-PROCEDURE
/IF (VERBOSE-MODE)
/    WRITE-TEXT 'CURRENT PROCEDURE AT LEVEL &(LEVEL).'
/END-IF
```

The called procedure ’MY-PROCEDURE’ has access to the variable ’VERBOSE-MODE’ via the calling procedure as a result of using IMPORT-VARIABLE.
Each of the procedures has a local variable named 'LEVEL'.

See section “Scope of variables” on page 157.

```
/DECLARE-VARIABLE TVAR-1 (TYPE= *STRING, INIT-VAL='TV1')
/DECLARE-VARIABLE TVAR-2 (TYPE= *STRING, INIT-VAL='TV2')
/DECLARE-VARIABLE TVAR-3 (TYPE= *STRING, INIT-VAL='TV3')
/CALL-PROCEDURE IMPORT-TV-LIST ----------------------------------------  (1)
      --> /SET-PROCEDURE-OPTIONS
          /IMPORT-VARIABLE (TVAR-1, TVAR-2) -
          /      ,FROM=*SCOPE(SCOPE=*CALLING-PROCEDURE)
          /SHOW-VARIABLE
          /EXIT-PROCEDURE
TVAR-1 = TV1
TVAR-2 = TV2
*END-OF-CMD
/CALL-PROCEDURE IMPORT-TV-WILDCARD ------------------------------------  (2)
      --> /SET-PROCEDURE-OPTIONS
          /IMPORT-VARIABLE TVAR-<1,3> -
          /      ,FROM=*SCOPE(SCOPE=*CALLING-PROCEDURE)
          /SHOW-VARIABLE
          /EXIT-PROCEDURE
TVAR-1 = TV1
TVAR-3 = TV3
*END-OF-CMD
```

(1) The IMPORT-TV-LIST procedure uses list specification to import the variables TVAR-1 and TVAR-2.

(2) The IMPORT-TV-WILDCARD procedure uses wildcard specification to import the variables TVAR-1 and TVAR-3.

INCLUDE-BLOCK
Executing a BEGIN block as a subprocedure

Domain: **PROCEDURE**

Command description

The INCLUDE-BLOCK command jumps to the BEGIN-BLOCK command with the specified tag (name). After executing the (next) END-BLOCK or EXIT-BLOCK command, execution returns to the command following the INCLUDE-BLOCK command. (See the section “The beginning of a block as a jump destination” on page 101 for more on BEGIN blocks.)

The INCLUDE-BLOCK command allows you to comfortably execute a subprocedure defined between the BEGIN-BLOCK and END-BLOCK commands. In this case the following rules must be observed:

- /INCLUDE-BLOCK jumps to only one /BEGIN-BLOCK with a tag (name). In order to jump to other commands with a tag, you must use the GOTO command.
- The BEGIN block with the subprocedure should be placed after the EXIT-PROCEDURE command, of course. This will avoid the unwanted execution of the subprocedure while the procedure is executing.
- The subprocedure runs in the same procedure environment as its parent procedure (i.e. in the same variables context, with the same SYSFILE environment, etc.).
- The subprocedure may not be called recursively.
- Procedures may be nested although this should be avoided as it will make the procedure complicated and avoid the unwanted execution of BEGIN blocks. The procedure can jump directly to the subprocedures regardless of the level of nesting.
- The GOTO command can be used in the subprocedure, but the subprocedure itself should only branch to tags within the subprocedure because otherwise execution will leave the subprocedure.
- The tags next to the BEGIN-BLOCK command must be unique within the entire procedure.

The INCLUDE-BLOCK command is rejected in the dialog mode.

Format

INCLUDE-BLOCK
BLOCK = <structured-name 1..255>

Operands

BLOCK = <structured-name 1..255>
Name of the BEGIN block that is to be executed as a subprocedure.
The name is to be entered without the colon at the end.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	1	SDP0223	Incorrect environment
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)

Example

```
/ASSIGN-SYSLST TO=PROT.EINGABE,SYSLST-NUMBER=1
/...
/IF (EING='*START')
/ INCLUDE-BLOCK INFO-1
/END-IF &* Execution continues here after the subprocedure INFO-1 is complete
/...
/IF (EING='*END')
/ INCLUDE-BLOCK INFO-1
/END-IF &* Execution continues here after the subprocedure INFO-1 is complete
/...
/...
/...
/INFO-1: BEGIN-BLOCK    &* Beginning of the subprocedure INFO-1
/      WRITE-TEXT '&(TIME()): You have entered &(EING)',OUTPUT=*SYSLST(1)
/END-BLOCK &*End + jump back to the command line following INCLUDE-BLOCK
```

INCLUDE-CMD
Call command sequence from program

Domain: **PROCEDURE**

Command description

The INCLUDE-CMD command is used to include a command or command sequence from a program for execution. The command can be executed only in the CMD macro (TU program) and in EXECUTE-SYSTEM-CMD statements.

You should make a note of the following: during execution of INCLUDE-CMD, the system rejects a number of operations, such as program start, restart and termination, in order to avoid possible inconsistencies since the command is called in program mode (see notes below for more details).

Output of the command to SYSOUT is divided into two parts:

– Output of the result of the analysis of the INCLUDE-CMD command: this can be saved in the SYSOUT buffer of the CMD macro (error report or other types of information).

– Output of command execution currently contained in SYSOUT: it is independent of the CMD call parameter.
Output of this command to SYSOUT can be influenced by means of the ASSIGN-SYSOUT command provided ASSIGN-SYSOUT is contained in the command list specified with CMD=... . .

Format

INCLUDE-CMD	**Alias: INCMD**
CMD = <text 0..1800 with-low>	

Operands

CMD = <text 0...1800 with-low>
Command or command sequence to be executed. Commands in a sequence must be separated by semicolons and must be enclosed in parentheses. A leading slash (/) is not permitted.

Execution of the command list specified in the CMD operand is identical to execution of an include procedure containing the same commands, albeit with certain restrictions:

- Commands normally specified in the procedure head must not be specified in the CMD operand. The commands SET-PROCEDURE-OPTIONS, BEGIN-PARAMETER-DECLARATION, END-PARAMETER-DECLARATION and DECLARE-PARAMETER will therefore be rejected.
- The default values for the logging options apply. These can be modified with MODIFY-PROCEDURE-OPTIONS.
- If the EXIT-PROCEDURE command was specified in the command list for CMD (at the top level), INCLUDE-CMD ends with this command. No subsequent commands are executed.
- The commands DO, ENDP, END-PROCEDURE, ENDP-RESUME cannot be specified in the command sequence in the CMD operand.
- If INCLUDE-CMD is specified in a list of command inputs in the buffer of the CMD macro, any subsequent commands are ignored.
  Example: CMD ’cmd1;cmd2;INCLUDE-CMD CMD=(PRINT-DOCUMENT X);cmd3’
  The input “cmd3” in this example is ignored and is therefore superfluous.

Notes

- The commands specified in the CMD operand are executed in the same manner as input in an S procedure called with INCLUDE-PROCEDURE. They have implicit access to the variables of the current procedure level (just as if they had been called with INCLUDE-PROCEDURE and as if they inherited the system files of the current procedure level (SYSTEM-FILE-CONTEXT=*STD).
- In contrast to INCLUDE-PROCEDURE, INCLUDE-CMD does not terminate a program when it was called by the CMD macro. The program is also not terminated when a procedure is called in the command sequence that is specified in the CMD operand (see the example).
- INCLUDE-CMD may be executed in the CMD macro (TU program) and also in EXECUTE-SYTEM-CMD statements.

- During execution of INCLUDE-CMD, the system rejects the following operations in order to avoid possible inconsistencies since the command is called in program mode:
  - Start and terminate program: START utility, LOAD-/START-EXECUTABLE-PROGRAM (or LOAD-/START-PROGRAM), RESTART-PROGRAM (see CALL-PROCEDURE ...UNLOAD-ALLOWED=*NO).
  - Resume program: AID commands, RESUME-PROGRAM, EXIT-PROCEDURE RESUME-PROGRAM=*YES, ENDP-RESUME, INFORM-PROGRAM, SEND-MSG with TO=*PROGRAM.
  - Abort procedure: CANCEL-PROCEDURE, $\boxed{\text{K2}}$ when prompted for parameters.
  - Call INCLUDE-CMD recursively.
  - BEGIN-BLOCK PROGRAM-INPUT=*MIXED-WITH-CMD
  - SET-JOB-STEP if a program is loaded.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0138	Error during procedure preanalysis
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available

Example

```
CMD 'INCLUDE-CMD CMD=(DECLARE-VARIABLE A;
                       CALL-PROCEDURE MYPROC,(RET=A);
                       IF(A = ''OKAY'');
                       WRITE-TEXT ''SUCCESSFULL''
                       ELSE
                       WRITE-TEXT ''ERROR''
                       END-IF)'
"RETURN TO PROGRAM MODE"
```

INCLUDE-PROCEDURE
Start command sequence as include procedure

Domain: **PROCEDURE**

Command description

An include procedure is a procedure which is visible in the data environment of the calling procedure. This means that all variables visible in the calling procedure are visible in the include procedure. However, a variable in the calling procedure can be overlaid by a new declaration in the include procedure (see example).

The INCLUDE-PROCEDURE command starts a stored command sequence (procedure). During processing, symbolic parameters contained in the sequence are replaced by the values specified in the command call (PROCEDURE-PARAMETERS operand).

– Current parameters may be transferred as variables; these are also used by the procedure to return output values.
– Current parameters may be transferred as positional parameters or as keyword parameters. The sequence of positional parameters corresponds to the dynamic sequence of the DECLARE-PARAMETER commands; the names of keywords correspond to the names of formal procedure parameters. Keywords may be abbreviated as long as they remain unequivocal.
– Logging is set in the command call; the same applies to the specification regardless of whether or not an already loaded program may be unloaded.

Procedures can be stored as:

– a cataloged SAM or ISAM file (even a temporary one) with records of variable length
– a type J or SYSJ element in a PLAM library
– an S variable of the “list” type

Procedure formats:

– text procedure
  The S procedure is in its original text format. The full SDF-P functionality is available only if the chargeable SDF-P subsystem is loaded when the procedure is called. In libraries, element type J should be used for text procedures.
– object procedure
  An S procedure in text format has been translated to object format with the COMPILE-PROCEDURE command. An object procedure can utilize the full functionality of SDF-P (apart from the COMPILE-PROCEDURE command) regardless of whether or not the SDF-P subsystem is currently available. In libraries, element type SYSJ (the default for COMPILE-PROCEDURE) should be used for object procedures.

Format

```
INCLUDE-PROCEDURE                                                              Alias: INP

FROM-FILE = <filename 1..54 without-gen> / *LIBRARY-ELEMENT(...) / *VARIABLE(...)
  *LIBRARY-ELEMENT(...)
  |   LIBRARY = <filename 1..54 without-gen>
  |  ,ELEMENT = <composed-name 1..64>(...)
  |      <composed-name 1..64>(...)
  |      |   VERSION = *HIGHEST-EXISTING / <composed-name 1..24>
  |  ,TYPE = *STD / *BY-LATEST-MODIFICATION / <alphanum-name 1..8>
  *VARIABLE(...)
  |   VARIABLE-NAME = <composed-name 1..255>
,PROCEDURE-PARAMETERS = *NO / <text 0..1800 with-low expr>
,LOGGING = *PARAMETERS(...) / YES / *NO /
  *PARAMETERS(...)
  |   CMD = *BY-PROC-TEST-OPTION / *YES / *NO
  |  ,DATA = *BY-PROC-TEST-OPTION / *YES / *NO
,UNLOAD-ALLOWED = *YES / *NO
,EXECUTION = *YES / *NO
```

Operands

FROM-FILE = <filename 1..54 without-gen> / *LIBRARY-ELEMENT(...) / *VARIABLE(...)
Name of the procedure file.

FROM-FILE = *LIBRARY-ELEMENT(...)
The procedure is stored in a PLAM library.

LIBRARY = <filename 1..54 without-gen>
Name of the PLAM library containing the procedure as an element (type J or SYSJ; see the TYPE operand).

ELEMENT = <composed-name 1..64>(...)
Name of the element.

VERSION = *HIGHEST-EXISTING / <composed-name 1..24>
Version of the library element. The default value is HIGHEST-EXISTING, i.e. the procedure is taken from the element with the highest version.

TYPE = *STD / *BY-LATEST-MODIFICATION / <alphanum-name 1..8>
Designates the element type the procedure file is stored under in the PLAM library.

TYPE = *STD
The procedure file can be stored as an element of type SYSJ or J.
The specified element is first searched for among the type SYSJ elements.
If it is not found there, the search proceeds to the type J elements.

A non-S procedure can only be a type J element.
An S procedure may be either a text procedure (original text format) or an object procedure (compiled object format). To simplify maintenance of the two formats in a library, text procedures should be stored as type J elements, object procedures as type SYSJ elements. The COMPILE-PROCEDURE command by default generates an object procedure of type SYSJ (default) from a text procedure of type J.
If this convention is followed, specifying TYPE=*STD (the default value) ensures that object procedures will be given precedence over text procedures.

TYPE = *BY-LATEST-MODIFICATION
The procedure file can be stored as an element of type SYSJ or J.
If the specified element exists both as type SYSJ and as type J, the element most recently modified will be called. If the time stamp is identical, the type SYSJ element will be called.
Specifying TYPE=*BY-LATEST-MODIFICATION ensures that the most up-to-date element will be called, typically during the debugging phase when a procedure is being written or modified.

TYPE = <alphanum-name 1..8>
The procedure file will be searched among elements of the specified type only.

FROM-FILE = *VARIABLE(...)
The procedure is stored in an S variable of the “list” type.

VARIABLE-NAME = <composed-name 1..255>
Name of the S variable.

PROCEDURE-PARAMETERS = *NO / <text 0..1800 with-low *expr*>
Defines the current procedure parameters; the parameters must be enclosed in parentheses.
See section “Passing procedure parameters” on page 106 for more details about procedure parameters.

LOGGING = <u>*PARAMETERS</u>(...) / *YES / *NO
This controls logging of procedure execution.
The LOGGING operand is ignored when *non-S procedures* are called, since in this case logging can only be declared in the procedure head (see the LOGGING operand in the BEGIN-PROCEDURE command).
When an S procedure is logged, every procedure line that is processed is output with the line number and procedure level prefixed to it.

See section “Setting the logging” on page 84 for more details about logging.

LOGGING = <u>*PARAMETERS</u>(...)
Logging can be set separately for command/statement lines and for data lines.

CMD = <u>*BY-PROC-TEST-OPTION</u> / *YES / *NO
This specifies whether commands are to be logged. The default value is BY-PROC-TEST-OPTION, i.e. no logging (equivalent to *NO) or the value selected as the default by the user with the MODIFY-PROC-TEST-OPTIONS command.

DATA = <u>*BY-PROC-TEST-OPTION</u> / *YES / *NO
This specifies whether data lines are to be logged. The default value is BY-PROC-TEST-OPTION, i.e. no logging (equivalent to *NO) or the value selected as the default by the user with the MODIFY-PROC-TEST-OPTIONS command.

UNLOAD-ALLOWED = <u>*YES</u> / *NO
This specifies whether a program that was loaded when the procedure was called may be unloaded.
Protection against unloading is guaranteed *only* for unloading by means of the commands START-EXECUTABLE-PROGRAM, LOAD-EXECUTABLE-PROGRAM and CANCEL-PROGRAM.
The specification YES is ignored if the procedure is called from a procedure for which UNLOAD-ALLOWED=*NO was declared.

EXECUTION = <u>*YES</u> / *NO
This specifies whether the procedure is merely to be analyzed for test purposes or whether it is also to be executed.
Testing is possible via the MODE operand of the MODIFY-SDF-OPTIONS command.

Command return codes

The following command return codes can only be returned if the called procedure does not supply any command return code itself (e.g. EXIT-PROCEDURE not executed due to an error).

Command return codes whose maincode begins with “SSM” can only be returned when a non-S procedure is called.

Command return codes whose maincode begins with “SDP” can only be returned when an S procedure is called.

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
2	0	SSM2058	Protocol type error
2	0	SSM2065	EOF on procedure file, /END-PROC simulated
	1	SSM2036	Incomplete operand
	1	SSM2054	Symbolic operand error
	1	SSM2055	Symbolic operand error in /BEGIN-PROC
	1	SDP0138	Error in pre-analysis of text procedure, or object procedure invalid
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0093	Non-S procedure can only be type J element
	64	SDP0144	Error on parameter transfer
	64	SSM2052	DMS error (Open error)
	64	SSM2053	Not a SAM/ISAM file or file does not begin with /BEGIN-PROC or /PROC
	64	SSM2056	/CALL-PROC and /BEGIN-PROC parameters incompatible
	64	SSM2061	Error on accessing library element
	64	SSM2064	Procedure file cannot be fetched by remote processor
	130	SDP0099	No further address space available
xx	xx	xxxxxxx	Other return codes from the called procedure

Example

Procedure 1:

```
/DECLARE-VARIABLE A(TYPE = *STRING)
/DECLARE-VARIABLE B(TYPE = *STRING)
/DECLARE-VARIABLE C(TYPE = *STRING)
/DECLARE-VARIABLE D
/INCLUDE-PROCEDURE PROC2
```

Procedure PROC2:

```
/DECLARE-VARIABLE NAME=A(INIT-VALUE=1,TYPE=*INTEGER),SCOPE =*CURRENT ——  (1)
/DECLARE-VARIABLE NAME=C(TYPE=*STRING),SCOPE = *PROCEDURE  ————————————  (2)
/DECLARE-VARIABLE NAME=B(TYPE=*INTEGER),SCOPE = *PROCEDURE ————————————  (3)
```

(1) A variable A, which is valid only in PROC2 is declared in PROC2. Variable A in the calling procedure (Procedure 1) is therefore not visible in PROC2, but only after PROC2 is terminated.

(2) This is a permissible multiple declaration. It is ignored. The procedure variables C and D in Procedure 1 are visible in PROC2. The procedure variable A in Procedure 1 is covered by variable A in PROC2.

(3) ERROR: The third declaration produces an error, because a procedure variable B having another data type already exists. In a multiple declaration, all attributes must match the original declaration.

MODIFY-PROCEDURE-OPTIONS
Modify procedure attributes during procedure execution

Domain: **PROCEDURE**

Command description

MODIFY-PROCEDURE-OPTIONS can be used to modify, during procedure execution, most of the procedure attributes set with SET-PROCEDURE-OPTIONS at the beginning of the procedure execution.

MODIFY-PROCEDURE-OPTIONS cannot be called if the procedure execution has been interrupted.

If MODIFY-PROCEDURE-OPTIONS is called within an include procedure, it affects this include procedure only, i.e. changes are not transferred to the calling procedure.

Format

```
MODIFY-PROCEDURE-OPTIONS

 IMPLICIT-DECLARATION = *UNCHANGED / *YES / *NO
,LOGGING-ALLOWED = *PARAMETERS(...) / *NO  / *YES
   *PARAMETERS(...)
     |  CMD = *UNCHANGED / *YES / *NO
     | ,DATA = *UNCHANGED / *YES / *NO
,INTERRUPT-ALLOWED = *UNCHANGED / *YES / *NO
,DATA-ESCAPE-CHAR = *UNCHANGED / *NONE / '&&' / '#' / '*' / '@' / '$' / *STD
,DATA-ERROR-HANDLING = *UNCHANGED / *YES / *NO
,JV-REPLACEMENT = *UNCHANGED / *NONE / *AFTER-BUILTIN-FUNCTION
,ERROR-MECHANISM = *UNCHANGED / *SPIN-OFF-COMPATIBLE / *BY-RETURNCODE
,SUPPRESS-SDP-MSG = *UNCHANGED / *NONE / *ADD(...) / *REMOVE(...)
   *ADD(...)
     |  MSG-ID = list-poss(2000): <alphanum-name 7..7>)
   *REMOVE(...)
     |  MSG-ID = list-poss(2000): <alphanum-name 7..7>)
```

Operands

IMPLICIT-DECLARATION = *UNCHANGED / *YES / *NO
Indicates whether implicit declarations are permitted.
If UNCHANGED is specified, the previous declaration is accepted unchanged.
IMPLICIT-DECLARATION can be specified in dialog.

LOGGING-ALLOWED =
Defines whether logging is permitted in the procedure.

LOGGING-ALLOWED = *PARAMETERS(...)
In the following entries, defines what can be logged.

CMD = *UNCHANGED / *YES / *NO
Specifies whether commands can be logged.
If *UNCHANGED is specified, the previous declaration is accepted unchanged.

DATA = *UNCHANGED / *YES / *NO
Specifies whether data can be logged.
If *UNCHANGED is specified, the previous declaration is accepted unchanged.

LOGGING-ALLOWED = *YES
Commands and data can be logged.

LOGGING-ALLOWED = *NO
Logging is not allowed.

INTERRUPT-ALLOWED = *UNCHANGED
The previous declaration is to be accepted unchanged.

INTERRUPT-ALLOWED = *YES
Specifies that the procedure can be interrupted with function key K2 and resumed with the RESUME-PROCEDURE command.

INTERRUPT-ALLOWED = *NO
Specifies that the procedure cannot be interrupted with function key K2.

DATA-ESCAPE-CHAR =
Defines the character to be used as the escape character. The escape character is the character which identifies symbolic parameters in data records.

DATA-ESCAPE-CHAR = *UNCHANGED
The previous declaration is to be accepted unchanged.

DATA-ESCAPE-CHAR = *NONE
Expression replacement is not to be carried out in data records.

DATA-ESCAPE-CHAR = ’&&’ / ’#’ / ’*’ / ’@’ / ’$’
Defines the escape character.

DATA-ESCAPE-CHAR = *STD
The character ’&’ is to be used as the escape character.

DATA-ERROR-HANDLING = <u>*UNCHANGED</u>
The previous declaration is to be accepted unchanged.

DATA-ERROR-HANDLING = *YES
Specifies that error handling is initiated in the following instances:

– if a procedure line contains data where commands are expected
– if a requested expression replacement operation cannot be carried out in data lines
– if a procedure record contains a single escape character

DATA-ERROR-HANDLING = *NO
Error handling is not initiated; &varname remains unchanged in data if varname is not known as either a function or a variable.

JV-REPLACEMENT = <u>*UNCHANGED</u> / *NONE / *AFTER-BUILTIN-FUNCTION
Specifies whether job variable replacement is to be carried out during expression replacement.

JV-REPLACEMENT = <u>*UNCHANGED</u>
The existing setting is to be used unchanged.

JV-REPLACEMENT = *NONE
During expression replacement, names are not interpreted as job variable names.

JV-REPLACEMENT = *AFTER-BUILTIN-FUNCTION
In an expression in the form &(name), name is interpreted as a job variable name if there is no variable or built-in function with this name. This operand value is provided to permit behavior compatible with non-S procedures during expression replacement.

ERROR-MECHANISM =
Specifies whether error handling is to be initiated in a manner compatible with the spin-off behavior of non-S procedures or whether “subcode1 not equal to zero” is to be taken into account. The operand setting has no influence on error handling for statements.

ERROR-MECHANISM = <u>*UNCHANGED</u>
The existing setting is to be used unchanged.

ERROR-MECHANISM = *SPIN-OFF-COMPATIBLE
Error handling is to be initiated in a manner compatible with the previous spin-off behavior. Subcode1 is **not** taken into account. This ensures that the behavior of S procedures created under BS2000 V10.0 remains compatible.

ERROR-MECHANISM = *BY-RETURNCODE
Error handling is initiated if subcode1 of the last command return code is not equal to zero. The spin-off behavior is not taken into account. If *BY-RETURNCODE is specified, the error handling in S procedures must be matched to the command return codes of the various commands.

Notes

– In order to avoid problems which could result from the modification of the default value in the user syntax file, the selected value should be specified explicitly in the procedure.
– The IMPLICIT-DECLARATION and JV-REPLACEMENT operands can also be specified in dialog.
  At the beginning of the task, the following settings are valid in interactive mode:
  IMPLICIT-DECLARATION = *YES
  JV-REPLACEMENT = *AFTER-BUILTIN-FUNCTION

SUPPRESS-SDP-MSG =
Specifies whether output is to be suppressed for specific SDF-P messages (message class SDP). The option is valid for the calling procedure only (i.e. it is not “inherited” by other procedures).

SUPPRESS-SDP-MSG = <u>*UNCHANGED</u>
The existing setting is to be used unchanged (/SET-PROCEDURE-OPTIONS or previous /MODIFY-PROCEDURE-OPTIONS command).

SUPPRESS-SDP-MSG = *NONE
All SDF-P messages are to be output.

SUPPRESS-SDP-MSG = *ADD(...)
Set of SDF-P messages that are not to be output (possibly in addition to any previously specified messages).

> **MSG-ID=list-poss(2000): <alphanum-name 7..7>**
> List of message numbers (message class SDP).

SUPPRESS-SDP-MSG = *REMOVE(...)
Set of (currently suppressed) SDF-P messages that are to be output again.

> **MSG-ID=list-poss(2000): <alphanum-name 7..7>**
> List of message numbers (message class SDP).

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	130	SDP0099	No further address space available

Example

```
/SET-PROCEDURE-OPTIONS, LOGGING-ALLOWED=*NO
...
/MODIFY-PROCEDURE-OPTIONS, LOGGING-ALLOWED=*YES
...
```

At the start of the procedure, the specification in SET-PROCEDURE-OPTIONS forbids logging. After the MODIFY-PROCEDURE-OPTIONS command, commands and data may be logged.

MODIFY-PROCEDURE-TEST-OPTIONS
Modify logging and limit number of back branches

Domain: **PROCEDURE**

Command description

The following settings for executing the procedure can be changed for test purposes with the MODIFY-PROCEDURE-TEST-OPTIONS command:

– Logging of commands and data. Logging can only be enabled when this is allowed for the procedure.
– Limiting back branches prevents continuous loops and similar phenomena.
– Simulation of SDF-P-BASYS response if full SDF-P functionality is available. These settings apply to the task level.

The following settings are valid at the beginning of the task:

– CMD = *NO, DATA = *NO
– BACK-BRANCH-LIMIT = *NONE
– FUNCTIONALITY = *FULL

MODIFY-PROCEDURE-TEST-OPTIONS is ignored in non-S procedures; logging is controlled with the BEGIN-PROCEDURE command in these procedures.

For details of the logging function, refer to section “Requesting logging” on page 111.

Format

```
MODIFY-PROCEDURE-TEST-OPTIONS

LOGGING = *PARAMETERS(...) / *YES / *NO /

  *PARAMETERS(...)

     |  CMD = *UNCHANGED / *YES / *NO
     |
     | ,DATA = *UNCHANGED / *YES / *NO

,BACK-BRANCH-LIMIT = *UNCHANGED / *NONE / <text 0..1800 with-low arith-expr>

,FUNCTIONALITY = *UNCHANGED / *FULL / *BASIC
```

Operands

LOGGING = *PARAMETERS(...)
Controls logging. This operand has no effect on procedures which are protected by a read password (see section "Setting the logging" on page 84).

CMD = *UNCHANGED / *YES / *NO
Specifies whether commands are logged.
If *UNCHANGED is specified, the current declaration is accepted unchanged.
The specification *YES is effective only if command logging is allowed in the first place.

DATA = *UNCHANGED / *YES / *NO
Specifies whether data is logged.
If *UNCHANGED is specified, the current declaration is accepted unchanged. The specification *YES is effective only if data logging is allowed in the first place.

LOGGING = *YES
Enables the declared logging function.
The specification *YES is effective only if command logging is allowed in the first place.

LOGGING = *NO
Disables logging.

Note

The entries LOGGING = *YES and LOGGING = *PARAMETERS(CMD=*YES, DATA=*YES) are equivalent, as are LOGGING = *NO and LOGGING = *PARAMETERS(CMD=*NO,DATA=*NO).

BACK-BRANCH-LIMIT =
Determines the maximum number of back branches permitted in the procedure. This does not include back branches initiated with the SKIP-COMMANDS command.

BACK-BRANCH-LIMIT = *UNCHANGED
The current declaration is used unchanged.

BACK-BRANCH-LIMIT = *NONE
The number of back branches in unlimited.

BACK-BRANCH-LIMIT = <text 0..1800 with-low *arith-expr*>
Integer expression indicating the maximum number of permissible back branches.
Error handling is activated when this limit is reached.

FUNCTIONALITY =
Determines which SDF-P functionality is to be activated or simulated. The setting is valid in the task level, i.e. it exists until the task is terminated or until the next modification.

FUNCTIONALITY = *UNCHANGED / *FULL / *BASIC
The previous definition is still valid.

FUNCTIONALITY = *FULL
If the SDF-P subsystem (purchased separately) is loaded in the system, then the full SDF-P functionality is activated.

FUNCTIONALITY = *BASIC
SDF-P-BASYS functionality will be simulated. If the SDF-P product (purchased separately) is available in the system, the MODIFY-PROCEDURE-TEST-OPTIONS command will also be executed with this setting.

Command return code

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error
	64	SDP0133	Invalid data type in expression
	130	SDP0099	No further address space available

Example

See the function LOGGING-MODE( ), page 444.

OPEN-VARIABLE-CONTAINER
Open variable container

Domain: **PROCEDURE**

Command description

The OPEN-VARIABLE-CONTAINER command is used to open variable containers, which are stored as PLAM library elements. If such a variable container or element does not yet exist when the command is called, it is automatically created.

This makes it possible to create S variables which are permanently available, i.e. S variables whose existence is not dependent on the current task.

Format

```
OPEN-VARIABLE-CONTAINER

 CONTAINER-NAME = <composed-name 1..64>
,FROM-FILE = *LIBRARY-ELEMENT (...)
    *LIBRARY-ELEMENT(...)
       |  LIBRARY = <filename 1..54 without-vers>
       | ,ELEMENT = *CONTAINER-NAME / <composed-name 1..64>(...)
       |     <composed-name 1..64>(...)
       |        |  VERSION = *HIGHEST-EXISTING / <composed-name 1..24>
,LOCK-ELEMENT = *NO / *YES
,SCOPE = *CURRENT / *PROCEDURE / *TASK(...)
    *TASK(...)
       |  SAVE-AT-TERMINATION = *NO / *YES
,AUTOMATIC-DECLARE = *ALL / *NONE / <structured-name 1..20 with-wild(40)> /
                        list-poss(2000): <structured-name 1..20>
```

Operands

CONTAINER-NAME = <composed-name 1..64>
Name of the variable container.

FROM-FILE = *LIBRARY-ELEMENT(...)
The library element which contains the variable container.
The data type of this element is SYSVCONT.

LIBRARY = <filename 1..54 without-vers>
Name of the PLAM library. Specification of a list of libraries
(S variable SYSPLAMALT-<name>) is permissible.

ELEMENT =
Name of the element.

ELEMENT = <u>*CONTAINER-NAME</u>
The name of the element is identical with that of the variable container.

ELEMENT = <composed-name 1..64>(...)
The name of the element may differ from that of the variable container.

VERSION =
Designates the version number of the element.

VERSION = <u>*HIGHEST-EXISTING</u>
Selects the highest existing version number.

VERSION = <composed-name 1..24>
Selects the specified version number.

LOCK-ELEMENT =
Specifies whether or not the element is locked.

LOCK-ELEMENT = <u>*NO</u>
The element is opened in input mode. The container variables are copied into this element. The element is then locked.

LOCK-ELEMENT = *YES
The element is opened in input and output mode. The container variables are copied from this element into the variable container. The element then remains open until the CLOSE-VARIABLE-CONTAINER command is issued. Any subsequent OPEN-VARIABLE-CONTAINER which is issued in the same task or another task is rejected.

SCOPE =
Defines the scope of the variable container. This controls access to the variables held in the variable container.
The scope of the container variable must not be greater than that of the variable container.

SCOPE = <u>*CURRENT</u>
The scope of the variable container is procedure-local (for further details see section "Scope of variables" on page 157).
The variable container can only be used in the local procedure and in any lower-level include procedures, but not in the calling procedure. The container is implicitly closed at the end of the current procedure.

SCOPE = *PROCEDURE
The scope of the variable container is procedure-local (for further details see section "Scope of variables" on page 157).
The variable container can be used in the local procedure and in any lower-level include procedures. It can also be used in the calling procedures if these were called by INCLUDE-PROCEDURE. It is implicitly closed at the end of the first-called procedure. I.e. it is open across all Include procedures until termination of the highest-level calling procedure.

SCOPE = *TASK(...)
The scope of the variable container is task-global (for further details see section "Scope of variables" on page 157).
The variable container can be used until it is closed or the task is terminated. Unlike the scope of variables, it is not necessary to import the name of the container before it is used.

SAVE-AT-TERMINATION =
Specifies whether the variable container must be saved at EXIT-JOB or LOGOFF.

SAVE-AT-TERMINATION = <u>*NO</u>
The variable container is not saved at EXIT-JOB.

SAVE-AT-TERMINATION = *YES
The variable container is saved at EXIT-JOB. However, it is not saved at an abnormal task termination, as for example with the setting EXIT-JOB MODE = ABNORMAL.

AUTOMATIC-DECLARE =
Specifies whether the container variables are to be automatically declared.

AUTOMATIC-DECLARE = <u>*ALL</u>
The container variables are automatically declared, with the scope of the variable container.

AUTOMATIC-DECLARE = *NONE
Container variables are not automatically declared.

AUTOMATIC-DECLARE = list-poss(2000): <structured-name 1..20>
The specified container variables are automatically declared with the scope of the variable container.

AUTOMATIC-DECLARE = <structured-name 1..20 with-wild(40)>
The container variables which match the specified search pattern are automatically declared with the scope of the variable container.

Notes

- Variables in a variable container can be created by means of the CONTAINER operand in the DECLARE-VARIABLE command.
- Variables which are declared as static structure layouts are saved with the name of the corresponding structure layout.
- A reference to a variable container is not allowed until it has been created using OPEN-VARIABLE-CONTAINER.
- If variables are automatically created, using OPEN-VARIABLE-CONTAINER, and the variables already exist with different attributes, the declaration is rejected and error message SDP1018 is returned as a warning. Notwithstanding this, the opening process continues.
  The user can interrogate the rejected variables by means of the S variable stream SYSMSG.
- If container variables are created by AUTOMATIC-DECLARE, and if they relate to static structures, they are converted to structures of type '*BY-SYSCMD'.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
2	0	SDP00xx	Warning that the following has occurred: guaranteed messages: SDP1008, SDP1018
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	CMD0216	Do not have required privilege
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available

Example

See the SHOW-VARIABLE-CONTAINER-ATTR command, page 776.

RAISE-ERROR
Generate return code

Domain: **PROCEDURE**

Command description

RAISE-ERROR generates a command return code and subsequently activates error handling if SUBCODE 1 is a number other than zero.

Format

RAISE-ERROR
SUBCODE1 = **64** / <integer 0..255> ,**SUBCODE2** = **0** / <integer 0..255> ,**MAIN**CODE = **SDP0018** / <alphanum-name 7..7>

Operands

SUBCODE1 = 64 / <integer 0..255>
Number indicating the error class;
64 = error class “SEMANTIC-ERROR”.

SUBCODE2 = 0 / <integer 0..255>
Additional information on the error class.

MAINCODE = SDP0018/ <alphanum-name 7..7>
Error code for determining various error causes.
The preset value for MAINCODE is SDP0018.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	130	SDP0099	No further address space available
xx	xx	xxxxxxx	Return code as specified in operands

READ-VARIABLE
Assign values to variables

Domain: **PROCEDURE**

Command description

The READ-VARIABLE command is used to read data from an input medium and to store the data in a variable. The data is read record by record. The operation is terminated when the string *END-OF-CMD or the end of a file (EOF) is encountered.

The input medium may be any of the following:

– the terminal
– a cataloged file
– a variable
– a library element
– the SYSDTA system file

Notes concerning reading data from SYSDTA

– Data is read from SYSDTA exactly as it is from a file. However, there is one exception: If the read operation is terminated before SYSDTA EOF, the next SYSDTA record is accessed the next time that data is read in from SYSDTA. This allows reading of specific data from SYSDTA. This is useful, for example, if only a single simple variable is to be read.
– Input ends at the end of the SYSDTA system file, i.e. either at the end of the file if a cataloged file is assigned to SYSDTA, or with the next command if SYSDTA is assigned to SYSCMD (default value for S procedures).
– It is not possible to interrupt the input of data read from SYSDTA by means of HOLD-PROGRAM, K2 or BEGIN-BLOCK PROGRAM-INPUT=*MIXED-WITH-CMD (in order to switch to command mode). These commands and actions only result in the termination of input, i.e. SYSDTA EOF (end-of-file) being reported to the appropriate command. Only the SEND-DATA command does not terminate the READ-VARIABLE command.

Note on reading in a file

When the complete contents of a file are not required for further processing, the volume of data can be restricted while the file is being read in in the following cases:

1. Only a contiguous section of the records is required. The numbers of the first and last records required are specified in the BEGIN-RECORD and END-RECORD operands for the range of records to be read in.

2. Only records which contain specific strings are required. The records to be read in are selected by means of the search string specified in the PATTERN operand. The PATTERN-TYPE operand determines whether the search string is to be evaluated as a string or as a regular expression in accordance with POSIX rules.
   A simple search string can be entered directly as a string. A regular expression enables a complex search string to be specified which cannot be specified directly as a string (see “POSIX wildcards” on page 554; “regular expressions” are described in the “POSIX Commands” manual [18]).

   Note

   A search string is always searched for in the entire record, even when only part (range of columns) of the record is to be read in (see Example 5, Case 3 on page 714).

3. Only part of each record is needed. The part of the record to be read in is specified as a range of columns with the columns of the first and last characters required being specified in the BEGIN-COLUMN and END-COLUMN operands.

Format

```
READ-VARIABLE                                                              Alias: RDV

VARIABLE-NAME = *BY-INPUT(...) / *LIST(...) / list-poss(2000): <composed-name 1..255>
   *BY-INPUT(...)
     |  PREFIX = *NONE / <composed-name 1..255>
   *LIST(...)
     |  LIST-NAME = <composed-name 1..255>
     | ,WRITE-MODE = *REPLACE / *EXTEND
,STRING-QUOTES = *OPTIONAL / *YES / *NO
,INPUT = *TERMINAL(...) / <filename 1..54 without-gen-vers>(...) / *VARIABLE(...) /
          *LIBRARY-ELEMENT(...) / *SYSDTA(...)
   *TERMINAL(...)
     |  PROMPT-STRING = '>>' / <text 0..1800 with-low>
     | ,SECRET-INPUT = *NO / *YES
   <filename 1..54 without-gen-vers>(...)
     |  REMOVE-KEY = *YES / *NO
     | ,BEGIN-RECORD = *FIRST / <integer 1..2147483647>
     | ,END-RECORD = *LAST / <integer 1..2147483647>
     | ,BEGIN-COLUMN = *FIRST / <integer 1..2147483647>
     | ,END-COLUMN = *LAST / <integer 1..2147483647>
     | ,PATTERN = *NONE / <c-string 0..1800 with-low>
     | ,PATTERN-TYPE = *STRING / *REGULAR-EXPRESSION
   *VARIABLE(...)
     |  VARIABLE-NAME = <composed-name 1..255>
   *LIBRARY-ELEMENT(...)
     |  LIBRARY = <filename 1..54 without-vers>
     | ,ELEMENT = <composed-name 1..64>(...)
     |     <composed-name 1..64>(...)
     |       |  VERSION = *HIGHEST-EXISTING / <composed-name 1..24>
     | ,TYPE = S / <alphanum-name 1..8>
   *SYSDTA(...)
     |  REMOVE-KEY = *YES / *NO
```

Operands

VARIABLE-NAME =
Designates the variables to which values are to be assigned.

VARIABLE-NAME = *BY-INPUT(...)
Specifies that the variables are in the format generated by the SHOW-VARIABLE command via the PREFIX and FORM operands.
If the variables are not generated by the SHOW-VARIABLE command, they must exist in the SHOW-VARIABLE output format (e.g. there must be a blank before and after the '=').
If the variables already exist, they are assigned the value.

If a variable does not yet exist, it is implicitly declared, as long as implicit declaration is allowed.
If the superordinate complex variable for a variable element does not exist, it is declared with SCOPE = *CURRENT.
Static structures can be regenerated only as dynamic structures. STRING, INTEGER and BOOLEAN are mapped to ANY if the variables do not exist.

If *BY-INPUT(...) refers to a list element (variable name with (*LIST)), the command is terminated with errors.
If a variable is assigned the value *NO-INIT, its previous contents are deleted (equivalent to a FREE-VARIABLE command call).
The value *END-OF-VAR is ignored. The value *END-OF-CMD terminates the assignment.

PREFIX = <u>*NONE</u>
The variable name is not preceded by a prefix.

PREFIX = <composed-name 1..255>
Designates the name to precede the variable name as a prefix.

VARIABLE-NAME = *LIST(...)
Designates a list variable.

LIST-NAME = <composed-name 1..255>
Name of the list.
If the list already exists, it must have the type ANY, STRING, BOOLEAN or INTEGER.
If the list does not exist, or if it cannot be located, it is re-declared with the scope SCOPE = *CURRENT and the data type TYPE = *STRING.

WRITE-MODE=<u>*REPLACE</u>
The list is overwritten.

WRITE-MODE=*EXTEND
New elements are appended to the end of the list.

VARIABLE-NAME = list-poss(2000): <composed-name 1..255>
Names of the variables to which new values are to be assigned.

The values which are read in are assigned in turn to the variables (variable elements) - starting with the first element in the structure.
Error handling is initiated if not all of the variables are assigned values, due to a premature EOF or *END-OF-CMD.

STRING-QUOTES =
Defines how the input value is to be interpreted.

STRING-QUOTES = <u>*OPTIONAL</u>
Makes it possible to specify variable values like procedure parameters.
The data type of the variable to which the value is to be assigned is taken into account in interpreting the entry:

– If the variable has the data type ANY or STRING, the value is interpreted as a string. Single quotes within the string must be doubled (this, however, applies only if the entire string is enclosed in quotes).
– If the variable has the data type INTEGER, the value is interpreted as an integer.
– If the variable has the data type BOOLEAN, the value is interpreted as a Boolean value.

STRING-QUOTES = *YES
Interprets values beginning with a single quote (’) or with C and single quote (C’) as a string. If the string does not end with a single quote, or if single quotes are not doubled within the value, this is interpreted as an error.
If a value which is not enclosed in quotes corresponds to one of the keywords for Boolean values, it is interpreted as a Boolean value. In all other instances, it is interpreted as an integer.

STRING-QUOTES = *NO
Interprets all values as a string.

INPUT =
Defines the location from which the values are to be input.

INPUT = <u>*TERMINAL</u>(...)
Prompting: the values are input from the data station. (However, unlike DECLARE-PARAMETER INITIAL-VALUE =*PROMPT, this does not convert the values to uppercase.)

PROMPT-STRING = <u>’>>’</u> / <text 0..1800 with-low>
String expression. Defines the character or character string which is to be used as prompt.

SECRET-INPUT = <u>*NO</u> / *YES
Defines whether or not the user’s input is to be concealed.

INPUT = <filename 1..54 without-gen-vers>(...)
The values are to be input from the specified file.
Each input record is interpreted as a single value.

REMOVE-KEY =
Specifies whether the key is to be omitted when reading from an ISAM file.

REMOVE-KEY = *YES
In the case of an ISAM file, the key is not read in. In the case of a SAM file, only this specification is accepted.

REMOVE-KEY = *NO
In the case of ISAM files with the standard attributes KEY-POSITION=5 and KEY-LENGTH=8, the key is also read in. In the case of ISAM files with other attributes, this specification leads to an error. This specification always leads to an error with SAM files.

BEGIN-RECORD = *FIRST / <integer 1..2147483647>
Specifies the record which is to be the first to be read in. *FIRST specifies the first record of the file as the default.
If the value specified is greater than the number of existing records, an empty file is assumed. In the case of VARIABLE-NAME=*LIST(...), an empty list is created.

END-RECORD = *LAST / <integer 1..2147483647>
Specifies the record after which transfer is to be terminated. *LAST specifies the last record of the file as the default.
If the value specified is greater than the number of existing records, *LAST is assumed.

BEGIN-COLUMN = *FIRST / <integer 1..2147483647>
Specifies the column position from which the content of a record is to be read in. *FIRST specifies the start of the record (column position 1) as the default.
If the value specified is greater than the record, no data is transferred (no list elements are created).

END-COLUMN = *LAST / <integer 1..2147483647>
Specifies the column position after which the transfer of a record is to be terminated. *LAST specifies the end of the record as the default.
If the value specified is greater than the record, *LAST is assumed.

PATTERN =
Specifies whether only records which contain a particular search string are to be read in.

PATTERN = *NONE
Records are read in regardless of any particular search string.

PATTERN = <c-string 0..1800 with-low>
Only records which contain the search string specified are read in. The PATTERN-TYPE operand determines how the search string is to be evaluated.

PATTERN-TYPE =
Specifies how the search string is to be evaluated.

PATTERN-TYPE = *STRING
Records are selected which contain the specified string.

PATTERN-TYPE = *REGULAR-EXPRESSION
Records are selected which contain the string specified as a regular expression.

INPUT = *VARIABLE(...)
The values are to be input from a list variable.

VARIABLE-NAME = <composed-name 1..255>
Name of the list variable.
This variable must exist, and must have been declared with the data type STRING. (It can also be declared with the data type ANY; however, it may still contain strings only.)

INPUT = *LIBRARY-ELEMENT(...)
The values are to be read in from an element in a PLAM library. Specifying a list of libraries (S variable SYSPLAMAT-<name>) is permissible.

LIBRARY = <filename 1..54 without-vers>
Name of the library.

ELEMENT = <composed-name 1..64>(...)
Name of the library element.

VERSION = *HIGHEST-EXISTING / <composed-name 1..24>
Version of the library element.

TYPE = S / <alphanum-name 1..8>
Type of the library element.

INPUT = *SYSDTA(...)
The values are to be read in from SYSDTA.

REMOVE-KEY =
If SYSDTA is assigned to a file:
Specifies whether the key is to be omitted when reading from an ISAM file.

REMOVE-KEY = *YES
In the case of an ISAM file, the key is omitted. In the case of a SAM file, only this specification is accepted.

REMOVE-KEY = *NO
Only in the case of an ISAM file with the standard attributes KEY-POSITION=5 and KEY-LENGTH=8 is the key also read in. This specification leads to an error tn the case of an ISAM file with other attributes and in the case of a SAM file.

Command return codes

It is possible that part of the command has already been processed and executed before the error occurs. In this case, the result of the command is not guaranteed.

(SC2)	SC1	Maincode	Meaning/Guaranteed messages
	0	CMD0001	No error
2	0	SDP2000	Warning: not all elements of the input list could be processed successfully. Guaranteed message: SDP2000
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0089	INPUT error
	64	SDP0091	Semantic error Guaranteed messages: SDP1008
	64	SDP2001	None of the elements could be read in
	130	SDP0099	No further address space available

Example 1

If a certain number of values must be used in a procedure, these values can be listed in SYSCMD; they do not have to be put in a different file.

```
/DECLARE-VARIABLE FILES(TYPE=*STRING),MULTIPLE-ELEMENTS=*LIST
/READ-VARIABLE *LIST(FILES),INPUT=*SYSDTA
FILE1
FILE2
FILE3
/FOR FILE=*LIST(FILES)
/   IF (NOT IS-CATALOGED-FILE(FILE))
/      WRITE-TEXT 'FILE &FILE NOT FOUND.'
/   END-IF
/END-FOR
```

Example 2

A certain number of lines, which remains constant, is to be read from a file. In this example, the number of lines is 10. Each time the loop is executed, 10 lines of the file ’MY-FILE’ are read and a list ’LINES’ is declared with a limit of 10 list elements. In brief: this example shows the contents of the file ’MY-FILE’.

```
/ASSIGN-SYSDTA TO=MY-FILE
/DECLARE-VARIABLE LINES(TYPE=*STRING),MULTIPLE-ELEMENTS=*LIST(LIMIT=10)
/REPEAT
/   READ-VARIABLE *LIST(LINES),INPUT=*SYSDTA
/   SHOW-VARIABLE LINES,INFORMATION=*PARAMETERS(NAME=*NONE)
/UNTIL (SIZE('LINES') LT 10)
```

Example 3

A procedure has the following contents:

```
/DECLARE-VARIABLE FORENAME
/WRITE-TEXT 'Please enter a forename'
/READ-VARIABLE FORENAME
/SHOW-VARIABLE FORENAME
```

Output
```
Please enter a forename
```

Input
```
John
```

Output
```
FORENAME= John
```

Example 4

A file is to be read from SYSDTA line by line. An error is reported if READ-VARIABLE detects EOF for SYSDTA. The example simply shows the contents of the file ’MY-FILE’.

```
/ASSIGN-SYSDTA TO=MY-FILE
/EOF=FALSE
/WHILE (NOT EOF)
/   READ-VARIABLE LINE,INPUT=*SYSDTA
/   IF-CMD-ERROR
/       EOF=TRUE
/   ELSE
/       SHOW-VARIABLE LINE,INFORMATION=*PARAMETERS(NAME=*NONE)
/   END-IF
/END-WHILE
```

Example 5

The DATA-FILE file contains the following records:

```
MUELLER       : 55900 : DE : Gladbecker Strasse 7
GRANDY        : 74663 : UK : Albert Street 12
DANFERTH      : 83092 : DE : Colmberger Strasse 2
GRABATER      : 01927 : FR : rue du Couedic 27
SMITH         : 54920 : UK : Elizabeth Street 54
DUPONT        : 45888 : FR : rue de la Maderie 78
VANDENMORSE   : 94958 : BE : Brusselssteenweg 101
```

1. Read in range of records

```
/DECLARE-VARIABLE FILE (TYPE=*STRING), MULTIPLE-ELEMENTS=*LIST
/READ-VARIABLE *LIST(FILE), INPUT=SDF-P-FILE (BEGIN-RECORD = 4, -
/                                             END-RECORD = 7)
/SHOW-VARIABLE FILE
FILE(*LIST) = GRABATER      : 01927 : FR : rue du Couedic 27
FILE(*LIST) = SMITH         : 54920 : UK : Elizabeth Street 54
FILE(*LIST) = DUPONT        : 45888 : FR : rue de la Maderie 78
FILE(*LIST) = VANDENMORSE   : 94958 : BE : Brusselssteenweg 101
```

Records 4 through 7 were read in to the FILE variable.

2. Read in range of columns from range of records

```
/DECLARE-VARIABLE FILE (TYPE=*STRING), MULTIPLE-ELEMENTS=*LIST
/READ-VARIABLE *LIST(FILE), INPUT=SDF-P-FILE (BEGIN-RECORD=5, -
/                                             END-RECORD=8, -
/                                             BEGIN-COLUMN=17, -
/                                             END-COLUMN=21)
/SHOW-VARIABLE FILE
FILE(*LIST) = 54920
FILE(*LIST) = 45888
FILE(*LIST) = 94958
```

Columns 17 through 21 from records 5 through 7 were read in to the FILE variable (no 8th record exists).

3. Search records for string and read in hits

```
/DECLARE-VARIABLE FILE (TYPE=*STRING), MULTIPLE-ELEMENTS=*LIST
/READ-VARIABLE *LIST(FILE), INPUT=SDF-P-FILE (PATTERN=': DE :', -
/                                             PATTERN-TYPE=*STRING)
/SHOW-VARIABLE FILE
FILE(*LIST) = MUELLER       : 55900 : DE : Gladbecker Strasse 7
FILE(*LIST) = DANFERTH      : 83092 : DE : Colmberger Strasse 2
```

All records which contain the string ": DE :" were read in to the FILE variable.

```
/READ-VARIABLE *LIST(FILE), INPUT=SDF-P-FILE (BEGIN-COLUMN=23, -
/                                             END-COLUMN=28, -
/                                             PATTERN=': DE :', -
/                                             PATTERN-TYPE=*STRING)
/SHOW-VARIABLE FILE
FILE(*LIST) = : DE :
FILE(*LIST) = : DE :
```

Columns 23 through 28 from all records which contain the string “: DE :” were read in to the FILE variable.

```
/READ-VARIABLE *LIST(FILE), INPUT=SDF-P-FILE (BEGIN-COLUMN=1, -
/                                             END-COLUMN=22, -
/                                             PATTERN=': DE :', -
/                                             PATTERN-TYPE=*STRING)
/SHOW-VARIABLE FILE
FILE(*LIST) = MUELLER       : 55900
FILE(*LIST) = DANFERTH      : 83092
```

Columns 1 through 22 from all records which contain the string “: DE :” were read in to the FILE variable.

4. Search records for string or regular expression and read in hits

```
/DECLARE-VARIABLE FILE (TYPE=*STRING), MULTIPLE-ELEMENTS=*LIST
/READ-VARIABLE *LIST(FILE), INPUT=SDF-P-FILE (PATTERN=': DE :', -
/                                             PATTERN-TYPE=*STRING)
/SHOW-VARIABLE FILE
FILE(*LIST) = MUELLER       : 55900 : DE : Gladbecker Strasse 7
FILE(*LIST) = DANFERTH      : 83092 : DE : Colmberger Strasse 2
```

All records which contain the string “: DE :” were read in to the FILE variable.

```
/READ-VARIABLE *LIST(FILE) -
/            , INPUT=SDF-P-FILE (PATTERN=': [DB]E :', -
/                                PATTERN-TYPE=*REGULAR-EXPRESSION)
/SHOW-VARIABLE FILE
FILE(*LIST) = MUELLER       : 55900 : DE : Gladbecker Strasse 7
FILE(*LIST) = DANFERTH      : 83092 : DE : Colmberger Strasse 2
FILE(*LIST) = VANDENMORSE   : 94958 : BE : Brusselssteenweg 101
```

All records which contain the regular expression “: [DB]E :” (i.e “: DE :” or “: BE :”) were read in to the FILE variable.

REPEAT
Initiate REPEAT block

Domain: **PROCEDURE**

Command description

REPEAT initiates a REPEAT block (a REPEAT loop). Execution of the command sequence within the REPEAT block is repeated until the loop condition is met.

The loop condition is checked in the UNTIL command which terminates the REPEAT block. When the condition is not met, the system returns to the first command in the REPEAT block; otherwise command execution resumes after the UNTIL command (see also the section "REPEAT block" on page 99).

Format

REPEAT

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	1	SDP0223	Incorrect environment
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	130	SDP0099	No further address space available

Example

See section "Creating the procedure body" on page 92.

REPEAT-CMD
Repeat a command

Domain: **PROCEDURE**

Command description

The REPEAT-CMD command allows you to repeat the execution of a command. During the repetition of the command, special parts of the command (name part, operands, operand values) are substituted one after the other by the elements of an input list. These elements can be specified as records in a file, a library element or as list elements of an S variable, or they can be entered directly at the terminal. The command specified will be called for each element in the input list.

The substitution can be done in principle at any location in the command. It can also be defined at more than one location. The command is only repeated, however, in a simple loop (no nested loops).

You can specify whether the elements from the input list are to be output again in a selection menu so that you can check them.

The command is especially suited for executing commands that do not support the specification of more than one value (list-possible option).

Format

```
REPEAT-CMD                                                                  Alias: REPCMD

 CMD = <text 0..1800 with-low>

,SUBSTITUTION-LIST = *TERMINAL / <filename 1..54 without-gen-vers> / *VARIABLE(...) /
                     *LIBRARY-ELEMENT(...)

   *VARIABLE(...)
     |  VARIABLE-NAME = <composed-name 1..255>

   *LIBRARY-ELEMENT(...)
     |  LIBRARY = <filename 1..54 without-vers>
     | ,ELEMENT = <composed-name 1..64>(...)
     |    <composed-name 1..64>(...)
     |      |  VERSION = *HIGHEST-EXISTING / <composed-name 1..24>
     | ,TYPE = S / <alphanum-name 1..8>

,DIALOG-SELECTION = *NO / *YES

,CONTINUE-AFTER-ERROR = *YES / *NO
```

Operands

CMD = <text 0..1800 with-low>
Specification of the BS2000 command whose execution is to be repeated. The specification is to be enclosed in parentheses. The %* characters are to be entered as placeholders for the substitution. The name of the command is to be specified without the leading slash.
Example: `CMD=(WRITE-TEXT TEXT='%*')`

SUBSTITUTION-LIST = *TERMINAL / <filename 1..54 without-gen-vers> / VARIABLE(...) / *LIBRARY-ELEMENT(...)
Designates the input medium for the elements of the input list.

SUBSTITUTION-LIST = *TERMINAL
The elements of the input list are read from the terminal. After sending the REPEAT-CMD command, the “%>>:” string is output as a prompt. Every input for an element is to be confirmed with the $\boxed{\text{EM}}$ $\boxed{\text{DUE}}$ keys. The prompt will then reappear. The reading of input is terminated when the *END-OF-CMD string is entered.

SUBSTITUTION-LIST = <filename 1..54 without-gen-vers>
The elements of the input list are read from the specified file (ISAM file with standard codes or SAM file). The reading of input is terminated when the end of the file is detected or the *END-OF-CMD string is read.

SUBSTITUTION-LIST = VARIABLE(...)
The elements of the input list of the specified variable list are read. The reading of input is terminated when the end of the list is detected or the *END-OF-CMD string is read.

VARIABLE-NAME = <composed-name 1..255>
Name of the list variable.
The variable must exist and have been declared as data of type STRING. (It can also be declared with the ANY data type, but then it may only contain string values).

SUBSTITUTION-LIST = *LIBRARY-ELEMENT(...)
The elements of the input list from the specified library element are read. The reading of input is terminated when the end of the file is detected or the *END-OF-CMD string is read.

LIBRARY = <filename 1..54 without-vers>
Name of the library.

ELEMENT = <composed-name 1..64>(...)
Name of the library element.

VERSION = *HIGHEST-EXISTING / <composed-name 1..24>
Version of the library element.

TYPE = S / <alphanum-name 1..8>
Type of library element.

DIALOG-SELECTION = *NO / *YES
Determines if a selection menu will be output on the terminal. All elements of the input list are listed in this menu for verification purposes. Any character can be used to mark the entries.

CONTINUE-AFTER-ERROR = *YES / *NO
Determines if execution will continue with the next element from the input list or if execution will be cancelled after a command has executed with errors.

Command return code

(SC2)	SC1	Maincode	Meaning/Guaranteed messages
	0	CMD0001	No error
2	0	SDP2000	Warning: Not all elements in the input list could be successfully processed. Guaranteed message: SDP2000
	1	SDP2001	None of the elements in the input list could be successfully processed. Guaranteed message: SDP2001

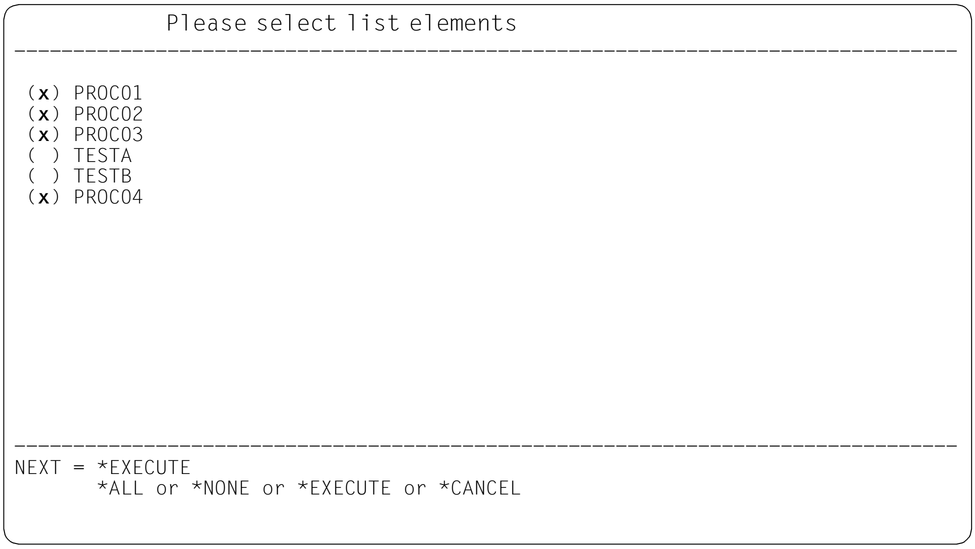
If processing was aborted after the first error due to the CONTINUE-AFTER-ERROR=*NO specification (message SDP2003), then the return code of the faulty command is returned.

Example

The ENTER-PROCEDURE command is to be called several times for the same procedure, but with different job names. The values are to be entered at the terminal and then checked once:

```
/REPEAT-CMD (ENTER-PROC FROM=PROC.WAIT-600,JOB-CLASS=JCB00050,
             JOB-NAME=%*),SUBSTITUTION-LIST=*TERMINAL,DIALOG-SELECT=*YES
%>>: PROC01
%>>: PROC02
%>>: PROC03
%>>: TESTA
%>>: TESTB
%>>: PROC04
%>>: *END-OF-CMD
```

The inputting of the values was ended with *END-OF-CMD. The job names entered will be offered again for selection in a selection menu:



```
             Please select list elements
--------------------------------------------------------------------------------

 (x) PROC01
 (x) PROC02
 (x) PROC03
 ( ) TESTA
 ( ) TESTB
 (x) PROC04










--------------------------------------------------------------------------------
NEXT = *EXECUTE
       *ALL or *NONE or *EXECUTE or *CANCEL
```

The ENTER-PROCEDURE command starts the four jobs with the job names selected (marked in the menu with an x):

```
%  JMS0066 JOB 'PROC01' ACCEPTED ON 07-03-30 AT 11:17, TSN = 050L
%  JMS0066 JOB 'PROC02' ACCEPTED ON 07-03-30 AT 11:17, TSN = 050P
%  JMS0066 JOB 'PROC03' ACCEPTED ON 07-03-30 AT 11:17, TSN = 050Q
%  JMS0066 JOB 'PROC04' ACCEPTED ON 07-03-30 AT 11:17, TSN = 050R
/
```

REPEAT-STMT
Repeat a statement

Domain: **PROCEDURE**

Command description

The REPEAT-STMT command allows you to repeat the execution of a statement (SDF format). During the repetition of the statement, special parts of the statement (name part, operands, operand values) are substituted one after the other by the elements of an input list. These elements can be specified as records in a file, a library element or as list elements of an S variable, or they can be entered directly at the terminal. The statement specified will be called for each element in the input list.

The substitution can be done in principle at any location in the statement. It can also be defined at more than one location. The statement is only repeated, however, in a simple loop (no nested loops).

You can specify whether the elements from the input list are to be output again in a selection menu so that you can check them.

The statement is especially suited for executing statement that do not support the specification of more than one value (list-possible option).

Format

```
REPEAT-STMT                                                                Alias: REPSTMT
------------------------------------------------------------------------------------------
STMT = <text 0..1800 with-low>

,SUBSTITUTION-LIST = *TERMINAL / <filename 1..54 without-gen-vers> / *VARIABLE(...) /
                     *LIBRARY-ELEMENT(...)

   *VARIABLE(...)
     |   VARIABLE-NAME = <composed-name 1..255>

   *LIBRARY-ELEMENT(...)
     |   LIBRARY = <filename 1..54 without-vers>
     |  ,ELEMENT = <composed-name 1..64>(...)
     |     <composed-name 1..64>(...)
     |       |   VERSION = *HIGHEST-EXISTING / <composed-name 1..24>
     |  ,TYPE = S / <alphanum-name 1..8>

,DIALOG-SELECTION = *NO / *YES

,CONTINUE-AFTER-ERROR = *YES / *NO
```

Operands

STMT= <text 0..1800 with-low>
Specification of the statement in SDF format whose execution is to be repeated. The specification is to be enclosed in parentheses. The %* characters are to be entered as placeholders for the substitution. The name of the statement is to be specified without the leading slash.
Example: `STMT=(WRITE-TEXT TEXT='%*')`

SUBSTITUTION-LIST = *TERMINAL / <filename 1..54 without-gen-vers> / VARIABLE(...) / *LIBRARY-ELEMENT(...)
Designates the input medium for the elements of the input list.

SUBSTITUTION-LIST = *TERMINAL
The elements of the input list are read from the terminal. After sending the REPEAT-STMT command, the “%>>:” string is output as a prompt. Every input for an element is to be confirmed with the EM DUE keys. The prompt will then reappear. The reading of input is terminated when the *END-OF-CMD string is entered.

SUBSTITUTION-LIST = <filename 1..54 without-gen-vers>
The elements of the input list are read from the specified file (ISAM file with standard codes or SAM file). The reading of input is terminated when the end of the file is detected or the *END-OF-CMD string is read.

SUBSTITUTION-LIST = VARIABLE(...)
The elements of the input list of the specified variable list are read. The reading of input is terminated when the end of the list is detected or the *END-OF-CMD string is read.

VARIABLE-NAME = <composed-name 1..255>
Name of the list variable.
The variable must exist and have been declared as data of type STRING. (It can also be declared with the ANY data type, but then it may only contain string values).

SUBSTITUTION-LIST = *LIBRARY-ELEMENT(...)
The elements of the input list from the specified library element are read. The reading of input is terminated when the end of the file is detected or the *END-OF-CMD string is read.

LIBRARY = <filename 1..54 without-vers>
Name of the library.

ELEMENT = <composed-name 1..64>(...)
Name of the library element.

VERSION = *HIGHEST-EXISTING / <composed-name 1..24>
Version of the library element.

TYPE = S / <alphanum-name 1..8>
Type of library element.

DIALOG-SELECTION = *NO / *YES
Determines if a selection menu will be output on the terminal. All elements of the input list are listed in this menu for verification purposes. Any character can be used to mark the entries.

CONTINUE-AFTER-ERROR = *YES / *NO
Determines if execution will continue with the next element from the input list or if execution will be cancelled after a command has executed with errors.

Command return code

(SC2)	SC1	Maincode	Meaning/Guaranteed messages
	0	CMD0001	No error
2	0	SDP2000	Warning: Not all elements in the input list could be successfully processed. Guaranteed message: SDP2000
	1	SDP2001	None of the elements in the input list could be successfully processed. Guaranteed message: SDP2001

If processing was aborted after the first error due to the CONTINUE-AFTER-ERROR=*NO specification (message `SDP2003`), then the return code of the faulty statement is returned.

SAVE-RETURNCODE
Save current command return code

Domain: **PROCEDURE**

Command description

SAVE-RETURNCODE saves the current command return code so that it can be evaluated with SDF-P functions. The predefined functions SUBCODE1( ), SUBCODE2( ) and MAINCODE( ) are available for evaluating the return code (see chapter “Predefined functions” on page 347).

SAVE-RETURNCODE is necessary if an error did not occur but the return code is to be evaluated.

The return code which is saved is then available until the next time an error occurs (in any command), or until a SAVE-RETURNCODE command is executed.

Note

If the IF-CMD-ERROR command is entered, SDF-P implicitly executes a SAVE-RETURNCODE command.

Format

SAVE-RETURNCODE

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	130	SDP0099	No further address space available

SAVE-VARIABLE-CONTAINER
Save variable container

Domain: **PROCEDURE**

Command description

The SAVE-VARIABLE-CONTAINER command is used to save variable containers.

Format

```
SAVE-VARIABLE-CONTAINER

CONTAINER-NAME = <composed-name 1..64 with-wild(80)>(...) /
                 list-poss(2000): <composed-name 1..64>(...)

  <composed-name 1..64 with-wild(80)>(...)
    |  ELEMENT-VERSION = *SAME / *INCREMENT
  list-poss(2000):<composed-name 1..64>(...)
    |  ELEMENT-VERSION = *SAME / *INCREMENT
```

Operands

CONTAINER-NAME =
Name of the variable container.

CONTAINER-NAME = <composed-name 1..64 with-wild(80)>(...)
Variable container which matches the specified search pattern.

ELEMENT-VERSION =
Designates the version number of the library element.

ELEMENT-VERSION = *SAME
The version number of the element remains unchanged.

ELEMENT-VERSION = *INCREMENT
The version number of the element is incremented.
LOCK-ELEMENT = *NO must be specified in OPEN-VARIABLE-CONTAINER.

CONTAINER-NAME = list-poss(2000): <composed-name 1..64>(...)
List of names of the variable containers.

ELEMENT-VERSION =
Designates the version number of the library element.

ELEMENT-VERSION = *SAME
The version number of the element remains unchanged.

ELEMENT-VERSION = *INCREMENT
The version number of the element is incremented.
LOCK-ELEMENT = *NO must be specified in OPEN-VARIABLE-CONTAINER.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	CMD0216	Do not have required privilege
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available

Example

See the SHOW-VARIABLE-CONTAINER-ATTR command, page 776.

SELECT-VARIABLE-ELEMENTS
Select elements from list variable

Domain: **PROCEDURE**

Command description

The SELECT-VARIABLE-ELEMENTS command displays the elements of an S variable (list variable) on the screen and writes any elements selected from these to another S variable. The screen output can be structured by means of headings for user guidance. Scrolling is implemented in a similar way as in guided dialog under SDF; function keys can be assigned using the /MODIFY-SDF-OPTIONS command.

Layout of screen output

The first line of the screen is an underlined heading, followed by a line with column titles indicating the type of element values displayed in the respective column. The two text lines are defined by means of the command operands HEADER-LINE and TITLE.
Each subsequent line contains the value(s) of a single element. Each line starts with a check field “(␣)”, followed by a blank and up to 75 characters. The column spacing can be defined implicitly by means of the LENGTH operand (default = ␣). See the example given at the end of this command description.

Selecting a line

An element (line) is selected by entering any character between the parentheses in the check field. The command can define that an element containing this character is to be automatically added to the output variable.

Format

```
SELECT-VARIABLE-ELEMENTS

FROM-VARIABLE = <composed-name 1..255>
,TO-VARIABLE = <composed-name 1..255>(...)

    <composed-name 1..255>(...)
      |  SELECTION-CODE = *NO / *YES

,HEADER-LINE = *NONE / <c-string 1..80>
,MESSAGE = *NONE / <text 1..240 with-low>
,DISPLAYED-ELEMENTS = *STD / list-poss(5): <composed-name 1..255>(...)

    <composed-name 1..255>(...)
      |  LENGTH = *BY-VALUE / <integer 1..75>
      | ,TITLE = *BY-ELEMENT-NAME / <c-string 1..75>
```

Operands

FROM-VARIABLE = <composed-name_1..255>
Name of the S variable (input variable, list variable) whose elements are to be displayed.

TO-VARIABLE = <composed-name_1..255>(..)
Name of an S variable (output variable) to which the selected elements are to be written. Any existing variable of that name will be overwritten.

SELECTION-CODE=
Specifies whether the character used for selecting is to be written to the output variable.

SELECTION-CODE= *NO
The character used for selecting is not to be written to the output variable.

SELECTION-CODE= *YES
The variable element SELECTION-CODE is automatically added to the output variable. If the input variable is a simple list (integer, Boolean or string), the element VALUE is also added to the output variable; the selected list element will be written to this element.
The output variable must be declared as a dynamic structure; otherwise, the command will be rejected.

HEADER-LINE = *NONE / <c-string_1..80>
Defines the heading (text line) for screen output.

MESSAGE = *NONE / <text 1..240 with-low>
Message which is displayed in the first menu below (in the last 3 lines). The message is no longer displayed after the screen has been sent.

Note: The message is only displayed in SDF ≥ V2.6B.

DISPLAYED-ELEMENTS
Designates one or more elements of the input variable and determines what will be displayed on the screen.
Only first-level values will be output for structure elements.

DISPLAYED-ELEMENTS = *STD
This specification has a different effect for simple and for complex variables:
Simple variable:
All element values will be output, one element value per line.
Complex variable (structure):
Only the element value of the first structure element will be output in each case, one element value per line.

DISPLAYED-ELEMENTS = <composed-name_1..255>(...)
Names of one or more variable elements whose values are to be output, where all the values of an element will be written in the same line. Element values will be output in the order in which they are specified in the operand list. The elements must be of data type “integer”, “Boolean” or “string”.

LENGTH = *BY-VALUE / <integer_1..76>
Output length of element values. The sum of all element values output to a line must not exceed 75 characters, including spaces between individual values. Lines exceeding 75 characters are truncated. Specifying *BY-VALUE means that the (implicit) length of the element value will be used; maximum 75 characters.

TITLE = *BY-ELEMENT-NAME / <c-string_1..76>
Text indicating the element value displayed in that column; the value specified with LENGTH=.... determines the maximum length. When LENGTH=*BY-VALUE is defined, the specified text is truncated if it exceeds the maximum length. Specifying *BY-ELEMENT-NAME means that the element name will be used as a title.

Command return codes

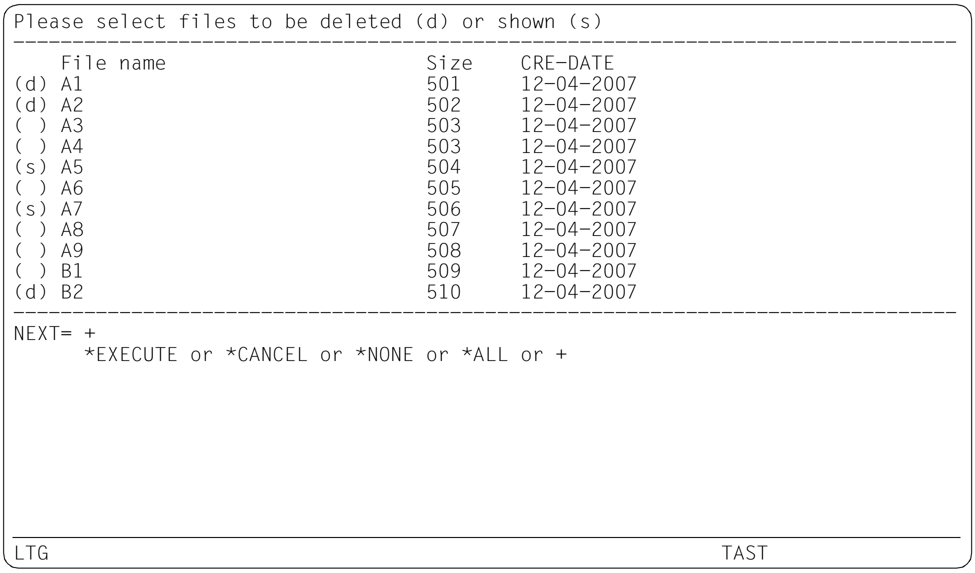
(SC2)	SC1	Maincode	Meaning/Guaranteed messages
	0	CMD0001	No error
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error
			Guaranteed messages: SDP1120, SPD1121, SPD1122
	64	SPD0259	Operation aborted: selection ignored
	130	SDP0099	No further address space available

Example

Section of a procedure (delete/display files):

The S variable OPS contains output from the /SHOW-FILE-ATTRIBUTES command. Subelements of this variable (F-NAME, F-SIZE, CRE-DATE) that are selected by means of the /SELECT-VARIABLE-ELEMENTS command are displayed on the screen. Some elements are then selected and marked with either “d” or “s”; the elements are transferred to the S variable SELECTED together with the character used for marking. The (small) procedure shown below the sample screen deletes any files marked with “d” and displays the contents of any files marked with “s”.

```
/declare-variable OPS(type=*structure),multiple-elements=*list
/declare-variable SELECTED(type=*structure),multiple-elements=*list
/declare-variable STRUC(type=*structure)

/execute-cmd (show-file-attributes *all,select=*by-attributes(-
/             size=*interval(from=500),information=*all)),-
/             structure-output=OPS,text-output=*NONE
/select-variable-elements from-variable=OPS,-
/          ,to-variable=SELECTED(selection-code=*yes),-
/          header-line='Please select files to be deleted (d) or shown (s)',-
/          displayed-elements=(F-NAME(LENGTH=54,TITLE='File name'),-
/          F-SIZE(TITLE='Size'),CRE-DATE)
```



```
Please select files to be deleted (d) or shown (s)
--------------------------------------------------------------------------------
    File name                      Size    CRE-DATE
(d) A1                             501     12-04-2007
(d) A2                             502     12-04-2007
( ) A3                             503     12-04-2007
( ) A4                             503     12-04-2007
(s) A5                             504     12-04-2007
( ) A6                             505     12-04-2007
(s) A7                             506     12-04-2007
( ) A8                             507     12-04-2007
( ) A9                             508     12-04-2007
( ) B1                             509     12-04-2007
(d) B2                             510     12-04-2007
--------------------------------------------------------------------------------
NEXT= +
      *EXECUTE or *CANCEL or *NONE or *ALL or +







LTG                                                           TAST
```

```
/for STRUC=*list(SELECTED)
/  struc.selection-code = upper-case(struc.selection-code)
/  if struc.selection-code == 'D'
/     delete-file &(STRUC.F-NAME)
/  end-if'
/  if struc.selection-code == 'S'
/     show-file &(STRUC.F-NAME)
/  end-if'
/end-for
```

SEND-DATA
Transfer data record to program

Domain: **PROCEDURE**

Command description

SEND-DATA should always be used when data records and commands are to be mixed. SEND-DATA also offers the following advantages:

– Data records can be provided with a label.
– Data records can contain comments.
– Data records can be specified with multiple lines (continuation handling).
– Data records can begin with a slash.
– Data and EOF conditions can be generated in the command via a standard interface.

If a SEND-DATA command occurs in the “data stream”, an EOF condition is not activated implicitly, as for other commands; instead, this is controlled by the RECORD operand.

Format

SEND-DATA
RECORD = ***EOF** / <text 0..1800 with-low *string-expr*>

Operands

RECORD = *EOF
Sets the EOF condition.

RECORD = <text 0..1800 with-low *string-expr*>
String expression. Evaluation of the expression produces the data record.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available

SEND-STMT
Transfer statement record to program

Domain: **PROCEDURE**

Command description

SEND-STMT should always be used when commands and statements are mixed. Just like the SEND-DATA command, a SEND-STMT command in the data stream does not implicitly activate an EOF condition.

Format

SEND-STMT
RECORD = ***EOF** / <text 0..1800 with-low *string-expr*>

Operands

RECORD = *EOF
Sets the EOF condition, i.e. end of statement entry.

RECORD = <text 0..1800 with-low *string-expr*>
String expression. Evaluation of the expression produces the statement.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available

SET-PROCEDURE-OPTIONS
Set procedure attributes

Domain: **PROCEDURE**

Command description

The SET-PROCEDURE-OPTIONS command serves to define the attributes of an *S procedure*. The command is optional. If used, however, it must be the *first* command in the procedure head. If no SET-PROCEDURE-OPTIONS command is specified, the SDF-P default settings for procedure attributes apply. The following attributes can be defined by means of the SET-PROCEDURE-OPTIONS command (the SDF-P default settings are given in parentheses):

– valid procedure call (CALLER=*ANY)
– implicit declaration of S variables (IMPLICIT-DECLARATION=*YES)
– extent of logging (LOGGING=*YES)
– procedure interruption (INTERRUPT-ALLOWED=*YES)
– procedure format (INPUT-FORMAT=*FREE-RECORD-LENGTH)
– variable replacement within data records (DATA-ESCAPE-CHAR=*NONE)
– SYSFILE environment of the current procedure level (SYSTEM-FILE-CONTEXT=*STD)
– error handling in the event of mixed data and command lines (DATA-ERROR-HANDLING=*YES)
– setting for job variable replacement (default for interactive mode: JV-REPLACEMENT=*AFTER-BUILTIN-FUNCTION; for S procedures: JV-REPLACEMENT=*NO)
– setting for error handling (ERROR-MECHANISM=*SPIN-OFF-COMPATIBLE)
– suppression of selected SDF-P messages (SUPPRESS-SDP-MSG=...)

Notes

– Explicit specification via the SET-PROCEDURE-OPTIONS command is required if defaults that are modified in the activated syntax file are to take effect for the procedure.
– SET-PROCEDURE-OPTIONS may be called no more than once and only as the first command in the procedure (see also section “Creating the procedure head” on page 81).

Format

```
SET-PROCEDURE-OPTIONS

 CALLER = *ANY / *CALL / *INCLUDE /
,IMPLICIT-DECLARATION = *YES / *NO
,LOGGING-ALLOWED = *PARAMETERS(...) / *YES / *NO
    *PARAMETERS(...)
       | CMD = *YES / *NO
       |,DATA = *YES / *NO
,INTERRUPT-ALLOWED = *YES / *NO
,INPUT-FORMAT = *FREE-RECORD-LENGTH / *BY-SDF-OPTION
,DATA-ESCAPE-CHAR = *NONE / '&&' / '#' / '*' / '@' / '$' / *STD
,SYSTEM-FILE-CONTEXT = *STD / *SAME-AS-CALLER / *OWN
,DATA-ERROR-HANDLING = *YES / *NO
,JV-REPLACEMENT = *NONE / *AFTER-BUILTIN-FUNCTION
,ERROR-MECHANISM = *SPIN-OFF-COMPATIBLE / *BY-RETURNCODE
,SUPPRESS-SDP-MSG = *NONE / list-poss(2000): <alphanum-name 7..7>
```

Operands

CALLER =
Specifies which commands can be used to call the procedure.

CALLER = *ANY
The procedure can be called with CALL-PROCEDURE as well as with INCLUDE-PROCEDURE (ENTER-PROCEDURE issues a CALL-PROCEDURE command internally).

CALLER = *CALL
The procedure can be called only with CALL-PROCEDURE (or ENTER-PROCEDURE).

CALLER = *INCLUDE
The procedure can be called only with INCLUDE-PROCEDURE.

IMPLICIT-DECLARATION = *YES / *NO
Specifies whether implicit declaration of variables is allowed.

LOGGING-ALLOWED =
Specifies whether logging is allowed for the procedure and what can be logged.

LOGGING-ALLOWED = <u>*PARAMETERS</u>(...)
In the entries below, defines what can be logged.

CMD = <u>*YES</u> / *NO
Specifies whether commands can be logged.

DATA = <u>*YES</u> / *NO
Specifies whether data can be logged.

LOGGING-ALLOWED = *YES
Logging is allowed, i.e. both commands and data can be logged.

LOGGING-ALLOWED = *NO
Logging is not allowed.

INTERRUPT-ALLOWED = <u>*YES</u>
Specifies that the procedure can be interrupted with function key K2 and resumed with the RESUME-PROCEDURE command.

INTERRUPT-ALLOWED = *NO
Specifies that the procedure cannot be interrupted with function key K2.

INPUT-FORMAT =
Designates the input format for procedure records.

INPUT-FORMAT = <u>*FREE-RECORD-LENGTH</u>
Records are interpreted in their full length. The records can contain only blanks between the continuation sign and the end of the record.

INPUT-FORMAT = *BY-SDF-OPTION
The input format of the procedure records is defined via the CONTINUATION operand in the MODIFY-SDF-OPTIONS command (see the “Commands, Vol. 1-5” manuals [3]).
The procedure record containing the SET-PROCEDURE-OPTIONS command is generally interpreted in its full length (it is equivalent to the entry FREE-RECORD-LENGTH).

DATA-ESCAPE-CHAR =
Sets the escape character. The escape character is the character which initiates expression replacement.

DATA-ESCAPE-CHAR = <u>*NONE</u>
Expression replacement is not to be carried out in data records.

DATA-ESCAPE-CHAR = / ’&&’ / ’#’ / ’*’ / ’@’ / ’$’
Defines an escape character.

DATA-ESCAPE-CHAR = *STD
The character & is used as the escape character.

SYSTEM-FILE-CONTEXT =
Determines the system file context in which the procedure is to run.

SYSTEM-FILE-CONTEXT = <u>*STD</u>
A separate system file context is set up. The system file SYSDTA is automatically assigned to the system file SYSCMD (i.e. to the procedure file). The caller’s assignments are used for all other system files. Any changes made to the assignments are valid only within the current procedure level. At the end of the procedure, the caller’s system file assignments apply again.

SYSTEM-FILE-CONTEXT = *SAME-AS-CALLER
The procedure is executed within the caller’s system file context. Any changes to the assignments within the current procedure level therefore *always* affect the caller’s system file context.

SYSTEM-FILE-CONTEXT = *OWN
A caller-specific system file context is set up. The caller’s assignments for *all* system files (including SYSDTA!) are used. Any changes made to the assignments are valid only within the current procedure level. At the end of the procedure, the caller’s system file assignments apply again.
The setting *OWN also corresponds to the previous behavior of *non-S procedures*.

DATA-ERROR-HANDLING =
Specifies that error handling is to be initiated in certain cases.

DATA-ERROR-HANDLING = <u>*YES</u>
Specifies that error handling is to be initiated in the following cases:
- if a procedure line contains data where commands are expected
- if expression replacement required in data lines cannot be executed
- if a data record contains a single escape character.

DATA-ERROR-HANDLING = *NO
No error handling of the cases described above is initiated. &varname remains unchanged in the data if varname is not known as a function or a variable.

JV-REPLACEMENT =
Specifies whether job variable replacement is to be carried out during expression replacement.

JV-REPLACEMENT = <u>*NONE</u>
During expression replacement, names are not interpreted as job variable names.

JV-REPLACEMENT = *AFTER-BUILTIN-FUNCTION
In an expression in the form &(name), “name” is interpreted as a job variable name if there is no variable or built-in function with this name. This operand value is provided to permit behavior compatible with non-S procedures during expression replacement. Since the job variable name can be overwritten at any time by new variable declarations or built-in functions, we strongly recommend that this operand value should not be used; instead, job variable replacement should be executed by using the built-in function JV (i.e. by entering &(JV(’name’)).

ERROR-MECHANISM =
Specifies whether error handling is to be initiated in a manner compatible with the spin-off behavior of non-S procedures or whether “subcode1 not equal to zero” is to be taken into account. The operand setting has no influence on error handling for statements.

ERROR-MECHANISM = *SPIN-OFF-COMPATIBLE
Error handling is to be initiated in a manner compatible with the previous spin-off behavior. Subcode1 is **not** taken into account. This ensures that the behavior of S procedures created under BS2000 V10.0 remains compatible.

ERROR-MECHANISM = *BY-RETURNCODE
Error handling is initiated if subcode1 of the last command return code is not equal to zero. The spin-off behavior is not taken into account. If *BY-RETURNCODE is specified, the error handling in S procedures must be matched to the command return codes of the various commands.

Note
In order to avoid problems which could result from the modification of the default value in the user syntax file, the selected value should be specified explicitly in the procedure.

SUPPRESS-SDP-MSG =
Specifies whether output is to be suppressed for specific SDF-P messages (message class SDP). The option is valid for the calling procedure only (i.e. it is not “inherited” by other procedures).

SUPPRESS-SDP-MSG = *NONE
No message output suppressed; all SDF-P messages are to be output.

SUPPRESS-SDP-MSG = list-poss(2000): <alphanum-name 7..7>
Set of SDF-P messages that are not to be output.

Command return codes

SET-PROCEDURE-OPTIONS may be called only as the first command in the procedure head of an S procedure. SDF-P will detect any error in the procedure head during preanalysis and will subsequently abort the procedure call.

The following return codes can thus appear only if SET-PROCEDURE-OPTIONS is used outside the procedure head.

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	130	SDP0099	No further address space available

Example

See the MODIFY-PROCEDURE-OPTIONS command, page 692.

SET-VARIABLE
Assign value to variable

Domain: **PROCEDURE**

Command description

SET-VARIABLE can be used to assign values to simple as well as complex variables.

When assigning simple variables or variable elements, the data types must match. The structure of the variables must also match when assigning complex variables globally.

Note the following when complex variables are positioned on both sides of the assignment:

– Are variable elements overwritten?
– Is the complex variable extended?
– Are variable elements of the right-hand complex variable ignored?

If the assignment affects complex variables having the type “array”, note that the order in which the contents of array elements are assigned to other elements is determined by the order of the array elements on the right-hand side of the assignment. The beginning and end of a section of list elements can be defined with list variables.

When entering the command, it is sufficient to enter

/<variable$_1$> = <variable$_2$> / <text>

instead of

/SET-VARIABLE <variable$_1$> = <variable$_2$> / <text>

Entering the command without using the command name is recommended for performance reasons (see page 272).

Note

The operands of the SET-VARIABLE command are evaluated only by SDF-P and must be entered as shown below. The command name may be omitted. The SDF abbreviation rules apply to the operands. SDF functions such as information about possible operand values or a correction dialog are not available at the operand level. In guided dialog, SDF provides only an input field with “# =”.

Format

```
SET-VARIABLE                                                    Kurzname: STV

<composed-name_1 1..255> = <text 0..1800 with-low expr> / <composed-name_2 1..255> /
                           *STRING-TO-VARIABLE(...) / *LIST(...)

  *STRING-TO-VARIABLE(...)
    |  STRING = <text 0..1800 with-low expr>
    | ,VALUE-TYPE = *STD / *STRING

  *LIST(...)
    |  LIST-NAME = <composed-name 1..255>
    | ,FROM-INDEX = *FIRST / *LAST / <integer 1..2147483647>
    | ,NUMBER-OF-ELEMENTS = 1 / *REST / <integer 1..2147483647>

,WRITE-MODE = *REPLACE / *MERGE / *EXTEND / *PREFIX
```

Operands

<composed-name$_1$ 1..255> =
Designates the variable whose value or contents are to be assigned to a different variable. The variable <composed-name$_1$> can be a simple or a complex variable.
A simple variable is assigned a value that is determined by an expression. The expression must return a result whose type matches the data type used to declare the variable.
The rules for assigning values to complex variables are summarized in the table at the end of this command description.
If the variable is linked to another job variable through the container mechanism, then the result of the expression must fulfill the following conditions: data type = STRING, maximum length = 255 bytes.

<composed-name$_1$ 1..255> = <text 0..1800 with-low *expr*>
The variable composed-name$_1$ is assigned a value determined by an expression.

<composed-name$_1$ 1..255> = <composed-name$_2$ 1..255>
The variable composed-name$_1$ is assigned the contents of the variable composed-name$_2$.
The variable composed-name$_2$ is a simple or a complex variable.
If composed-name$_1$ is a list element, then this element must already exist, it cannot be created implicitly, even when implicit declarations are permitted. (Exception: The list header can be created with /list# = <value>). The rules for assigning complex variables are presented in the table at the end of this operand description.

<composed-name$_1$ 1..255> = *STRING-TO-VARIABLE(...)
Specifies that a string will be converted to an S variable of type Structure in accordance with the conversion rules (see section "Converting SDF command strings to S variables and vice versa" on page 180).

STRING = <text 0..1800 with-low *expr*>
Input string to be converted to an S variable.

VALUE-TYPE =
Specifies if the input string is to be converted to a variable of type string all the time or depending on its value. See conversion rule 4 on page 180 for more information.

VALUE-TYPE = *STD
The input string will be converted depending on its value (string/integer/Boolean). For empty list elements or list elements consisting of spaces, the corresponding elements of list variables are not created.

VALUE-TYPE = *STRING
The input string will always be converted to a variable of type string. For empty list elements or list elements consisting of spaces, the corresponding elements of list variables are created.

<composed-name$_1$ 1..255> = *LIST(...)
The variable <composed-name$_1$> is assigned elements from a list variable.
The variable <composed-name$_1$> must be a simple or complex variable depending on the number of list elements assigned.

LIST-NAME = <composed-name 1..255>
Name of the list variable.

FROM-INDEX = *FIRST / *LAST / <integer 1..2147483647>
Index of the element of the list variable beginning with which a specified number of list elements will be assigned to the variable <composed-name$_1$>.
*FIRST: The first assignment is made with the first element of the list (default setting).
*LAST assigns precisely the last element if the list. In this case the NUMBER-OF-ELEMENTS operand is ignored.

NUMBER-OF-ELEMENTS = 1 / *REST / <integer 1..2147483647>
Number of list elements to be assigned. Default: One element will be assigned.
*REST assigns all elements from the start element specified (FROM-INDEX operand) to the last element of the list.

WRITE-MODE =
Specifies where the new variable contents are to be put by the assignment.
The table at the end of this operand description indicates which variables and values can be combined during the assignment, using the operand values of WRITE-MODE =.

WRITE-MODE = <u>*REPLACE</u>
The variable name on the left-hand side of the assignment must designate a simple or complex variable.
It must be possible to overwrite the variable or the elements of complex variables. (As a rule, a variable can be overwritten only if it is a container for a job variable with a write password and this password is not in the current password list.)
The contents of variables positioned to the left of the equals sign are deleted implicitly with FREE-VARIABLE; the variable is then overwritten by the value resulting from the entry to the right of the equals sign.

Arrays
An array must be positioned on both sides of the equals sign.
The elements in the left-hand array are overwritten by the elements on the right in the order in which they occur. If the right-hand array contains more elements, the left-hand array is extended. If the right-hand array contains fewer elements, the excess elements in the left-hand array are deleted (implicit FREE-VARIABLE).

Lists
A list must be positioned on both sides of the equals sign.
The elements in the left-hand list are overwritten by the elements on the right in the order in which they occur. If the right-hand list contains more elements, the left-hand list is extended. If the right-hand list contains fewer elements, the excess elements in the left-hand list are deleted (implicit FREE-VARIABLE).

Structures
A structure must be positioned on both sides of the equals sign.

– Assignment to a static structure: elements in the left-hand structure are overwritten by the contents of elements on the right-hand side of the equals sign if the elements have the same element names. If the right-hand structure contains elements for which there is no counterpart in the left-hand structure, these elements are ignored.
– Assignment to a dynamic structure: all elements in the right-hand structure are declared implicitly as elements in the left-hand structure.

WRITE-MODE = *MERGE
The variable name on the left-hand side of the assignment must designate a complex variable having the type “array” or “structure”.
The contents of the right-hand complex variable are assigned to the complex variable to the left of the equals sign.
If the complex variables to the left and right of the equals sign are identical, WRITE-MODE = *MERGE has the same effect as WRITE-MODE = *REPLACE.

Arrays
An array must be positioned on both sides of the equals sign. The following applies if the arrays contain elements with the same array index:
The element in the left-hand array is assigned the contents of the right-hand array element, which has the same array index. Array elements for which there is no counterpart with the same array index on the left-hand side of the assignment are declared.

Structures
A structure must be specified on both sides of the equals sign.

– Assignment to a static structure: elements in the left-hand structure are assigned the contents of elements on the right-hand side of the equals sign if the elements have the same element names. If the right-hand structure contains elements for where there is no counterpart in the left-hand structure, the elements in the left-hand structure are ignored. If the left-hand structure contains elements for which there is no counterpart in the right-hand structure, these elements remain unaffected.
– Assignment to a dynamic structure: all elements in the right-hand structure are declared implicitly as elements in the left-hand structure.

WRITE-MODE = *EXTEND
Only for variables having the type “list”.
The right-hand side of the assignment must be an expression or designate a complex variable having the type “list”.

The list on the left-hand side of the assignment is extended:
– If the assignment on the right-hand side contains an expression, one element is added to the list; the result of the expression is assigned to this element.
– If the assignment on the right-hand side contains the variable name of a list, this list (right) is appended to the list (left): the corresponding number of elements is added to the original list; the contents of the elements in the right-hand list are assigned to these elements in the order in which they occur.

WRITE-MODE = *PREFIX
Only for variables having the type “list”.
The right-hand side of the assignment must be an expression or designate a complex variable having the type “list”.

The list on the left-hand side of the assignment is extended:
– If the assignment on the right-hand side contains an expression, a new element is inserted in front of what was previously the first element in the list; the result of the expression is assigned to this element.
– If the assignment on the right-hand side contains the variable name of a list, this (right-hand) list is inserted into the (left-hand) list in front of what was previously the first element. The (left-hand) list is extended forward to include the same number of elements present in the (right-hand) list. The contents of the elements in the right-hand list are assigned to the new elements in the order in which they occur.

Command return codes

During the assignment of structures, arrays or lists, it is possible that part of the command has been processed and executed when the error occurs. In this case, the result of the command is not guaranteed.

(SC2)	SC1	Maincode	Meaning/Guaranteed messages
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error Guaranteed messages: SDP1030
	130	SDP0099	No further address space available

Permissible combinations of variable types and operands

Variable	Value/variable	WRITE-MODE =			
(left)	(right)	*REPLACE	*MERGE	*EXTEND	*PREFIX
Simple variable	Expression	x	-	-	-
	Simple variable	x	-	-	-
Array	Expression	-	-	-	-
	Simple variable	-	-	-	-
	Array	x	x	-	-
	List	-	-	-	-
	Structure	-	-	-	-
List	Expression	-	-	x	x
	Simple variable	-	-	x	x
	Array	-	-	x	x
	List	x	-	x	x
	Structure	-	-	x	x
Structure	Expression	-	-	-	-
	Simple variable	-	-	-	-
	Array	-	-	-	-
	List	-	-	-	-
	Structure	x	x	-	-

Key

x Combination permitted; see the description of the relevant operand for effects
- Combination produces an error.

Example

```
/SET-VARIABLE A = B
/A = B
```

The two assignments are equivalent: the contents of variable B are assigned to variable A.

```
/SET-VARIABLE PERSON = 'HUGO'
/PERSON = 'HUGO'
```

In both assignments the string ’HUGO’ is assigned to the variable PERSON.

```
/DECLARE-VARIABLE TOTAL
/SET-VARIABLE TOTAL = -(1 + 3) * 4
/TOTAL = -(1 + 3) * 4
```

DECLARE-VARIABLE is used to declare a variable TOTAL, with the data type ANY being the default. The following two assignments are equivalent: The integer value -16 is assigned to the variable TOTAL.

Example: Arrays

```
/DECLARE-VARIABLE A, MULTIPLE-ELEMENTS = *ARRAY
/DECLARE-VARIABLE B, MULTIPLE-ELEMENTS = *ARRAY
/SET-VARIABLE A#1 = 5
/SET-VARIABLE A#3 = 3
/SET-VARIABLE B#5 = 1
/SET-VARIABLE B = A
/SHOW-VARIABLE B
```

Output

```
B#1 = 5
B#3 = 3
```

Variant

```
/SET-VARIABLE A#4 = 5
/SET-VARIABLE B#5 = 1
/SET-VARIABLE A#6 = 3
/SET-VARIABLE B = A, MODE = *MERGE
/SHOW-VARIABLE B
```

Output

```
B#4 = 5
B#5 = 1
B#6 = 3
```

Example: Structures

```
/DECLARE-VARIABLE S1(TYPE = *STRUCTURE(*BY-SYSCMD))
/ BEGIN-STRUCTURE
/ DECLARE-ELEMENT X(INITIAL-VALUE = 11)
/ DECLARE-ELEMENT Y(INITIAL-VALUE = 12)
/ DECLARE-ELEMENT Z(INITIAL-VALUE = 13)
/ END-STRUCTURE
/DECLARE-VARIABLE S2(TYPE = *STRUCTURE(*DYNAMIC))
/ DECLARE-ELEMENT S2.X('AB')
/ DECLARE-ELEMENT S2.Y('CD')
/ SET-VARIABLE S1 = S2
/SHOW-VARIABLE S1, SELECT=*BY-ATTRIBUTES(INITIALIZATION=*ANY)
```

Output

```
S1.X = AB
S1.Y = CD
S1.Z = *NO-INIT
```

Variant

```
/DECLARE-VARIABLE S1(TYPE = *STRUCTURE(*BY-SYSCMD))
/ BEGIN-STRUCTURE
/ DECLARE-ELEMENT X(INIT = 11)
/ DECLARE-ELEMENT Y(INIT = 12)
/ DECLARE-ELEMENT Z(INIT = 13)
/ END-STRUCTURE
/DECLARE-VARIABLE S2(TYPE = *STRUCTURE(*DYNAMIC))
/ DECLARE-ELEMENT S2.X('AB')
/ DECLARE-ELEMENT S2.Y('CD')
/ SET-VARIABLE S2 = S1
/SHOW-VARIABLE S2
```

Output

```
S2.X = 11
S2.Y = 12
S2.Z = 13
```

Example: managing lists of structures

The following example shows how lists of structures can be managed. A list of structures is created, containing the name and telephone number of every user. When the list is complete, it is copied into a file.

```
/BEGIN-STRUCTURE MYLAYOUT
/  DECLARE-ELEMENT NAME(TYPE=*STRING)
/  DECLARE-ELEMENT TELEPHONENUMBER(TYPE=*STRING)
/END-STRUCTURE
/DECLARE-VARIABLE LIST-OF-STRUCT(TYPE=*STRUCTURE(MYLAYOUT))-
/                  ,MULTIPLE-ELEMENTS=*LIST
/DECLARE-VARIABLE STRUCT(TYPE=*STRUCTURE(MYLAYOUT))
/READ-VARIABLE VARIABLE-NAME = STRUCT.NAME,INPUT=*TERMINAL-
/          (PROMPT-STRING = 'ENTER A USER NAME (END WITH '''')')
/WHILE (STRUCT.NAME <>'')
/   READ-VARIABLE VARIABLE-NAME = STRUCT.TELEPHONENUMBER,INPUT=*TERMINAL-
/         (PROMPT-STRING = 'ENTER A USER TELEPHONE#')
/   LIST-OF-STRUCT=STRUCT,WRITE-MODE=*EXTEND
/   READ-VARIABLE VARIABLE-NAME = STRUCT.NAME,INPUT=*TERMINAL-
/          (PROMPT-STRING = 'ENTER A USER NAME (END WITH '''')')
/END-WHILE
/
/SHOW-VARIABLE LIST-OF-STRUCT
```

Examples: converting an SDF command string

Example 1

```
/DECLARE-VARIABLE MYSTRUCT(TYPE=*STRUCTURE(*DYNAMIC))
/SET-VARIABLE MYSTRUCT = *STRING-TO-VARIABLE-
/('OPERAND1=VALUE1(OPERAND2=VALUE2)')
/SHOW-VARIABLE MYSTRUCT
```

Output

```
MYSTRUCT.OPERAND1.SYSSTRUC = VALUE1
MYSTRUCT.OPERAND1.OPERAND2 = VALUE2
```

Example 2

```
/DCV V1(TYPE=*ANY),MULT-ELEM=*LIST
/DCV V2(TYPE=*ANY),MULT-ELEM=*LIST
/S = '(A,B,1, ,,F)'
/V1 = *STR-TO-VAR(S)
/V2 = *STR-TO-VAR(S,VAL-TYPE=*STR)
/SHV V1,VAL=*C-LIT,LIST-INDEX=YES
V1#1 = 'A'
V1#2 = 'B'
V1#3 = 1
V1#4 = 'F'
/SHV V2,VAL=*C-LIT,LIST-INDEX=YES
V2#1 = 'A'
V2#2 = 'B'
V2#3 = '1'
V2#4 = ' '
V2#5 = ''
V2#6 = 'F'
```

SHOW-STREAM-ASSIGNMENT
Show S variable stream

Domain: **PROCEDURE**

Command description

SHOW-STREAM-ASSIGNMENT shows the current assignment of the specified S variable streams.

Format

```
SHOW-STREAM-ASSIGNMENT

STREAM-NAME = *ALL / *STD-STREAMS / <structured-name 1..20 with-wild(40)> /
                list-poss(100): <structured-name 1..20>
,INFORMATION = *CURRENT-ASSIGNMENT / *FINAL-DESTINATION
,OUTPUT = *SYSOUT / *SYSLST
```

Operands

STREAM-NAME =
Name of the S variable stream to be displayed.

STREAM-NAME = *_ALL_
All the S variable streams which are visible in the current procedure will be listed. In other words, all the S variable streams which were created in the current procedure or in any dependent procedure are listed.

STREAM-NAME = *STD-STREAMS
All the standard variable streams which are implemented in the system will be displayed. The names of these variable streams all have the prefix “SYS”. That is, the output comprises all the SYS streams which are listed in the description of the STREAM-NAME operand of the ASSIGN-STREAM command.

STREAM-NAME = <structured-name 1..20 with-wild(40)>
All the S variable streams which match this search pattern will be displayed.

STREAM-NAME = list-poss(100): <structured-name 1..20>
List of the names of S variable streams which are to be displayed.

INFORMATION =
Specifies what information has to be output.

INFORMATION = *CURRENT-ASSIGNMENT
Outputs the name which is set in the TO operand of the ASSIGN-STREAM command.
If the variable stream has been assigned to another variable stream name, then this name is output.

INFORMATION = *FINAL-DESTINATION
Outputs the name of the current server which is linked to the S variable stream.
If the variable stream has been assigned to another variable stream name, then the last assignment is output. If the variable stream is assigned to *STD, *DUMMY, *VAR or *SERVER, this value will be output.

OUTPUT =
Specifies where the output from the command is to be sent.

OUTPUT = *SYSOUT
Output is sent to SYSOUT. The ASSIGN-SYSOUT or ASSIGN-STREAM command can be used to stipulate output to an S variable or an S variable stream, respectively.

OUTPUT = *SYSLST
Output is written to SYSLST only. Structured output is not supported.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	CMD2009	Error during S variable replacement
	64	CMD0216	Do not have required privilege
	64	OPS0001	No storage for S variables
	64	SDP0091	Semantic error
	64	SDP0517	Specified variable steam name does not exist
	64	SDP0519	No match for specified wildcards

Example

Input

```
/DECLARE-VARIABLE OPS-VAR(TYPE=*STRUCTURE),MULTIPLE-ELEMENTS=*LIST
/ASSIGN-STREAM SYSINF,TO=*VARIABLE(OPS-VAR)
/ASSIGN-SYSOUT TO=#ERROR-SYSOUT
/SHOW-STREAM-ASSIGNMENT SYSINF
```

Output

```
STREAM-NAME = SYSINF
 ASSIGN-LEVEL = 0
 DESTINATION = *VARIABLE
           VARIABLE-NAME = OPS-VAR
                      VAR-MODE = *EXTEND
           RETURN-VARIABLE-NAME = *NONE
           CONTROL-VAR-NAME = *NONE
           RET-CONTROL-VAR-NAME = *NONE
```

Structured outputs

To permit structured output into variables, the following operands are supported for the SHOW-STREAM-ASSIGNMENT command:

– STREAM-NAME (all values)
– INFORMATION (all values)

Further details, such as for example the conditions on the usage of individual variables, will be found in the following table.

Output structure

Output information	**Name of the S variable**[1]	**T**[2]	**Contents**	**Condition**
Procedure level	var#.ASS-LEV	I	<integer>	
Type of list extension for control variable	var#.CONTR-VAR-MODE	S	*EXT *PREFIX	DEST= '*VARIABLE'
Name of the control variable	var#.CONTR-VAR-NAME	S	*NONE <comp.-name 1..255>	DEST= '*VARIABLE'
Output destination	var#.DEST	S	*DUMMY *SERVER *VAR <struc.-name 1..20>	INF
Type of list extension for return control variable	var#.RET-CONTR-VAR-MODE	S	*EXT *PREFIX	DEST= '*VARIABLE'
Name of the return control variable	var#.RET-CONTR-VAR-NAME	S	*NONE <comp.-name 1..255>	DEST= '*VARIABLE'
Type of list extension for return variable	var#.RET-VAR-MODE	S	*EXT *PREFIX	DEST= '*VARIABLE'
Name of the return variable	var#.RET-VAR-NAME	S	*NONE <comp.-name 1..255>	DEST= '*VARIABLE'
Server information	var#.SERVER-INFO	S	*NONE <string 1..1800>	DEST= '*SERVER'
Name of the server	var#.SERVER-NAME	S	<struc.-name 1..30>	DEST= '*SERVER'
Name of the S variable stream	var#.STREAM-NAME	S	<struc.-name 1..20> SYSVAR SYSMSG SYSINF	
Type of list extension for S variable	var#.VAR-MODE	S	*EXT *PREFIX	DEST= '*VARIABLE'
Name of the S variable	var#.VAR-NAME	S	*NONE <comp.-name 1..255>	DEST= '*VARIABLE'

[1] The individual variables are arranged in alphabetical order in this table.

[2] The column headed T identifies the data types: B stands for Boolean, S for string and I for Integer

SHOW-STRUCTURE-LAYOUT
Output element name of structure layout

Domain: **PROCEDURE**

Command description

Output medium: SYSOUT / SYSLST / file / list variable / library element

The SHOW-STRUCTURE-LAYOUT command outputs the structure layout specified with NAME=.... . The structure layout must have first been defined with BEGIN-STRUCTURE.

Note
Static structures are output with SHOW-VARIABLE ... , INFO=*PAR (VALUE = *NONE).

Format

```
SHOW-STRUCTURE-LAYOUT                                                    Alias: SHSTRL

 NAME = *ALL / <structured-name 1..20 with-wild(40)> / list-poss(2000): <structured-name 1..20>
,SCOPE = *VISIBLE / *PROCEDURE / *CURRENT / *TASK
,OUTPUT = *SYSOUT / *SYSLST / <filename 1..54 without-gen-vers>(...) / *VARIABLE(...) /
          *LIBRARY-ELEMENT(...)
   <filename 1..54 without-gen-vers>(...)
     |  WRITE-MODE = *REPLACE / *EXTEND
   *VARIABLE(...)
     |  VARIABLE-NAME = <composed-name 1..20>
     | ,WRITE-MODE = *REPLACE / *EXTEND
   *LIBRARY-ELEMENT(...)
     |  LIBRARY = <filename 1..54 without-vers>
     | ,ELEMENT = <composed-name 1..64>(...)
     |     <composed-name 1..64>(...)
     |       |  VERSION = *HIGHEST-EXISTING / *UPPER-LIMIT / <composed-name 1..24>
     | ,TYPE = S / <alphanum-name 1..8>
```

Operands

NAME =
Designates the structure layout.

NAME = *ALL
Selects all structure layouts.

NAME = <structured-name 1..20 with-wild(40)>
Name of the structure layout to be displayed.
When the name contains wildcards, all structure layouts are displayed whose names match the specified search pattern. When a wildcard string matches no structure layout, message SPD0519 is issued.

NAME = list-poss(2000): <structured-name 1..20>
One or more names of structure layouts which are to be displayed. These are displayed in the specified order.

SCOPE =
Designates the scope of the structure layout to be output.

SCOPE = *VISIBLE
Outputs all visible structure layouts in the current procedure.
A structure layout is visible if it is not covered by a declaration in an include procedure.

SCOPE = *PROCEDURE
Outputs all structure layouts, including those covered by declarations in include procedures.

SCOPE = *CURRENT
Outputs the current structure layout: within a call procedure, this is the structure layout of the procedure; within an include procedure, this is the structure layout of the include procedure.

SCOPE = *TASK
Outputs the task-global structure layout.

OUTPUT =
Designates the output medium.

OUTPUT = *SYSOUT
Output to SYSOUT.

OUTPUT = *SYSLST
Output to SYSLST.

OUTPUT = <filename 1..54 without-gen-vers>(...)
Output to the specified SAM file.

WRITE-MODE = *REPLACE
The current contents of the file are to be overwritten.

WRITE-MODE = *EXTEND
The output is to be appended to the current contents.

OUTPUT = *VARIABLE(...)
Output to a list variable.

VARIABLE-NAME = <structured-name 1..20>
Name of the list variable.

WRITE-MODE = *REPLACE
The current contents of the list variable are to be overwritten.

WRITE-MODE = *EXTEND
The list variable is to be extended, i.e the output is to be appended to the current contents.

OUTPUT = *LIBRARY-ELEMENT(...)
Output to an element in a PLAM library.

LIBRARY = <filename 1..54 without-vers>
Name of the PLAM library.

ELEMENT = <composed-name 1..64>(...)
Name of the element.

VERSION =
Designates the version number of the element.

VERSION = *HIGHEST-EXISTING
Selects the highest existing version number.

VERSION = *UPPER-LIMIT
Selects the highest possible version number.

VERSION = <composed-name 1..24>
Selects the specified version number.

TYPE = S / <alphanum-name 1..8>
Designates the element type.

Command return codes

It is possible that part of the command has been processed and executed when the error occurs. In this case, the result of the command is not guaranteed.

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
1	0	CMD0001	Warning: no layout found
2	0	SDP2000	Warning: not all elements of the input list could be processed successfully. Guaranteed message: SDP2000
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error
	64	SDP1008	Variable does not exist
	64	SDP2001	None of the elements could be displayed
	130	SDP0099	No further address space available

Example

```
/BEGIN-STRUCTURE STRUCT-01 ——————————————————————————————————————————————  (1)
/  DECLARE-ELEMENT ( X, Y )
/END-STRUCT STRUCT-01
/BEGIN-STRUCTURE STRUCT-02
/  DECLARE-ELEMENT ( W, A )
/END-STRUCT STRUCT-02
/BEGIN-STRUCTURE STRUCT-03
/  DECLARE-ELEMENT ( U, P )
/END-STRUCTURE STRUCT-03
/SHOW-STRUCTUR-LAYOUT STRUCT-0<1,3> —————————————————————————————————————  (2)
STRUCT-01.X
STRUCT-01.Y
STRUCT-03.U
STRUCT-03.P
*END-OF-CMD
```

(1) Structure layouts `STRUCT-01`, `STRUCT-02`, `STRUCT-03` are defined one after the other.

(2) Output of structure layouts `STRUCT-01` and `STRUCT-03` is requested with the wildcard string `STRUCT-0<1,3>`. Alternatively, the two names could also be specified in a list `(STRUCT-01,STRUCT-03)`.

SHOW-VARIABLE
Output contents of variables

Domain: **PROCEDURE**

Command description

Output medium: SYSOUT / SYSLST / file / list variable / library element

Output format:

– The contents of variables of data type `INTEGER` are output as strings of the numerics 0-9, possibly prefixed by a minus sign.
– The contents of variables of data type `BOOLEAN` are output as either of the strings FALSE or `TRUE`.

With structure-type complex variables, the contents of the variable elements are output in the order of element declarations, with array-type complex variables, the contents of the variable elements are output in the numeric order of the array indices. A new output line is started for each variable.

Format

(part 1 of 2)

```
SHOW-VARIABLE                                                              Kurzname: SHV

VARIABLE-NAME = *ALL / *LIST(...) /
                      list-poss(2000): <composed-name 1..255> /<structured-name 1..20 with-wild(40)>

   *LIST(...)

      |  LIST-NAME = <composed-name 1..255>
      | ,FROM-INDEX = *FIRST / *LAST / <integer 1..2147483647>
      | ,NUMBER-OF-ELEMENTS = 1 / *REST / <integer 1..2147483647>

,SELECT = *BY-ATTRIBUTES(...)

   *BY-ATTRIBUTES(...)

      |  SCOPE = *VISIBLE / *PROCEDURE / *CURRENT / *CURRENT-PARAMETERS / *TASK-VISIBLE /
      |               *TASK-ALL / *CALLING-PROCEDURES
      | ,INITIALIZATION = *YES / *ANY
```

continued ➠

(part 2 of 2)

```
,INFORMATION = *PARAMETERS(...)

   *PARAMETERS(...)

     |  VALUE = *WITHOUT-QUOTES / *C-LITERAL / *X-LITERAL / *NONE
     | ,NAME = *FULL-NAME (...) / *ELEMENT-NAME (...) / *NONE
     |    *FULL-NAME(...)
     |       |  LIST-INDEX-NUMBER = *NO / *YES
     |    *ELEMENT-NAME(..)
     |       |  LIST-INDEX-NUMBER = *NO / *YES

,OUTPUT = *SYSOUT / *SYSLST / <filename 1..54 without-gen-vers>(...) / *VARIABLE(...) /
            *LIBRARY-ELEMENT(...)

   <filename 1..54 without-gen-vers>(...)
     |  WRITE-MODE = *REPLACE / *EXTEND

   *VARIABLE(...)
     |  VARIABLE-NAME = <composed-name 1..20>
     | ,WRITE-MODE = *REPLACE / *EXTEND

   *LIBRARY-ELEMENT(...)
     |  LIBRARY = <filename 1..54 without-vers>
     | ,ELEMENT = <composed-name 1..64>(...)
     |    <composed-name 1..64>(...)
     |       |  VERSION = *HIGHEST-EXISTING / *UPPER-LIMIT / <composed-name 1..24>
     | ,TYPE = S / <alphanum-name 1..8>
     | ,WRITE-MODE = *REPLACE / *EXTEND
```

Operands

VARIABLE-NAME =
Designates the variables to be output.

VARIABLE-NAME = *ALL
All variables having the scope specified under SCOPE are output in the lexical order of their variable names. Elements in structures are output in the order of their declarations, while array elements are output in the numerical order of their array indices.

VARIABLE-NAME = *LIST(...)
The elements of a list variable are to be output

LIST-NAME = <composed-name 1..255>
Name of the list variable.

FROM-INDEX = *FIRST / *LAST / <integer 1..2147483647>
Index of the element of the list variable with which the output is to begin.
*FIRST: The output will begin with the first element in the list; default.
Specifying *LAST causes the last element in the list to be output. In this case the NUMBER-OF-ELEMENTS operand is ignored.

NUMBER-OF-ELEMENTS = 1 / *REST / <integer 1..2147483647>
Number of list elements which are to be output. Default: one element is output.
Specifying *REST causes all elements from the specified start element (FROM-INDEX operand) to the last element in the list to be output.

VARIABLE-NAME = list-poss(2000): <composed-name 1..255>
Names of the variables to be output.
These names are output in the specified order.

VARIABLE-NAME = <structured-name 1..20 with-wild(40)>
The variables whose names match the search pattern are output is alphabetical order by name.

SELECT = *BY-ATTRIBUTES(...)
Designates the variables to be output in greater detail.

SCOPE =
Designates the scope of the variables to be output.

SCOPE = *VISIBLE
Outputs all visible variables.
A variable is visible if it is not overlaid by a declaration in an include procedure.

SCOPE = *PROCEDURE
Outputs all variables, even if they are overlaid by a declaration in an include procedure.

SCOPE = *CURRENT
Outputs the current variables, namely: within a call procedure, the variables of the call procedure; within an include procedure, the variables of the include procedure.

SCOPE = *CURRENT-PARAMETERS
Outputs the current procedure parameters, i.e. the procedure parameters within a call procedure, and the procedure parameters of the include procedure within an include procedure.

SCOPE = *TASK-ALL
Outputs all the task-global variables.

SCOPE = *TASK-VISIBLE
Outputs the imported task-global variables, or the task-global variables which were declared in the procedure.

SCOPE = *CALLING-PROCEDURE
Outputs all the variables of the scope of the calling procedure declared with IMPORT-ALLOWED = *YES. In the case of foreground procedures, this scope consists of all calling procedures starting from the dialog level, and in the case of background procedures, all calling procedures starting from the first procedure.

INITIALIZATION =
Specifies whether or not non-initialized variables are to be output.

INITIALIZATION = <u>*YES</u>
Only initialized variables are output.

INITIALIZATION = *ANY
All variables (whether initialized or non-initialized) are output.

INFORMATION = <u>*PARAMETERS</u>(...)
Specifies the information which is output.

VALUE =
Specifies whether the values of the variables should be output, and in what format.

VALUE = <u>*WITHOUT-QUOTES</u>
Outputs variables with the data type `STRING` without apostrophes.

VALUE = *C-LITERAL
Outputs variables with the data type `STRING` as C literals. For non-initialized strings, the string '*NO-INIT' is output.

VALUE = *X-LITERAL
Outputs variables with the data type `STRING` as X literals. For non-initialized strings, the string '*NO-INIT' is output.

VALUE = *NONE
The value of the variable is not output, only its name (see the NAME operand).

NAME =
Specifies the format in which the names of the variables are output.

NAME = <u>*FULL-NAME</u>(...)
Outputs full variable names.

LIST-INDEX-NUMBER = <u>*NO</u> / *YES
You can specify if the element number is to be appended to the name instead of (*LIST) for list elements.

*LIST-INDEX-NUMBER = *NO*

Variablename(*LIST) = <contents>	for the first element
Variablename(*LIST) = <contents>	for the second element, etc.

*LIST-INDEX-NUMBER = *YES*

Variablename#1 = <contents>	for the first element
Variablename#2 = <contents>	for the second element, etc.

NAME = *ELEMENT-NAME(...)
Outputs the element names of the variables. This is also the output format in which the data type `STRUCTURE` is output.

LIST-INDEX-NUMBER = <u>*NO</u> / *YES
You can specify if the output is to start with (*LIST) or with the element number.

*LIST-INDEX-NUMBER = *NO*

(*LIST) = <contents>	for the first element
(*LIST) = <contents>	for the second element, etc.

*LIST-INDEX-NUMBER = *YES*

#1 = <contents>	for the first element
#2 = <contents>	for the second element, etc.

NAME = *NONE
The names of variables are not output.

OUTPUT =
Designates the output medium.

OUTPUT = <u>*SYSOUT</u>
Output to SYSOUT.

OUTPUT = *SYSLST
Output to SYSLST. (To ensure correct printed output, each data line starts with “␣”.)

OUTPUT = <filename 1..54 without-gen-vers>(...)
Output to the specified file, which must be a SAM file.

WRITE-MODE = *REPLACE
The current contents of the file are to be overwritten.

WRITE-MODE = *EXTEND
The output is to be appended to the current contents.

OUTPUT = *VARIABLE(...)
Output to a list variable.

VARIABLE-NAME = <structured-name 1..20>
Name of the list variable.

WRITE-MODE = *REPLACE
The current contents of the list variable are to be overwritten.

WRITE-MODE = *EXTEND
The list variable is to be extended, i.e. the output is to be appended to the current contents.

OUTPUT = *LIBRARY-ELEMENT(...)
Output to an element in a PLAM library.

LIBRARY = <filename 1..54 without-vers>
Name of the PLAM library.

ELEMENT = <composed-name 1..64>(...)
Name of the element.

VERSION =
Designates the version number of the element.

VERSION = *HIGHEST-EXISTING
Selects the highest existing version number.

VERSION = *UPPER-LIMIT
Selects the highest possible version number.

VERSION = <composed-name 1..24>
Selects the specified version number.

TYPE = S / <alphanum-name 1..8>
Designates the element type.

WRITE-MODE = *REPLACE
The existing content of the element is to be overwritten.

WRITE-MODE = *EXTEND
The element is to be extended, i.e. the output is to be appended to the existing content.

Command return codes

It is possible that part of the command has been processed and executed when the error occurs. In this case, the result of the command is not guaranteed.

(SC2)	SC1	Maincode	Meaning/Guaranteed messages
	0	CMD0001	No error
1	0	CMD0001	Warning; no variable found
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error Guaranteed message: SDP1008
	130	SDP0099	No further address space available

Example

The following variable is declared in the procedure “proc1“:

```
/DECLARE-VARIABLE VARIABLE-NAME=VALUE(TYPE=*INTEGER,*INITIAL-VALUE=122),-
/SCOPE=*PROCEDURE(IMPORT-ALLOWED=*YES)
/CALL-PROCEDURE Proc2
```

Procedure ”proc2“ contains the following:

```
/SHOW-VARIABLE VARIABLE-NAME=*ALL,SELECT=*BY-ATTRIBUTES-
/(SCOPE=*CALLING-PROCEDURES)
A=122
```

Description of output formats

Depending on the entries for INFORMATION, a number of different output formats are created. The output is not the same as a sequence of SET-VARIABLE commands.

Example

```
/DECLARE-VARIABLE NAME(INIT-VALUE = 'MILLER')
/DECLARE-VARIABLE AGE(TYPE = *INTEGER, INIT-VALUE = 22)
/DECLARE-VARIABLE LANGUAGES,MULTIPLE-ELEMENTS = *LIST
/LANGUAGES = 'GERMAN', WRITE-MODE = *EXTEND
/LANGUAGES = 'ENGLISH', WRITE-MODE = *EXTEND
/DECLARE-VARIABLE GRADES(TYPE = *STRUCTURE(*BY-SYSCMD))
/BEGIN-STRUCTURE
/DECLARE-ELEMENT GERMAN(TYPE = *INTEGER, INIT-VALUE = 2)
/DECLARE-ELEMENT ENGLISH(TYPE =*INTEGER)
/END-STRUCTURE
/DECLARE-VARIABLE INTERPRETER(TYPE = *BOOLEAN,INIT-VALUE = TRUE)
```

*1. Output using SELECT = *BY-ATTRIBUTES(INITIALIZATION = *YES)*

```
/SHOW-VARIABLE VARIABLE-NAME=*ALL, SELECT=*BY-ATTRIBUTES -
(INITIALIZATION = *YES)
```

Output

```
AGE = 22
INTERPRETER= TRUE
NAME = MILLER
GRADES.GERMAN = 2
LANGUAGES(*LIST)=GERMAN
LANGUAGES(*LIST)=ENGLISH
*END-OF-CMD
```

Only initialized variables are output.

*2. Output using SELECT = *BY-ATTRIBUTES(INITIALIZATION = *ANY)*

```
/SHOW-VARIABLE VARIABLE-NAME=*ALL, SELECT=*BY-ATTRIBUTES -
/(INITIALIZATION = *ANY)
```

Output

```
AGE = 22
INTERPRETER= TRUE
NAME = MILLER
GRADES.GERMAN = 2
GRADES.ENGLISH = *NO-INIT
LANGUAGES(*LIST)=GERMAN
LANGUAGES(*LIST)=ENGLISH
*END-OF-CMD
```

All variables are output, regardless of whether or not they are initialized.

*3. Output using INFORMATION = *PARAMETERS(VALUE = *WITHOUT-QUOTES)*

```
/SHOW-VARIABLE VARIABLE-NAME=*ALL, INFORMATION = *PARAMETERS -
/(VALUE = *WITHOUT-QUOTES)
```

Output

```
AGE = 22
INTERPRETER= TRUE
NAME = MILLER
GRADES.GERMAN = 2
LANGUAGES(*LIST)=GERMAN
LANGUAGES(*LIST)=ENGLISH
*END-OF-CMD
```

String variables are output without quotes.

*4. Output using INFORMATION = *PARAMETERS(VALUE = *C-LITERAL)*

```
/SHOW-VARIABLE VARIABLE-NAME=*ALL, INFORMATION = *PARAMETERS -
/(VALUE = *C-LITERAL)
```

Output

```
AGE = 22
INTERPRETER = TRUE
NAME = 'MILLER'
GRADES.GERMAN = 2
LANGUAGES(*LIST) = 'GERMAN'
LANGUAGES(*LIST) = 'ENGLISH'
*END-OF-CMD
```

String variables are output as C literals.

*5. Output using INFORMATION = *PARAMETERS(VALUE = *X-LITERAL)*

```
/SHOW-VARIABLE VARIABLE-NAME=*ALL, INFORMATION = *PARAMETERS -
/(VALUE = *X-LITERAL)
```

Output

```
AGE = 22
INTERPRETER = TRUE
NAME = X'D4C9D3D3C5D9'
GRADES.GERMAN = 2
LANGUAGES(*LIST) = X'C7C5D9D4C1D5'
LANGUAGES(*LIST) = X'C5D5C7D3C9E2C8'
*END-OF-CMD
```

String variables are output as X literals.

*6. Output using INFORMATION = *PARAMETERS(VALUE = *NONE)*

```
/SHOW-VARIABLE VARIABLE-NAME=*ALL, INFORMATION = *PARAMETERS -
/(VALUE = *NONE)
```

Output

```
AGE
INTERPRETER
NAME
GRADES.GERMAN
LANGUAGES(*LIST)
LANGUAGES(*LIST)
*END-OF-CMD
```

Only the variable names are output.

*7. Output using INFORMATION = *PARAMETERS(NAME=*FULL-NAME)*

```
/SHOW-VARIABLE VARIABLE-NAME=GRADES, INFORMATION = *PARAMETERS -
/(NAME=*FULL-NAME)
```

Output

```
GRADES.GERMAN = 2
```

The full element name is output.

*8. Output using INFORMATION = *PARAMETERS(NAME=*ELEMENT-NAME)*

```
/SHOW-VARIABLE VARIABLE-NAME=GRADES, INFORMATION = *PARAMETERS -
/(NAME=*ELEMENT-NAME)
```

Output

```
GERMAN = 2
```

Only element names are output.

*9. Output using INFORMATION = *PARAMETERS(NAME=*NONE)*

```
/SHOW-VARIABLE VARIABLE-NAME=GRADES, INFORMATION = *PARAMETERS -
/(NAME=*NONE)
```

Output

```
2
```

Only variable values are output.

Further examples

Input

```
/DECLARE-VARIABLE STREET('XYZ WAY', *STRING)
/DECLARE-VARIABLE NUMBER(12, *INTEGER)
/DECLARE-VARIABLE NAME('HUGO')
/DECLARE-VARIABLE MARRIED(TRUE, *BOOLEAN)
/SHOW-VARIABLE *ALL
```

Output

```
NAME = HUGO
NUMBER = 12
STREET = XYZ WAY
MARRIED = TRUE
*END-OF-CMD
```

Input

```
/SHOW-VARIABLE (NAME, STRASSE, NUMMER), -
/ INFO=*PAR(VALUE=*C-LITERAL,NAME= *FULL-NAME)
```

Output

```
NAME = 'HUGO'
STREET = 'XYZ WAY'
NUMBER = 12
```

Input

```
/DECLARE-VARIABLE A(TYPE = *STRUCTURE(*DYNAMIC))

/A.C = 'TWO'
/A.D = 'THREE'
/A.B = 'ONE'
/SHOW-VARIABLE A, INFO=*PAR(VALUE=*C-LITERAL,NAME= *FULL-NAME)
```

Output

```
A.C = 'TWO'
A.D = 'THREE'
A.B = 'ONE'
```

Input

```
/SHOW-VARIABLE A, INFO=*PAR(VALUE=*C-LITERAL,NAME= *ELEMENT-NAME)
```

Output

```
C = 'TWO'
D = 'THREE'
B = 'ONE'
```

Input

```
/DECLARE-VARIABLE V2(TYPE=*ANY),MULT-ELEM=*LIST
/S = '(ANTON,BERTA,CAESAR,HUGO,FRANZ)'
/V2 = *STRING-TO-VARIABLE(S)
```

Output

```
/SHOW-VARIABLE *LIST(LIST=V2,FROM-INDEX=3,NUM-OF-ELEM=*REST),-
/ INFO=*PAR(VALUE=*C-LITERAL,LIST-INDEX-NUMBER=*YES)
V2#3 = 'CAESAR'
V2#4 = 'HUGO'
V2#5 = 'FRANZ'
```

SHOW-VARIABLE-ATTRIBUTES
Output variable attributes

Domain: **PROCEDURE**

Command description

The SHOW-VARIABLE-ATTRIBUTES command supplies information about the attributes of the specified variables. The information is output to SYSOUT (or an S variable / S variable stream) or to SYSLST.
The attributes include the name of the variable, its initial value, data type, whether it is a simple or a complex variable, etc. Variable attributes are declared using the /DECLARE-VARIABLE command or preset at the time the variable is generated.

Format

```
SHOW-VARIABLE-ATTRIBUTES

VARIABLE-NAME = *ALL / <structured-name 1...20 with-wild(40)> / <composed-name 1...255> / *LIST(...)

  *LIST(...)
    |   LIST-NAME = <composed-name 1..255>
    |  ,FROM-INDEX = *FIRST / *LAST / <integer 1..2147483647>
    |  ,NUMBER-OF-ELEMENTS = 1 / *REST / <integer 1..2147483647>

,INFORMATION = *NAME / *VARIABLE-ATTRIBUTES-ONLY / *ALL-ATTRIBUTES

,ATTACHED-INFORMATION = *NO / *YES

,OUTPUT = *SYSOUT / *SYSLST
```

Operands

VARIABLE-NAME =
Specifies the variable whose attributes are to be output.

VARIABLE-NAME = *ALL
All procedure-local, visible variables are to be output.

VARIABLE-NAME = <composed-name 1...255>
Name of the variable whose attributes are to be output.

VARIABLE-NAME = <structured-name 1...20 with-wild(40)>
Specification of one or more variables via a name including wildcards.

VARIABLE-NAME = *LIST(...)
The attributes of the elements of a list variable are to be output.

LIST-NAME = <composed-name 1..255>
Name of the list variable.

FROM-INDEX = *FIRST / *LAST / <integer 1..2147483647>
Index of the element of the list variable with which the output is to begin.
*FIRST: the output will begin with the first element in the list; default.
Specifying *LAST causes the attributes of the last element in the list to be output. In this case the NUMBER-OF-ELEMENTS operand is ignored.

NUMBER-OF-ELEMENTS = 1 / *REST / <integer 1..2147483647>
Number of list elements whose attributes are to be output. Default: the attributes of one element are output.
Specifying *REST causes the attributes of all elements from the specified start element (FROM-INDEX operand) to the last element in the list to be output.

INFORMATION =
Defines the extent of the information to be output.

INFORMATION = *NAME
Only the name of the variable is to be output.

INFORMATION = *VARIABLE-ATTRIBUTES-ONLY
All attributes specified with the DECLARE-VARIABLE command are to be output.

INFORMATION = *ALL-ATTRIBUTES
All attributes specified with the /DECLARE-VARIABLE command are to be output. A list of the names of variable elements is additionally output.

ATTACHED-INFORMATION =
Determines whether the attributes of variable elements are to be output.

ATTACHED-INFORMATION = *NO
Only the attributes of the variable are to be output.

ATTACHED-INFORMATION = *YES
The attributes of the variable as well as its elements are to be output.

OUTPUT =
Determines whether the information is to be output to SYSOUT or to SYSLST.

OUTPUT = *SYSOUT
The information is output to SYSOUT. The ASSIGN-SYSOUT or ASSIGN-STREAM command can be used to stipulate output to an S variable or an S variable stream.

OUTPUT = *SYSLST
The information is output to SYSLST only.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error (variable does not exist)
			Guaranteed message: SDP1008
	130	SDP0099	No further address space available

Example

```
/declare-variable S(type = *structure),multiple-elements=*array
/S#81.A = 'First Value'
/S#81.B#5 = 'Second Value'
/S#81.B#27 = 'Third Value'
/S#97.A = 'First Value'
/S#97.B#8 = 'Second Value'
/S#97.B#13 = 'Third Value'


/show-variable-attributes s,info=*all-attributes, attached-information=*yes

VARIABLE-NAME = S
 TYPE = *STRUCTURE(DEFINITION = *DYNAMIC)
 MULTIPLE-ELEMENTS = *ARRAY(LOWER-BOUND = 0, UPPER-BOUND = 2147483647)
 SCOPE = *PROCEDURE(IMPORT-ALLOWED = *NO)
 CONTAINER = *STD
 CONSTANT = *NO
 VALUE-TYPE = *NONE
 VALUE =
 NUMBER-OF-ELEMENTS = 2
 ELEM#1 = S#81
 ELEM#2 = S#97
VARIABLE-NAME = S#81
 TYPE = *STRUCTURE(DEFINITION = *DYNAMIC)
 MULTIPLE-ELEMENTS = *NO
 SCOPE = *PROCEDURE(IMPORT-ALLOWED = *NO)
 CONTAINER = *STD
 CONSTANT = *NO
 VALUE-TYPE = *NONE
 VALUE =
 NUMBER-OF-ELEMENTS = 2
 ELEM#1 = S#81.A
 ELEM#2 = S#81.B
```

```
VARIABLE-NAME = S#81.A
 TYPE = *ANY
 MULTIPLE-ELEMENTS = *NO
 SCOPE = *PROCEDURE(IMPORT-ALLOWED = *NO)
 CONTAINER = *STD
 CONSTANT = *NO
 VALUE-TYPE = *STRING
 VALUE = First Value
 NUMBER-OF-ELEMENTS = 0
VARIABLE-NAME = S#81.B
 TYPE = *ANY
 MULTIPLE-ELEMENTS = *ARRAY(LOWER-BOUND = -2147483648, UPPER-BOUND =
                     2147483647)
 SCOPE = *PROCEDURE(IMPORT-ALLOWED = *NO)
 CONTAINER = *STD
 CONSTANT = *NO
 VALUE-TYPE = *NONE
 VALUE =
 NUMBER-OF-ELEMENTS = 2
 ELEM#1 = S#81.B#5
 ELEM#2 = S#81.B#27
VARIABLE-NAME = S#81.B#5
 TYPE = *ANY
 MULTIPLE-ELEMENTS = *NO
 SCOPE = *PROCEDURE(IMPORT-ALLOWED = *NO)
 CONTAINER = *STD
 CONSTANT = *NO
 VALUE-TYPE = *STRING
 VALUE = Second Value
 NUMBER-OF-ELEMENTS = 0
VARIABLE-NAME = S#81.B#27
 TYPE = *ANY
 MULTIPLE-ELEMENTS = *NO
 SCOPE = *PROCEDURE(IMPORT-ALLOWED = *NO)
 CONTAINER = *STD
 CONSTANT = *NO
 VALUE-TYPE = *STRING
 VALUE = Third Value
 NUMBER-OF-ELEMENTS = 0
VARIABLE-NAME = S#97
 TYPE = *STRUCTURE(DEFINITION = *DYNAMIC)
 MULTIPLE-ELEMENTS = *NO
 SCOPE = *PROCEDURE(IMPORT-ALLOWED = *NO)
 CONTAINER = *STD
 CONSTANT = *NO
 VALUE-TYPE = *NONE
 VALUE =
 NUMBER-OF-ELEMENTS = 3
 ELEM#1 = S#97.A
 ELEM#2 = S#97.B
 ELEM#3 = S#97.B13
VARIABLE-NAME = S#97.A
 TYPE = *ANY
 MULTIPLE-ELEMENTS = *NO
 SCOPE = *PROCEDURE(IMPORT-ALLOWED = *NO)
 CONTAINER = *STD
 CONSTANT = *NO
 VALUE-TYPE = *STRING
 VALUE = First Value
 NUMBER-OF-ELEMENTS = 0
VARIABLE-NAME = S#97.B
 TYPE = *ANY
 MULTIPLE-ELEMENTS = *ARRAY(LOWER-BOUND = -2147483648, UPPER-BOUND =
                     2147483647)
 SCOPE = *PROCEDURE(IMPORT-ALLOWED = *NO)
 CONTAINER = *STD
 CONSTANT = *NO
 VALUE-TYPE = *NONE
 VALUE =
 NUMBER-OF-ELEMENTS = 1
 ELEM#1 = S#97.B#8
VARIABLE-NAME = S#97.B#8
 TYPE = *ANY
 MULTIPLE-ELEMENTS = *NO
 SCOPE = *PROCEDURE(IMPORT-ALLOWED = *NO)
 CONTAINER = *STD
 CONSTANT = *NO
 VALUE-TYPE = *STRING
 VALUE = Second Value
 NUMBER-OF-ELEMENTS = 0
VARIABLE-NAME = S#97.B13
 TYPE = *ANY
 MULTIPLE-ELEMENTS = *NO
 SCOPE = *PROCEDURE(IMPORT-ALLOWED = *NO)
 CONTAINER = *STD
 CONSTANT = *NO
 VALUE-TYPE = *STRING
 VALUE = Third Value
 NUMBER-OF-ELEMENTS = 0
```

Note

If the variable is a list variable, the names of its elements are not listed. Variable elements are identified by the variable name followed by the “#” character and the (sequential) element number instead.

Formatted output

The following operands are supported for formatted output in variables for the SHOW-VARIABLE-ATTRIBUTES command:

– VARIABLE-NAME (all values)
– INFORMATION (all values)
– ATTACHED-INFORMATION (all values)

Additional information such as the conditions for assigning values to individual variables can be found in the following table.

Output structure

Output information	Name of the S variable[1]	T[2]	Contents
Does the variable have a constant value (/DECLARE-CONSTANT)?	var#.CONSTANT	S	*YES *NO
Type of the variable container	var#.CONTAIN	S	<composed-name 1..64> *STD *VAR *JV
Name of the variable container	var#.CONTAIN-NAME	S	<structured-name 1..20> <filename 1..54>
Element name	var#.ELEM(*LIST)	S	<composed-name 1..255>
Import of the variable permitted	var#.IMP-ALLOW	S	*YES *NO
Maximum number (limit) of list elements	var#.LIM	I	<integer 0..2147483647>
Lower limit of the array index	var#.LOWER-BOUND	I	<integer -2147483648..2147483647>
The variable is a list variable, an array variable or a simple variable (*NO)	var#.MULT-ELEM	S	*LIST *ARRAY *NO
Number of variable elements	var#.NUM-OF-ELEM	I	<integer 0..2147483647>
Scope of the variable	var#.SCOPE	S	*INC *PROC *TASK
Name of the structure layout	var#.STRUCT-DEFI	S	<structured-name 1..20> *BY-SYSCMD *DYNAMIC
Type of the variable	var#.TYPE	S	*ANY *BOOLEAN *STRING *INTEGER *STRUCT
Upper limit of the array index	var#.UPPER-BOUND	I	<integer -2147483648..2147483647>

Output information	Name of the S variable[1]	T[2]	Contents
Value of the variable	var#.VALUE	S I B	<string 0..4096> <integer -2147483648..2147483647> FALSE TRUE
Type of the variable value	var#.VALUE-TYPE	S	*NONE *BOOLEAN *STRING *INTEGER
Name of the variable	var#.VAR-NAME	S	<composed-name 1..255>

1 Variable names are sorted alphabetically

2 The T column identifies the data types: B = Boolean, I = INTEGER, S = string

SHOW-VARIABLE-CONTAINER-ATTR
Display open variable containers

Domain: **PROCEDURE**

Command description

The SHOW-VARIABLE-CONTAINER-ATTR command displays all the open variable containers.

Format

SHOW-VARIABLE-CONTAINER-ATTR
CONTAINER-NAME = ***ALL** / <composed-name 1..64 with-wild(80)> / list-poss: <composed-name 1..64> ,**CONTAINER-SCOPE** = ***VISIBLE** / ***PROC**EDURE / ***CURR**ENT / ***TASK** ,**OUTPUT** = ***SYSOUT** / ***SYSLST**

Operands

CONTAINER-NAME =
Name of the open variable container which is to be displayed.

CONTAINER-NAME = *ALL
All the container variables which are open are displayed.

CONTAINER-NAME = <composed-name 1..64 with-wild(80)>
All the open container variables which match the specified search pattern are displayed.

CONTAINER-NAME = list-poss: <composed-name 1..64>
List of the variable containers which are to be displayed.

CONTAINER-SCOPE =
Scope of the variable containers which are to be displayed.

CONTAINER-SCOPE = *VISIBLE
The variable containers which are displayed are those which can be accessed from the current procedure level. The current names of variable containers conceal variable containers at higher procedure levels or the task level.

CONTAINER-SCOPE = *PROCEDURE
The variable containers which are displayed are those which can be accessed from the current procedure level and which were opened with the scope PROCEDURE.

CONTAINER-SCOPE = *CURRENT
The variable containers which are displayed are those which can be accessed from the current procedure level and which were opened with the scope CURRENT.

CONTAINER-SCOPE = *TASK
Variable containers which have the scope TASK are displayed.

OUTPUT =
Address for the output information.

OUTPUT = <u>*SYSOUT</u>
The output information is sent to SYSOUT and/or to variables, depending on the ASSIGN-SYSOUT and ASSIGN-STREAM assignments.

OUTPUT = *SYSLST
The output information is sent to SYSLST, regardless of how the SYSINF variable stream has been assigned.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	CMD0216	Do not have required privilege
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available

Example

Input

```
/OPEN-VARIABLE-CONTAINER MY-CONT1, *LIBRARY-ELEMENT(#MY-CONTAINER-LIB)
/SHOW-VARIABLE-CONTAINER-ATTR MY-CONT1
```

Output

```
CONTAINER-NAME = MY-CONT1
  FROM-FILE = *LIBRARY-ELEMENT
    LIBRARY = :1OSN:$QM123.S.152.OWDK.MY-CONTAINER-LIB
    ELEMENT = MY-CONT1
    VERSION = *HIGHEST-EXISTING
  LOCK      = *NO
  SCOPE     = *PROCEDURE
```

Input

```
/SAVE-VARIABLE-CONTAINER MY-CONT1
/CLOSE-VARIABLE-CONTAINER MY-CONT1
```

Structured output

To permit structured output into variables, the following operands are supported for the SHOW-VARIABLE-CONTAINER-ATTR command:

– CONTAINER-NAME (all values)
– CONTAINER-SCOPE (all values)

Further details, such as for example the conditions on the usage of individual variables, will be found in the following table.

Output structure

Output information	Name of the S variable[1]	T[2]	Contents
Name of the variable container	var#.CONTAIN-NAME	S	<comp.-name 1..64>
Name of the library element which contains the variable container	var#.FROM-F	S	*LIB-ELEM(...)
Read/write access to the library element	var#.LOCK	S	*NO *YES
Saving of the variable container at EXIT-JOB/LOGOFF (only if SCOPE=*TASK)	var#.SAVE-AT-TERM	S	*NO *YES
Scope of the variable container	var#.SCOPE	S	*INC *PROC *TASK

[1] Variable names are sorted alphabetically

[2] The T column identifies the data types: S stands for string

SORT-VARIABLE
Sort list variable

Domain: **PROCEDURE**

Command description

The SORT-VARIABLE command sorts the elements of a list variable in ascending or descending order according to their content. Only list variables whose elements are simple variables of the same type (STRING, INTEGER, BOOLEAN oder ANY) can be sorted.

Format

SORT-VARIABLE
VARIABLE-**NAME** = list-poss(2000): <composed-name 1..255> ,**SORT**ING-**ORDER** = ***ASCEND**ING / ***DESCEND**ING

Operands

VARIABLE-NAME = list-poss(2000): <composed-name 1..255>
Specifies the variable to be sorted. When specified in list form, multiple variable names can be specified.

SORTING-ORDER = *ASCENDING / *DESCENDING
Determines the sorting order.

SORTING-ORDER = *ASCENDING
Sorts the elements from the lowest to the highest value. Depending on the type of elements to be compared, the size comparison is performed as follows:

– Elements of the type STRING
  In the case of a list with elements of the type STRING, the elements are sorted in ascending order according to the size of the string values. The value of the hexadecimal coding is compared for each byte (see also “String comparison” on page 261). This type of comparison also enables X literals to be sorted.

  The following applies:
  – When two strings begin with the same values but have different lengths, the shorter string contains the lower value.
  – An empty string always has the lowest possible value.

Example: The following size relationship results for the strings `BC`, `ABC` and `BCD`:

`ABC` < `BC` and `ABC` < `BCD` and `BC` < `BCD`

Sorting in ascending order results in the sequence `ABC`, `BC` and `BCD`.

– Elements of the type INTEGER
In the case of a list with elements of the type INTEGER, the elements are sorted in ascending order according to the size of the integer values (numerical comparison).

– Elements of the type BOOLEAN
In the case of a list with elements of the type BOOLEAN, the elements are sorted in ascending order according to the size of the Boolean values. The following applies: the value “FALSE” is less then “TRUE”.

– Elements of the type ANY
In the case of a list with elements of the type ANY, the elements are sorted in ascending order according to the type of the current value (ascending order of string, integer or Boolean values, as described above).
If the elements in the list have values of different types, the list cannot be sorted.

SORTING-ORDER = *DESCENDING
Sorts the elements from the highest to the lowest value. The type of size comparison depends on the type of elements to be compared (see SORTING-ORDER=*ASCENDING).

Command return code

When lists are sorted, it is possible that some of the commands will have been processed and executed before an error occurrs. In this case the result of the command is not guaranteed.

(SC2)	SC1	Maincode	Meaning/Guaranteed messages
	0	CMD0001	Ohne Fehler
1	0	SDP2000	Warning: some elements could not be sorted
	64	SDP2001	None of the elements could be sorted

Example

```
/DECLARE-VARIABLE LANGUAGE-LIST (TYPE = *STRING), -
/           MULTIPLE-ELEMENTS = *LIST
/ LANGUAGE-LIST = 'GERMAN', WRITE-MODE=*EXTEND
/ LANGUAGE-LIST = 'ENGLISH', WRITE-MODE=*EXTEND
/ LANGUAGE-LIST = 'FRENCH', WRITE-MODE=*EXTEND
/ LANGUAGE-LIST = 'ITALIAN', WRITE-MODE=*EXTEND
/ LANGUAGE-LIST = 'GREEK', WRITE-MODE=*EXTEND
/
/SORT-VARIABLE VARIABLE-NAME=LANGUAGE-LIST, SORTING-ORDER=*ASCENDING
/SHOW-VARIABLE VARIABLE-NAME=LANGUAGE-LIST
LANGUAGE-LIST(*LIST) = ENGLISH
LANGUAGE-LIST(*LIST) = FRENCH
LANGUAGE-LIST(*LIST) = GERMAN
LANGUAGE-LIST(*LIST) = GREEK
LANGUAGE-LIST(*LIST) = ITALIAN
```

TRACE-PROCEDURE
Resume interrupted procedure in stages

Domain: **PROCEDURE**

Command description

You can specify the number of commands to be executed before the procedure is interrupted again. If the interrupt point lies in an area in which procedure interrupts are not permitted, the interrupt does not take place until interrupts are again allowed.

If logging is permitted in the procedure, it is activated for the commands executed after the TRACE-PROCEDURE command (regardless of the LOGGING operand in the CALL-PROCEDURE or MODIFY-PROCEDURE-TEST-OPTIONS command).

Format

TRACE-PROCEDURE	Alias: **TCP**
STEPS = ***LAST-INP**UT / <text 0..1800 with-low *arith-expr*>	

Operands

STEPS =
Determines the number of commands to be executed before the next interrupt.

STEPS = *LAST-INPUT
Specifies the number last defined. If there is no previous declaration, the value 1 is used, i.e. an interrupt takes place after each command.

STEPS = <text 0..1800 with-low *arith-expr*>
Integer expression; specifies the number of commands before the next interrupt.

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available

TRANSMIT-BY-STREAM
Transmit variables

Domain: **PROCEDURE**

Command description

The TRANSMIT-BY-STREAM command carries out a variable transmission initiated from the client side, to or from the server which is addressed and using the specified S variable stream.

TRANSMIT-BY-STREAM first sends the variable specified in VARIABLE-NAME to the SDF-P controller, and receives data back from SDF-P in the variables specified in RETURN-VARIABLE-NAME.

If *VARIABLE is assigned to the S variable stream in an ASSIGN-STREAM command, then the variables contained in the RETURN-VARIABLE-NAME and RET-CONTROL-VAR-NAME of the TRANSMIT-BY-STREAM command are overwritten by the variables which have the same name in the ASSIGN-STREAM command.

If *DUMMY is assigned to the S variable stream, the command sends back a return code indicating that nothing has been changed.

The execution sequence is described in chapter “S variable streams” on page 187.

Format

TRANSMIT-BY-STREAM
STREAM-NAME = <structured-name 1..20>
,**VAR**IABLE-**NAME** = ***NONE** / <composed-name 1..255>
,**RETURN-VAR**IABLE-**NAME** = ***SAME** / ***NONE** / <composed-name 1..255>
,**CONTROL-VAR-NAME** = ***NONE** / <composed-name 1..255>
,**RET-CONTROL-VAR-NAME** = ***SAME** / ***NONE** / <composed-name 1..255>

Operands

STREAM-NAME = <structured-name 1..20>
Name of the S variable stream in which the variable is transmitted.

VARIABLE-NAME =
The S variable sent by the transmission (the “output” variable).

VARIABLE-NAME = *NONE
No S variable is transmitted.
If the variable stream is assigned to a variable, the assigned S variable remains unchanged.

VARIABLE-NAME = <composed-name 1..255>
Name of the S variable, which is sent to the server.
The specified S variable must be a structure, containing all the data which the client must send. If an empty S variable is specified, and the variable stream is assigned to *VARIABLE(...), an empty list element is created.

RETURN-VARIABLE-NAME =
Name of the return variable for the transmission.

RETURN-VARIABLE-NAME = <u>*SAME</u>
The return variable is the S variable which was sent.
The transmitted S variable is first written to the server, after which the server overwrites it with the return data.
For example, if the S variable stream is assigned to *VARIABLE(...), then the transmitted variable is first written into the S variable (list) specified by the VARIABLE-NAME operand in the ASSIGN-STREAM command; this is then overwritten by the S variable (list) specified by RETURN-VARIABLE-NAME.

RETURN-VARIABLE-NAME = *NONE
The client does not expect the return of any return variable.
However, if the S variable stream is assigned to *VARIABLE(...), the specified return variable (list element) is nevertheless handled as a though a return variable had been specified in TRANSMIT-BYSTREAM.

RETURN-VARIABLE-NAME = <composed-name 1..255>
Name of the S variable which is sent back by the server after the TRANSMIT-BY-STREAM has been executed.
If RETURN-VARIABLE-NAME =VARIABLE-NAME, the actions performed are the same as for RETURN-VARIABLE-NAME = *SAME.
If the server does not send back a RETURN-VARIABLE, the variable remains unaltered.

CONTROL-VAR-NAME =
Specifies the control data for the server.
This data is identified by a standard header, the format and structure of which are detailed after the description of the command return codes.
The nature of this control data is determined by the possible servers. The control data is not part of the user data. Separate field names are reserved within the server for this process; e.g. the FHS subsystem (TPR display) defines variables which, among other things, define panels which must be loaded and in which FHS performs its functions.

CONTROL-VAR-NAME = *NONE
No control variable is specified.
If the server requires control variables, then either standard variables will be used (when possible) or the transmission will be rejected.

CONTROL-VAR-NAME = <composed-name 1..255>(...)
Name of the S variable which contains the control data.
The specified S variable must be a structure.

Note

If the server requests a control variable and the caller either specifies none or specifies an incomplete one, there are two possible results of the transmission:

1. If the missing control data is optional for the controlling server, the transmission will be carried out using the default settings.
2. If the missing control data is mandatory for the controlling server, the transmission will be terminated, with an error message.

RET-CONTROL-VAR-NAME =
Specifies the S variable by which the return control data can be sent to the client.

RET-CONTROL-VAR-NAME = *SAME
Specifies the same S variable as for CONTROL-VAR-NAME.
The same rules apply in this case as when VARIABLE-NAME and RETURN-VARIABLE-NAME are both specified.

RET-CONTROL-VAR-NAME = *NONE
The client expects no return control data.

RET-CONTROL-VAR-NAME = <composed-name 1..255>
Name of the S variable by which the server transmits the return control data.
The specified S variable must be a structure.
If the server does not send back a RET-CONTROL-VARIABLE, the variable remains unaltered.

Command return codes

(SC2)	SC1	Maincode	Meaning/Guaranteed messages
	0	CMD0001	No error
1	0	CMD0001	Variable stream is assigned to *DUMMY; no transmission, variable remains unchanged
2	0	SDP0512	Server is no longer active. Data stream is assigned to *DUMMY
2	0	SDP0531	Warning returned by server; process continuing
	1	CMD0202	Syntax error
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	CMD0216	Do not have required privilege
	64	SDP0091	Semantic error Guaranteed messages: SDP1008
	64	SDP0532	Server error; command rejected
	64	SDP0534	Internal server error; command terminated. Server link terminated following unexpected event or due to shortage or absence of system resources
	64	SDP0517	Variable stream does not exist
	64	SDP0522	Transmitted data is incompatible with the format that the server handles
	64	SDP1132	Variable name too long
	130	SDP0099	No further address space available

Note

The following should be noted for return code SDP0512: if S variable streams which are dependent on the calling procedure are assigned to procedure-local variables, then after the procedure end they can no longer be used. Any subsequent transmission is ignored, with a warning, and the S variable stream is reset to *DUMMY.

Example

```
/DECLARE-VARIABLE OPS-VAR(TYPE=*STRUCTURE),MULTIPLE-ELEMENTS=*LIST
/DECLARE-VARIABLE OPS-VAR1(TYPE=*STRUCTURE)
/ASSIGN-STREAM SYSINF,TO=*VARIABLE(OPS-VAR)
/ASSIGN-SYSOUT TO=#ERROR-SYSOUT
/TRANSMIT-BY-STREAM SYSINF, VARIABLE=OPS-VAR1, RETURN-VARIABLE=*NONE
```

Standard header

SDF-P provides a standard header as a reserved structure layout. This standard header can be defined as the first element of the control variable structure.

Its layout is declared in a procedure which must be called by an INCLUDE-PROCEDURE.

This procedure is supplied in the $TSOS.SYSPRC.SDF-P-BASYS.024 library, in the FHDR element. It looks as follows:

```
/set-proc-options "caller=include    not supported by sdf-p-basys"
/begin-parameter-declaration
/   declare-parameter -
/   "------------ std param --------------------------------*"-
            /(PREFIX       (init='SYSSDP') -
            /,INCLUDE-FORM (init='LAYOUT') "/INITIALIZE" -
            /,VARIABLE-NAME(init='') -
/   "------------ include specific param -------------------*"-
            /,UNIT         (init='') -
            /,FUNCTION     (init='') -
            /,VERSION      (init=0) -
            /,SUBCODE2     (init=0) -
            /,SUBCODE1     (init=0) -
            /,MAINCODE     (init='CMD0001') -
            /)
/end-parameter-declaration
/
/if (not is-sdf-p() )
/   exit-procedure error=*yes(subcode1=41,maincode=cmd2241)
/end-if
/
/if (upper-case(INCLUDE-FORM) == 'LAYOUT')
/
/begin-structure name=&PREFIX.IFID-MDL,scope=proc
/  declare-element -
              /(UNIT      (type=string) -
              /,FUNCTION  (type=string) -
              /,VERSION   (type=integer) -
              /)
/end-structure
/begin-structure name=&PREFIX.RETC-MDL,scope=proc
/  declare-element -
              /(SUBCODE2 (type=integer) -
              /,SUBCODE1 (type=integer) -
              /,MAINCODE (type=string)  -
              /)
/end-structure
/begin-structure name=&PREFIX.FHDR,scope=proc
/  declare-element INTERFACE-ID(type=structure(&PREFIX.IFID-MDL))
/  declare-element RETURNCODE  (type=structure(&PREFIX.RETC-MDL))
/end-structure
/
/else-if (upper-case(INCLUDE-FORM) == 'INITIALIZE')
/
/   if (VARIABLE-NAME == '')
/      write-text '%  mandatory parameter variable-name missing.'
/      raise-error
/   end-if
/   declare-variable PARAM(type=string)
/   for PARAM = ('UNIT','FUNCTION')
/      &VARIABLE-NAME..INTERFACE-ID.&PARAM = &PARAM
/   end-for
/   &VARIABLE-NAME..INTERFACE-ID.VERSION = INTEGER(VERSION)
/   for PARAM = ('SUBCODE2','SUBCODE1')
/      &VARIABLE-NAME..RETURNCODE.&PARAM = INTEGER(&PARAM)
/   end-for
/   &VARIABLE-NAME..RETURNCODE.MAINCODE = MAINCODE
/
/else
/  write-text '%  form=&INCLUDE-FORM not supported; include aborts'
/  raise-error
/end-if
/exit-procedure
```

Description of the procedure parameters

UNIT (TYPE = *STRING)
Name of the server which is defining the control variable. This name should be identical with that defined in ASSIGN-STREAM.

FUNCTION (TYPE = *STRING)
Name of the function for which the server is defining the control variable layout. This name is defined by the server and passed on.

VERSION (TYPE = *INTEGER)
Version of the control variable. This allows the server to support earlier versions of the control variable compatibly.

SUBCODE2 (TYPE = *INTEGER)
Subcode2, which is returned by the server (in RET-CONTROL-VAR-NAME).

SUBCODE1 (TYPE = *INTEGER)
Subcode1, which is returned by the server (in RET-CONTROL-VAR-NAME).

MAINCODE (TYPE = *STRING)
Message ID, which is returned by the server (in RET-CONTROL-VAR-NAME).

The same conventions apply for the SUBCODE2, SUBCODE1 and MAINCODE operands as for the command return code.

The routers (TRANSMIT-BY-STREAM and TRANSVV) do not use these variables. Warnings and error messages from TRANSMIT-BY-STREAM and TRANSVV depend on the internal return code, which is returned by the server and not by means of these control variables.

Consequently, these control variables should be returned by the server, to provide a more precise error message than those from TRANSMIT-BY-STREAM and TRANSVV.

Note

The error and warning information which is written into these control variables should agree with the server's internal return code. For example, this means that if TRANSMIT-BY-STREAM returns the error class SUBCODE2=0,SUBCODE1=64 and MAINCODE=SDP0532, there will also be an error code in the return control variable; or, if SUBCODE2=2,SUBCODE1=0 and MAINCODE=SDP0531 is returned, then the return control variable should (also) contain a warning, and so on. However, the maintenance of this consistency is the responsibility of the server.

UNTIL
Terminate REPEAT block

Domain: **PROCEDURE**

Command description

UNTIL contains the loop condition for a REPEAT block, i.e. for a REPEAT loop, and terminates the REPEAT block (the initiation command for the REPEAT block is REPEAT). If the condition is not satisfied, a new loop pass is started with the first command in the REPEAT block. Otherwise, the loop is terminated and procedure execution resumes with the first command following the UNTIL command. (See also section “REPEAT block” on page 99).

Expressions are replaced in the condition each time the loop is executed.

Format

UNTIL
CONDITION = <text 0..1800 with-low *bool-expr*>

Operands

CONDITION = <text 0..1800 with-low *bool-expr*>
Logical expression as the condition for terminating the REPEAT block (see chapter “Expressions” on page 249 for information on logical expressions).

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	1	SDP0139	Back branch limit reached
	1	SDP0223	Incorrect environment
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available

Example

See page 99.

WHILE
Initiate WHILE block

Domain: **PROCEDURE**

Command description

WHILE initiates a WHILE block, i.e. a WHILE loop: execution of the command sequence within the WHILE block is repeated as long as the condition specified in the WHILE command is met. If the condition is not met, the loop is terminated and procedure execution resumes with the command following the terminating END-WHILE command. (See the remarks on control structures in section “WHILE block” on page 98 for more information.)

Expressions are replaced in the operand only when entering the WHILE loop, but not each time the loop is executed.

Format

WHILE
CONDITION = <text 0..1800 with-low *bool-expr*>

Operands

CONDITION = <text 0..1800 with-low *bool-expr*>
Logical expression as the condition for re-executing the commands in the WHILE loop (see chapter “Expressions” on page 249 for information on logical expressions).

Command return codes

(SC2)	SC1	Maincode	Meaning
	0	CMD0001	No error
	1	CMD0202	Syntax error
	1	SDP0118	Command in false context
	1	SDP0223	Incorrect environment
	3	CMD2203	Incorrect syntax file
	32	CMD0221	System error (internal error)
	64	SDP0091	Semantic error
	130	SDP0099	No further address space available

Example

```
/ "Reduce list variable LIST-A to last 250 elements"
/WHILE (SIZE ('LIST-A') > 250)
/  FREE-VARIABLE LISTE-A#
/END-WHILE
```

16 Installation and configuration

The following pages give a description of how to install SDF-P and of the required software configuration.

16.1 Installing SDF-P

SDF-P is delivered as a separate delivery unit, SDF-P-BASYS is part of the basic configuration. Installation is performed using the IMON installation monitor (see the “IMON” manual [12]).

The SDFPBASY (for SDF-P-BASYS) and SDF-P subsystems are loaded automatically during startup.

16.1.1 Installation files for SDF-P

The following files are required for installing SDF-P:

File name	**Contents**
SPMLNK.SDF-P.024	Autonomous part of SDF-P V2.4A loaded automatically at startup (SPARC systems)
SRMLNK.SDF-P.024	Autonomous part of SDF-P V2.4A loaded automatically at startup (RISC systems)
SYSFGM.SDF-P.024.D	Release Notices (German)
SYSFGM.SDF-P.024.E	Release Notices (English)
SYSLNK.SDF-P.024	Autonomous part of SDF-P V2.4A loaded automatically at startup (/390 systems)
SYSPRC.SDF-P.024	Procedures for SDF-P V2.4A (e.g. FHDR)
SYSRMS.SDF-P.024	RMS delivery sets for SDF-P V2.4A
SYSSDF.SDF-P.024	Syntax file containing only the COMPILE-PROCEDURE command which can only be executed with SDF-P itself
SYSSII.SDF-P.024	IMON file containing structure and installation information about Release Units and Release Items of SDF-P V2.4A
SYSSSC.SDF-P.024	SSCM catalog defining the SDF-P subsystem

IMON installation information for SDF-P

Logical IMON name	**Default path name**
SYSFGM.D	$TSOS.SYSFGM.SDF-P.024.D
SYSFGM.E	$TSOS.SYSFGM.SDF-P.024.E
SYSLNK	$TSOS.SYSLNK.SDF-P.024
SYSSSC	$TSOS.SYSSSC.SDF-P.024
SYSSDF	$TSOS.SYSSDF.SDF-P.024
SYSREP	$TSOS.SYSREP.SDF-P.024
SYSRMS	$TSOS.SYSRMS.SDF-P.024
SYSPRC	$TSOS.SYSPRC.SDF-P.024
SYSSII	$TSOS.SYSSII.SDF-P.024

16.1.2 Installation files for SDF-P-BASYS

The following files are required for installing SDF-P-BASYS:

File name	**Contents**
SIPLIB.SDF-P-BASYS.024	Library containing privileged macros
SPMLNK.SDF-P-BASYS.024	Autonomous part of SDF-P-BASYS V2.4A loaded automatically at startup (SPARC systems)
SRMLNK.SDF-P-BASYS.024	Autonomous part of SDF-P-BASYS V2.4A loaded automatically at startup (RISC systems)
SYSLIB.SDF-P-BASYS.024	SDF-P-BASYS Assembler macros
SYSLNK.SDF-P-BASYS.024	Autonomous part of SDF-P-BASYS V2.4A loaded automatically at startup (/390 systems)
SYSMES.SDF-P-BASYS.024	Message file with messages from SDF-P-BASYS V2.4A
SYSPRC.SDF-P-BASYS.024	Procedures for SDF-P V2.4A (e.g. FHDR)
SYSRMS.SDF-P-BASYS.024	RMS delivery sets for SDF-P-BASYS V2.4A
SYSSDF.SDF-P-BASYS.024	Syntax file containing all commands executed by SDF-P itself (either directly or within a compiled procedure)
SYSSII.SDF-P-BASYS.024	IMON file containing structure and installation information about Release Units and Release Items of SDF-P-BASYS V2.4A
SYSSSC.SDF-P-BASYS.024	SSCM catalog defining the SDFPBASY subsystem

IMON installation information for SDF-P-BASYS

Logical IMON name	Default path name
SIPLIB	$TSOS.SIPLIB.SDF-P-BASYS.024
SYSLIB	$TSOS.SYSLIB.SDF-P-BASYS.024
SYSLNK	$TSOS.SYSLNK.SDF-P-BASYS.024
SYSMES	$TSOS.SYSMES.SDF-P-BASYS.024
SYSPRC	$TSOS.SYSPRC.SDF-P-BASYS.024
SYSSSC	$TSOS.SYSSSC.SDF-P-BASYS.024
SYSSDF	$TSOS.SYSSDF.SDF-P-BASYS.024
SYSREP	$TSOS.SYSREP.SDF-P-BASYS.024
SYSRMS	$TSOS.SYSRMS.SDF-P-BASYS.024
SYSPRC	$TSOS.SYSPRC.SDF-P-BASYS.024
SYSSII	$TSOS.SYSSII.SDF-P-BASYS.024

16.2 Software configuration

SDF-P V2.4A requires the following subsystems to run:

BS2000/OSD-BC ≥ V5.0
SDF-P-BASYS V2.4A
SDF ≥ V4.5A
VAS ≥ V2.0A

The following are required for special functions:

PLAM / ILAM V3.1A
SDF-P-BIF ≥ V1.0B
JV ≥ V13.0D

17 Messages

As in the message file, the messages in this chapter are sorted according to their message codes, with letters ranging before numbers.

The system messages are listed below. For guaranteed messages the message attribute “Warranty” (see the manual “System Messages” [15]) is documented by “ ◆ Warranty: Y” in the line after the message text.

List of messages

SDPF001 `INTERNAL ERROR IN MODULE '(&00)', INTERNAL ERROR NUMBER '(&01)'; ADDITIONAL INFORMATION: '(&02)'. CONTACT THE SYSTEM ADMINISTRATOR`

Meaning
The inserts are meant to help the systemdiagnosis.
An entry has been written into the SERSLOG file.

Response
If present, please send the following data to the systemdiagnosis:
- error message with inserts or serslog file,
- systemdump,
- procedure, that caused the error.

SDP0001 `INCONSISTENCY BETWEEN SDF-P SYNTAX FILE AND EXECUTION MODULE`

Meaning
Possible reasons:
- an attribute of the command has been changed in the user syntax file.
- the command is overruled by a command defined in the user syntax file.
- a syntax file of a previous SDF-P or SDF-P-BASYS version is installed.

Response
Remove the command from the user syntax file or contact the system administrator.

SDP0002 `IRREGULAR PROCEDURE TERMINATION. CONTACT THE SYSTEM ADMINISTRATOR`

Meaning
Unexpected procedure termination, caused by a system error during procedure execution.

```
SDP0003     SYNTAX ERROR IN COMMAND
```

```
SDP0004     ERROR DETECTED AT COMMAND LINE: (&00) IN PROCEDURE '(&01)'
```

Meaning
An error has been detected for the command at the specified line or has been reported by the abnormal termination of a program at a previous line.

```
SDP0005     TOO MANY PARAMETERS SPECIFIED
```

Meaning
Possible reasons:
- Fewer parameters are declared in the procedure than specified when calling the procedure.
- The BUILTIN function expects fewer parameters.

```
SDP0006     POSITIONAL PARAMETERS ARE NOT ALLOWED FROM PARAMETER NO. (&00)
```

Meaning
Possible reasons:
- Positional parameters are not allowed after keyword parameters.
- The maximum number of positional parameters has been reached.

```
SDP0007     SYNTAX ERROR IN PARAMETER LIST, PARAMETER NO. (&00)
```

```
SDP0008     INVALID KEYWORD '(&00)' SPECIFIED IN PARAMETER LIST
```

Meaning
1. The keyword is neither a valid parameter name nor an
   unambiguous abbreviation of a parameter name.
2. The keyword is neither a valid constant parameter value
   nor an unambiguous abbreviation of a constant parameter
   value.

```
SDP0009     PARAMETER '(&00)' MUST BE SPECIFIED AT PROCEDURE CALL
```

```
SDP0010     TYPE OF PARAMETER '(&00)' INVALID
```

Meaning
Procedure: actual parameter-type is not convertible to the type of the formal parameter.
Builtin function: the type of the given expression does not correspond to
the type of the parameter or only keywords are allowed at this position.
FOR loop: FROM, TO, INCREMENT must be arithmetic expressions.
CONDITION must be boolean expression.

SDP0011 `ERROR IN OPERAND '(&00)'`

SDP0012 `VALUE OF EXPRESSION LESS THAN ZERO`

SDP0013 `FILE NAME INVALID OR MISSING`

SDP0014 `WARNING IN LINE: (&00) IN PROCEDURE '(&01)'`

Meaning
The previous message belongs to the line with the given line number or one line of its continuation lines.

SDP0015 `OPERAND VALUE '(&00)' NOT ALLOWED; DEFAULT VALUE ASSUMED`

SDP0016 `AGGREGATES NOT ALLOWED`

SDP0017 `NO STRUCTURED PROCEDURE`

Meaning
Possible reasons:
- the procedure starts with /BEGIN-PROC,
- the leading slash is missing.

SDP0018 `ERROR RAISED BY USER`

Meaning
The error was explicitly given by means of the command /RAISE-ERROR or /EXIT-PROCEDURE.

SDP0019 `THE FILE REFERENCED IN THE COMMAND CANNOT BE PROCESSED IN RFA MODE`

SDP0020 `SYNTAX ERROR IN FILE NAME OR NAME OF LIBRARY ELEMENT`

SDP0021 `THE STRUCTURE OF THE FILE '(&00)' DOES NOT MATCH THE REQUIRED ACCESS METHOD`

SDP0022 `COMMAND OR OPERAND VALUE '(&00)' NOT ALLOWED IN SDF-P BASIC-VERSION`

Meaning
The full functionality of SDF-P is not available or the needed version of SDF-P is not available.

Response
Please, contact system administrator to start subsystem SDF-P or upgrade to a newer version of SDF-P.

SDP0023 `PROCEDURE CANCELLED ON USER REQUEST`

Meaning
The procedure is cancelled after acknowledgement by user of K2 key pressed while a procedure parameter was requested at the terminal.

Response
If necessary, the procedure can be called again with correct parameter input.

SDP0024 `Error detected at command line: (&00) in current dialog block`

Meaning
An error has been detected for the command at the specified line or has been reported by the abnormal termination of a program at a previous line.

SDP0025 `Warning in line: (&00) in current dialog block`

Meaning
The previous message belongs to the line with the given line number or one of its continuation lines.

SDP0026 `Task specific default not allowed for this command`

Meaning
Task specific defaults (introduced by the character !) are not allowed for SDF-P control flow commands:
- BEGIN-BLOCK
- BEGIN-PARAMETER-DECLARATION
- CYCLE
- ELSE, ELSE-IF
- END-BLOCK, END-FOR, END-IF, END-WHILE
- END-PARAMETER-DECLARATION
- EXIT-BLOCK
- FOR
- GOTO
- IF, IF-BLOCK-ERROR, IF-CMD-ERROR
- REPEAT
- UNTIL
- WHILE

and for the commands
- ADD-CJC-ACTION
- END-CJC-ACTION
- DECLARE-PARAMETER
- OPEN-VARIABLE-CONTAINER when specified in a parameter declaration block
  (i.e.: between BEGIN-PARAMETER-DECLARATION and
  END-PARAMETER-DECLARATION)
- SET-PROCEDURE-OPTIONS

SDP0030 `SDF-P version not compatible with SDF-P-BASYS version. SDF-P start rejected`

SDP0039 `MORE THAN ONE VALUE SPECIFIED FOR OPERAND (&00). LAST VALUE IS USED`

SDP0040 `/FOR command overflows limit defined by system`

Meaning
Maximum length permitted for this command: 4000 characters.

SDP0089 `INPUT ERROR`

Meaning
The input is missing or does not match to the required type or format.

SDP0090 `WARNING: AN ACTION WAS PERFORMED`

- Warranty: Y

SDP0091 `SEMANTIC ERROR`

- Warranty: Y

SDP0092 `RESOURCE CURRENTLY NOT AVAILABLE`

- Warranty: Y

SDP0093 `ERROR DURING ACCESS OF FILE/LIBRARY '(&00)', ERROR '(&01)'. MORE INFORMATION: /HELP-MSG (&01)`

SDP0094 `CONTAINER NOT ACCESSIBLE`

Meaning
File, library element or variable is not accessible.

SDP0095 `UNEXPECTED ERROR RETURN CODE '(&00)' FROM COMPONENT '(&01)'. CONTACT SYSTEM ADMINISTRATOR`

SDP0096 `RECORD TOO LONG`

Meaning
Record whose size exceeds maximum size encountered during read/write operation.

SDP0097 `FILE, LIBRARY OR LIBRARY ELEMENT '(&00)' IS LOCKED`

SDP0098 `FILE, LIBRARY OR LIBRARY ELEMENT '(&00)' DOES NOT EXIST`

SDP0099 `NO MORE VIRTUAL MEMORY SPACE AVAILABLE AT THIS MOMENT`

◆ Warranty: Y

Meaning
The limit of the virtual memory space has been reached.
- the declared variables need too much memory space;
- the nest level of procedure calls is too deep;
- the sum of the length of all the variable values of the task exceeds
  the ADDRESS-SPACE-LIMIT (in megabytes) attribute of the USER-ID:
  - boolean value = 1 byte
  - integer value = 4 bytes
  - string value= length of string
  - variable declaration without value = 0 byte

If the error occurs during creation of a structure, then the structure is
not completely declared afterwards.

Response
- release memory by means of /FREE-VARIABLE on variables
- reduce the procedure level by means of /CANCEL-PROCEDURE
- cancel the task by means of /LOGOFF.

SDP0100 `Specified file '(&00)' is not a SAM/ISAM file`

SDP0101 `Operation name 'DECLARE-PARAMETER' unknown. Contact the system administrator`

Meaning
DECLARE-PARAMETER command not recognized by SDF.
SDF-P-BASYS syntax file not installed.

SDP0102 `CONTINUATION LINE MISSING, 'EOF' ASSUMED`

SDP0103 `BUFFER TOO SMALL FOR CONTINUATION LINES`

Meaning
Maximum length: 16364 characters.

SDP0104 `SLASHES MISSING`

Meaning
Commands must begin with one slash, statements with two slashes;
continuation lines are subject to the same rule.

SDP0108 `SLASH MISSING`

SDP0109 `SLASHES PROHIBITED`

SDP0110 `COMMAND FOUND, STATEMENT EXPECTED`

SDP0111 THE COMMAND COULD NOT BE IDENTIFIED AS /DECLARE-PARAMETER NOR AS /OPEN-VARIABLE-CONTAINER COMMAND

Meaning
In the procedure head a /DECLARE-PARAMETER or a /OPEN-VARIABLE-CONTAINER command is expected, but was not identified.
Possible reasons:
- spelling error on the operation name.
- an ambiguous abbreviation was used.
- the operation was interpreted as /SET-VARIABLE command because an equality character was written after the operation name.

SDP0112 SYNTAX ERROR IN SKIP LABEL

SDP0113 ERROR IN COMMAND NAME

SDP0114 ODD NUMBER OF APOSTROPHES

Meaning
there are too many or not enough apostrophes.

SDP0115 DOUBLE APOSTROPHE MISSING

SDP0116 PARENTHESIS MISSING

SDP0117 UNEXPECTED PARENTHESIS

SDP0118 COMMAND INVALID IN CURRENT ENVIRONMENT '(&00)'

Meaning
- there were more parameters specified by /CALL-command than have been declared in the procedure.
- /DECLARE-PARAMETER not found, because of a spelling error in the command name, so no parameter was expected.
- /BEGIN-PARAMETER-DECLARATION not found, because another BS2000 command was found before, so the declarations of the parameters could not be found.
- /BEGIN-PARAMETER-DECLARATION and /END-PARAMETER-DECLARATION are missing or erroneous, so the declarations of the parameters could not be found.
- a procedure-end command was found, but no procedure was active.
- a block-end command was found, but no block-begin command was given before.
- a control flow command was generated by escape-character replacement.
- the given command is not allowed in interrupt-state.
- no structure or layout initiated.
- layout declaration currently not allowed.
- the element name must not be of SDF data type composed-name.

`SDP0119` `INVALID COMMAND THAT SHOULD BE EXECUTED BY /EXEC-CMD`

`SDP0130` `SYNTAX ERROR IN EXPRESSION`

`SDP0131` `RECORD TOO LONG`

Meaning
Possible reasons:
- The expression or the value of the expression is too long.
- The record became too long after the escape character replacement.
Maximum length: 16364 Bytes.

`SDP0132` `NO PROCEDURE BODY EXISTS`

`SDP0133` `INVALID TYPE OF EXPRESSION`

`SDP0134` `DATA FOUND, COMMAND EXPECTED`

Meaning
The record does not start with a slash, so the command cannot be recognized.
A /SEND-DATA command was executed, although no data was expected.

`SDP0135` `DATA FOUND, STATEMENT EXPECTED`

Meaning
The record does not start with two slashes, so the statement cannot be recognized.
A /SEND-DATA command was executed although a statement was expected.

`SDP0136` `STATEMENT FOUND, COMMAND EXPECTED`

Meaning
The record starts with two slashes, so the command cannot be recognized.
A /SEND-STMT command was executed, although a command was expected.

`SDP0137` `STATEMENT FOUND, DATA EXPECTED`

Meaning
The record starts with two slashes, so no data was recognized.
A /SEND-STMT command was executed, although data was expected.

`SDP0138` `ERROR DURING PREANALYSIS OF THE PROCEDURE`

Meaning
The structure of the procedure generated by the control flow commands is not correct.
The procedure was not executed.

Response
Change the structure of the procedure.

SDP0139 `BACK BRANCH LIMIT REACHED`

Meaning
The BACK-BRANCH-LIMIT specified by means of the
/MODIFY-PROCEDURE-TEST-OPTIONS command has been reached.

SDP0140 `SYNTAX ERROR DURING ESCAPE CHARACTER REPLACEMENT`

- Warranty: Y

SDP0141 `SEMANTIC ERROR DURING ESCAPE-CHARACTER REPLACEMENT`

SDP0142 `LABEL IGNORED IN PROCEDURE HEAD`

SDP0143 `/CYCLE ONLY ALLOWED IN LOOPS`

SDP0144 `ERROR DURING PARAMETER TRANSFER`

Meaning
If the error could not be identified by previous error messages,
the following reason is possible:
the parameter declarations were not identified by /BEGIN-PARAMETER-DECLARATION and /END-PARAMETER-DECLARATION, so only the first declaration is evaluated.

SDP0145 `SPECIFICATION OF LIBRARY-ELEMENT VERSION NOT ALLOWED FOR NOT-STRUCTURED PROCEDURES`

Meaning
For Not-Structured procedures, only the default version value (*HIGHEST-EXISTING) is possible.

SDP0150 `INCORRECT SYMBOL AT THIS POSITION IN EXPRESSION : (&00)`

SDP0151 `INCORRECT SYMBOL IN HEX-STRING: (&00).`

Meaning
Only digits or letters A-F/a-f are allowed.

SDP0152 `INCORRECT SYMBOL FOLLOWING INTEGER: (&00)`

Meaning
There must be a blank between an integer and a letter.

SDP0153 `ERROR IN NAME: (&00)`

Meaning
Name part must not start with a dot.
Dot must not follow a "-".

SDP0154 `ERROR IN NAME PART: (&00).`

Meaning
A dot must be followed by a letter or a special sign.

SDP0155 `STACK OVERFLOW DURING EXPRESSION ANALYSIS`

SDP0156 `SYNTACTICAL ERROR IN EXPRESSION: (&00)`

SDP0157 `OPERATOR NOT ALLOWED IN SDF-P BASIC-VERSION`

SDP0158 `ERROR: EXPRESSION EMPTY (LENGTH = 0).`

SDP0200 `LABEL '(&00)' OCCURS MORE THAN ONCE`

SDP0201 `INVALID BLOCK-CLOSING COMMAND USED`

Meaning
The current open block must not be closed with this block end command.

SDP0202 `BLOCK-CLOSING COMMAND EXISTS, BLOCK-OPENING COMMAND MISSING`

Meaning
A block end command was found, although there is no block-opening command.

SDP0203 `CURRENT BLOCK NOT AN IF BLOCK`

Meaning
An /ELSE or /ELSE-IF command was found outside of an IF block.

SDP0204 `/ELSE COMMAND ALREADY PRESENT`

Meaning
The /ELSE command of the IF block was already found.
Another /ELSE or /ELSE-IF command is not allowed.

SDP0205 `SKIP LABEL NOT ALLOWED`

Meaning
Skip labels must not be used:
- inside of blocks
- before SDF-P control flow commands.

SDP0207 `Not all control structures closed`

Meaning
An END-BLOCK, END-IF, END-FOR, END-WHILE or UNTIL command is missing.

SDP0208 `TARGET LABEL '(&00)' NOT DEFINED`

SDP0209 `INVALID TARGET OF JUMP`

Meaning
Reasons:
- the target is not in current block hierarchy.
- the operand of a /CYCLE- or /EXIT-BLOCK-command is not the name of a block.
- the target of a /INCLUDE-BLOCK-command is not a /BEGIN-BLOCK-command with label.

SDP0210 /CYCLE OR /EXIT COMMAND NOT WITHIN A BLOCK

Meaning
No block-beginning command is specified for a /CYCLE or /EXIT command.

SDP0211 Only messages of the class SDP may be suppressed

SDP0212 /LOGON COMMAND DETECTED IN OR BEFORE PROCEDURE HEAD

Meaning
A /LOGON command is not allowed at this position.

SDP0213 COMMAND NOT YET IMPLEMENTED

SDP0215 LABEL OF BLOCK-CLOSING COMMAND DOES NOT MATCH LABEL OF BLOCK-OPENING COMMAND

SDP0216 THIS PROCEDURE CANNOT BE CALLED WITH THIS COMMAND

Meaning
/CALL command prohibited by the /SET-PROCEDURE-OPTIONS command.
Procedure was called with a /DO command, but has parameters with 'TRANSFER-TYPE=BY-REFERENCE'. Old procedures must not be called with /INCLUDE-PROC.

Response
If the calling command was /CALL-PROCEDURE, try again with /INCLUDE-PROCEDURE.
If the calling command was /INCLUDE-PROCEDURE, try again with /CALL-PROCEDURE.

SDP0217 THIS COMMAND IS NOT ALLOWED WITH THE OPERAND 'STRUCTURE-OUTPUT=VARNAME'

Meaning
Only the commands /SHOW-FILE-ATTRIBUTES with INFORMATION= NAME-AND-SPACE/ ALL-ATTRIBUTES and /SHOW-JOB-STATUS are allowed together with a structured variable.

SDP0218 VARIABLE MUST BE A LIST OF TYPE 'STRUCTURE'

SDP0219 ERROR DURING PROMPTING

Meaning
Error reported by TIAM during WRTRD operation.
Possible error conditions :
- prompting in batch mode,
- prompt string too long, ...

SDP0221 TOO MUCH INFORMATION. /EXEC-CMD NOT PROCESSED

SDP0222 OPERAND 'CMD' INVALID IN /EXEC-CMD, ERROR '(&00)'. IN SYSTEM MODE: /HELP-MSG (&00)

Meaning
For more detailed information about the error code enter /HELP-MSG in system mode.

SDP0223 `MODIFICATION OF CONTROL FLOW COMMANDS BY 'SYSTEM-EXIT' NOT ALLOWED`

Meaning
A SYSTEM-EXIT has modified a command that must be modified. This is especially not allowed for all SDF-P control flow commands.

Response
Contact the system administrator to correct or deactivate the SYSTEM-EXIT.

SDP0224 `LOGGING SUPPRESSED; CONTAINER '(&00)' IS READ PROTECTED`

SDP0225 `DO YOU WANT TO CANCEL ALL ACTIVATED PROCEDURES? REPLY (Y=YES; N=NO)`

SDP0226 `COMMAND NOT ALLOWED FOR /EXECUTE-CMD`

SDP0227 `WARNING WHILE EXECUTING THE OPERAND 'CMD'`

SDP0228 `TOO MANY VARIABLES ARE DECLARED. COMMAND IS ABORTED`

SDP0229 `ALL OPERANDS OF /EXECUTE-CMD IGNORED FOR COMMAND SPECIFIED BY 'CMD'. PROCESSING CONTINUES`

Meaning
The command specified by the operand 'CMD' cannot be processed by /execute-cmd; therefore all operands of /execute-cmd are ignored and the command is executed afterwards.

Response
For such commands, please use procedure-wide features:
* text-output-> /assign-sysout to=*variable(..)
* struct-output-> /assign-stream sysinf,to=*var(...)
* msg-struct-output -> /assign-stream sysmsg,to=*var(...)
* returncode-> /save-returncode (input after the cmd).
* returncode=*none-> /if-cmd-error ; end-if(after the cmd)
and input the command directly.

SDP0230 `Output larger than supported by /EXECUTE-CMD. /EXECUTE-CMD repeated; processing continues`

Meaning
The output of the command executed by /EXECUTE-CMD is larger than the default size foreseen by /EXECUTE-CMD (392.6 KBytes).
/EXECUTE-CMD repeats the command with a larger system buffer as long as the output of the command overflows the specified size.

Response
The amount of data output by the commands can be reduced by a more acute specification of the INFORMATION and SELECT operands of the executed command, if this command offers such capabilities.
If the repetition of the command must be avoided by all means, /EXECUTE-CMD must be replaced by the corresponding combination of /ASSIGN-SYSOUT, /ASSIGN-STREAM and /IF-CMD-ERROR commands.

```
SDP0231     OUTPUT OF SPECIFIED CMD LARGER THAN ACTUALLY SUPPORTED BY SYSTEM. CMD REJECTED
```

Meaning
The output of the specified command is greater than the maximal size foreseen by the command /EXECUTE-CMD (6.3 MBytes).
Therefore the output of the command cannot be completely stored in the system buffer reserved by /EXECUTE-CMD.
The command has been executed, but its output cannot be completely processed by /EXECUTE-CMD. So the command has been rejected.

Response
If the command specified in /EXECUTE-CMD offers operands to tune the output like "INFORMATION" or "SELECT", these operands should be used to reduce the amount of output data.
If this is not possible, the command /EXECUTE-CMD should be replaced by the corresponding combination of the commands /ASSIGN-SYSOUT, /ASSIGN-STREAM and /IF-CMD-ERROR.

```
SDP0232     ERROR IN OPERAND '(&00)'

SDP0234     OPERAND 'NAME' INVALID

SDP0237     OPERAND 'OUTPUT' INVALID

SDP0239     ERROR DURING EVALUATION OF RIGHT SIDE OF ASSIGNMENT
```

SDP0250 `Cmd or event not allowed in /INCLUDE-CMD context`

Meaning
The command or the event is not allowed during the execution of a procedure hierarchy called by /INCLUDE-CMD.
For example: /RESUME-PROGRAM, /SEND-MESSAGE, AID command, /START-PROGRAM, /CANCEL-PROCEDURE, ... lead to inconsistent situations which must be rejected.
/INCLUDE-CMD commands cannot be embedded.

Response
Suppress the command or the event from the input or do not call the procedure which activates it.

SDP0251 `Cmd or event not allowed in operand 'CMD' of /INCLUDE-CMD`

Meaning
The command or the event is not allowed in the operand 'CMD' of the command /INCLUDE-CMD.

Response
The command or event must be removed from the list of commands specified in the command /INCLUDE-CMD.

SDP0252 `Operand 'CMD' invalid`

Meaning
The specification of CMD value must be enclosed by parentheses and contain at least one character.

Response
Correct operand value and retry.

SDP0253 `/INCLUDE-CMD not allowed in this mode`

Meaning
/INCLUDE-CMD can only be specified from a program mode via CMD macro or //EXECUTE-SYSTEM-CMD statement.

Response
Use the command according to MEANING hints.

SDP0254 `All procedures cancelled due to program termination; processing continues`

Meaning
During the execution of procedures called by /INCLUDE-CMD, the program which issued this command has been terminated.
To avoid any discrepancy due to caller termination, all the procedures are cancelled.

Response
Identify the cause of the termination of the program to avoid this event.

SDP0255 `/RESUME-PROGRAM replaced by /RESUME-PROCEDURE; processing continues`

Meaning
After the interruption of a procedure called by /INCLUDE-CMD, not the caller program must be resumed but the interrupted procedure must be.

Response
Be quiet, system is running well.

SDP0256 `K2 request ignored in /INCLUDE-CMD context`

Meaning
The prompting of procedure parameters cannot be cancelled by K2 because /CANCEL-PROCEDURE is not allowed during execution of procedures called by /INCLUDE-CMD.

Response
Specify an empty value to provoke an error which leads to exit from procedure with error.

SDP0257 `Error in /INCLUDE-CMD cmd`

Meaning
The execution of a command specified by /INCLUDE-CMD reports an error.

Response
Identify the error cause and/or add an error handling block to intercept the error situation (/IF-CMD-ERROR).

SDP0258 `Warning in /INCLUDE-CMD cmd`

Meaning
A warning has been reported by a command specified by the command /INCLUDE-CMD.

Response
Identify the origin of the warning and correct it if necessary.

SDP0259 `Operation aborted: selection ignored`

Meaning
The CANCEL key was pressed in the dialog screen.
Selections are ignored, no element is returned.

SDP0260 `OPERAND MESSAGE IGNORED. PROCESSING CONTINUES`

Meaning
The operand MESSAGE cannot be processed by the present SDF version.

Response
Use a newer SDF version that can handle this operand.

SDP0300 `OPERAND(S) IN EXPRESSION DO NOT MATCH OPERATOR`

Meaning
The type of one or more operands does not match the operator.

SDP0301 INVALID VALUE IN EXPRESSION

SDP0302 ILLEGAL DIVISION BY ZERO

SDP0303 ERROR IN VARIABLE NAME '(&00)'

SDP0304 OVERFLOW, NUMBER OUT OF RANGE

Meaning
Only values between -2**31 and 2**31-1 maybe used.
Builtin function USER-SWITCH: there are only 32 switches (numbers 0-31), so the requested index is too big or too small.

SDP0305 VALUE WAS TRUNCATED BECAUSE BUFFER TOO SMALL

Meaning
The internally defined buffer is too small.

SDP0306 BUFFER TOO SMALL. OPERATION OR ASSIGNMENT NOT EXECUTED

SDP0307 Error in variable '(&00)'

Meaning
An error was detected when SDF-P processed the specified variable.

Response
Correct the variable and/or the related operations.

SDP0310 EVALUATION ERROR

SDP0373 DESCRIPTION OF SYSTEM INTERFACE MODIFIED IN SYNTAX FILE

Meaning
Possible reasons:
- an attribute of the command has been changed in the user syntax file.
- the command is overruled by a command defined in the user syntax file.

SDP0402 INVALID FUNCTION NAME. FUNCTION '(&00)' DOES NOT EXIST

SDP0403 Specified string is too long (more than 4 characters)

SDP0410 INCONSISTENCY BETWEEN 'WHEN' AND 'THEN' PARAMETERS

SDP0411 STRING EMPTY

SDP0412 START POSITION OUT OF RANGE

Meaning
The start position is greater than the length of the string or <= zero.

SDP0413	`ILLEGAL LENGTH`
SDP0414	`Warning: *REST-LENGTH value used for LENGTH operand`
SDP0415	`SYNTAX ERROR: INTEGER EXPECTED IN STRING. CONVERSION NOT POSSIBLE`
SDP0416	`NUMBER OUT OF RANGE`

Meaning
INTEGER-TO-CHARACTER() :
- only values between 0 and 255 may be used.
INTEGER() :
- only values between -2**31 and 2**31-1 may be used.

SDP0417	`SPECIFIED STRING EMPTY. FUNCTION NOT EXECUTED`
SDP0418	`INVALID MSG-IDENTIFICATION`
SDP0419	`JV: JOB VARIABLE '(&00)' NOT ACCESSIBLE`

Meaning
Possible reasons:
- The job variable is not shareable.
- The job variable is access protected.

SDP0420	`JV: JOB VARIABLE '(&00)' DOES NOT EXIST`
SDP0421	`JV: DMS ERROR '(&00)' WHILE ACCESSING JOB VARIABLE. IN SYSTEM MODE: /HELP-MSG DMS(&00)`

Meaning
For more detailed information about the DMS error code enter /HELP-MSG in system mode or see the BS2000 manual 'System Messages'.

SDP0422	`KEYWORD '(&00)' UNKNOWN FOR THIS FUNCTION`
SDP0423	`VARIABLE '(&00)' NOT AN ARRAY`
SDP0424	`NO CONTAINER ASSIGNED TO VARIABLE '(&00)'`
SDP0425	`BUILTIN FUNCTION VAR-ATTRIBUTES: VARIABLE NOT A STRUCTURE`

Meaning
The variable is no structure and so the information is not available (2nd Parameter *STRUCTURE-INFORMATION).

SDP0426	`VARIABLE '(&00)' NOT A LIST`
SDP0427	`ATTRIBUTE '(&00)' UNKNOWN`
SDP0428	`COMMAND RETURN CODE NOT AVAILABLE IN DIALOG`
SDP0429	`CONVERSION NOT POSSIBLE`
SDP0430	`VAR-ATTRIBUTE: CONTAINER HAS NO SCOPE`

SDP0431	`ERROR '(&00)' IN BUILTIN FUNCTION '(&01)'`
SDP0432	`FUNCTION NOT ALLOWED FOR LISTS`
SDP0433	`GIVEN STRING NOT A C-LITERAL`
SDP0434	`GIVEN STRING NOT A X-LITERAL`
SDP0435	`DESIRED INFORMATION NOT AVAILABLE`
SDP0436	`GIVEN LENGTH NOT BETWEEN ZERO AND MAXIMUM POSSIBLE STRING LENGTH`
SDP0437	`LENGTH OF PARAMETER 'FILL-BYTE' EQUAL TO ZERO`
SDP0438	`LENGTH OF PARAMETER 'SEPARATOR' EQUAL TO ZERO`
SDP0439	`LENGTH OF FILE NAME ZERO OR GREATER THAN 54`
SDP0440	`Name '(&00)' not a file name or not a specific file name`
SDP0441	`DMS ERROR '(&00)' WHEN CALLING FSTAT MACRO. IN SYSTEM MODE: /HELP-MSG (&00)`

Meaning
For more detailed information about the DMS error code enter /HELP-MSG in system mode or see the BS2000 manual 'System Messages'.

SDP0442	`LAYOUT DOES NOT EXIST`
SDP0443	`SYNTAX OF PATTERN IS NOT A CORRECT WILDCARD SYNTAX`
SDP0444	`INTERFACE ERROR CONCERNING BUILTIN FUNCTION. CONTACT SYSTEM ADMINISTRATOR`
SDP0445	`USERID WITH MORE THAN 8 CHARACTERS IS NOT POSSIBLE`
SDP0446	`USERID UNKNOWN`
SDP0447	`THE GIVEN STRING IS NO SDF-LIST`
SDP0448	`THE PARAMETER 'NUMBER' OUT OF RANGE`

Meaning
There are less sublists than 'NUMBER'.
'NUMBER' is less or equal 0.

SDP0449	`ARRAY ELEMENTS ARE NOT ALLOWED`
SDP0450	`USERID LOCKED`
SDP0451	`NOT ENOUGH SPACE FOR STORING RESULT`

Meaning
There is not enough space allocated for storing the result.

SDP0452	`INVALID DATE`

Meaning
The input/output date is not correct, is lower than 1582-10-15 or greater than 9999-12-31.

SDP0453 `(&00) PARAMETER IS EMPTY OR ITS LENGTH IS GREATER THAN (&01) CHARACTER(S) OR CONTAINS ONE OR MORE SPACES`

SDP0454 `INVALID PARAMETER : '(&00)'`

Meaning
The syntax of the parameter is not correct.

SDP0455 `INVALID LENGTH OF INPUT STRING (ALLOWED : 1..256)`

Meaning
The input string is maybe an empty string or its length
is greater than 256.

Response
Try to fill the string (if empty) or reduce the length.

SDP0456 `LENGTH PARAMETER IS OUT OF RANGE (ALLOWED : 1..256)`

Meaning
The LENGTH parameter is null or greater than 256.

Response
Try a value between 1 and 256.

SDP0457 `LENGTH PARAMETER IS NOT A NUMERIC VALUE`

Meaning
LENGTH parameter contains probably other characters than digit.

Response
Set LENGTH parameter with a numeric value.

SDP0458 `BEGIN-DATE GREATER THAN END-DATE`

Response
Swap the two dates.

SDP0459 `Parameter error or invalid parameters combination. Additional information: '(&00)'`

Meaning
Explanations: Additional info =
1 invalid INPUT parameter.
2 internal error.
3 bad combination of DATA-TYPE and/or PATTERN with one of the following parameter CAT-ID, USER-ID, VERSION, GENERATION, WILDCARD, KEYSTAR, SEPARATORS, UNDERSCORE, ODD, CORRECTION-STATE, USER-INTERFACE, ALIAS, VOLUME-ONLY or
bad combination of parameters DEVICE-CLASS, EXCEPT-DISKS, EXCEPT-TAPES.
4 bad LOWEST-LENGTH / HIGHEST-LENGTH limits (HIGHEST < LOWEST, >SDF-A limits, bad values, no decimal limit, no limit allowed, ...).
5 invalid PATTERN (length=0, syntax).
6 invalid values (VALUE parameter).

Response
Correct and retry.

SDP0460 `THE GIVEN STRING IS NOT A STRUCTURE`

SDP0461 `THE NUMERIC VALUE FOR THE OPERAND MUST BE GREATER THAN ZERO`

SDP0462 `'(&00)' IS NOT A STRUCTURED-NAME`

SDP0463 `The given operand '(&00)' is unknown`

SDP0464 `TOO MANY AMBIGUITIES FOR THE GIVEN OPERAND`

SDP0465 `OPERAND OF TYPE BOOLEAN NOT ALLOWED`

SDP0467 `NO NAME FOUND; PROCESSING CONTINUES`

SDP0468 `NO NAME FOUND`

SDP0469 `Invalid parameter '(&00)' specified`

Meaning
Refer to IMON manual (GETINSP interface) for a more detailed description of these parameters.

Response
Correct and retry.

SDP0470 `Internal error returned by GETINSP/GETINSV interface. Return code '(&00)' received`

Meaning
The interface GETINSP/GETINSV has returned a non expected return code.

Response
Contact the system administrator.

`SDP0471` `SDF-P VERSION NOT SUPPORTED BY SDF-P-BIF`

Meaning
The version of the system interface used by SDF-P is not supported by SDF-P-BIF.
All built-in functions loaded by SDF-P-BIF are ignored.

Response
Please use another version of SDF-P-BIF kernel which supports the current SDF-P version.
See SDF-P manual for valid SDF-P/SDF-P-BIF combinations.

`SDP0472` `Null byte (x'00') not allowed in STRING and FIELD-SEPARATOR operands`

`SDP0473` `Internal error during builtin function`

`SDP0474` `Syntax error in regular expression for operand FIELD-SEPARATOR`

`SDP0475` `Variable must be a STRUCTURE or a LIST/ARRAY`

`SDP0476` `Result string too long`

Meaning
The result string may not be longer than 4096 characters.

`SDP0477` `Incorrect SDF structure`

`SDP0478` `Invalid STRING syntax in *STRING-TO-VARIABLE value`

`SDP0479` `Invalid STRING syntax in *STRING-TO-VARIABLE value near '(&00)'`

`SDP0480` `Incorrect SDF structure for variable '(&00)'`

`SDP0481` `Value of operand 'POSITION' must be greater than zero`

`SDP0482` `An input string is too long (1..255)`

Meaning
The operands
- INPUT-NAME
- WILDCARD-PATTERN
- CONSTRUCTION-WILDCARD
may not be longer than 255 characters.

`SDP0483` `Incorrect CONSTRUCTION-WILDCARD value`

`SDP0484` `Too long result string (1..255)`

Meaning
The output string may not be longer than 255 characters.

`SDP0485` `Value of operand 'FIELD-NUMBER' must be greater than zero`

`SDP0486` `Odd number of apostrophes in STRING value`

SDP0487 `Invalid type of expression in *STRING-TO-VARIABLE() operand`

Meaning
*STRING-TO-VARIABLE() operand value must be of type string.

SDP0488 `Spaces not allowed in input strings`

Meaning
The operands
- INPUT-NAME
- WILDCARD-PATTERN
- CONSTRUCTION-WILDCARD
may not contain spaces (' ').

SDP0489 `Warning: Installation-Unit '(&00)' not found in IMON Software Inventory. Default value assumed`

SDP0490 `Installation-Unit '(&00)' version '(&01)' not found`

SDP0491 `Warning: Logical-id '(&00)' not found in Installation-Unit '(&01)' version '(&02)'. Default value assumed`

SDP0492 `Null byte (x'00') not allowed in PATTERN operand and list elements`

SDP0493 `Value of operands BEGIN-INDEX, END-INDEX, BEGIN-COLUMN and END-COLUMN must be greater than zero`

SDP0494 `Syntax error in regular expression for operand PATTERN`

SDP0495 `'(&00)' not a correct JV name`

Meaning
The JV name (&00) must be a full-filename_1..54.

SDP0496 `User is not privileged to see installation information`

SDP0497 `No path name assigned to logical-id '(&00)'`

SDP0498 `BEGIN-COLUMN must not be greater than END-COLUMN.`

SDP0499 `FIRST-RECORD must not be higher than LAST-RECORD.`

SDP0510 `Stream name '(&00)' invalid`

Meaning
A reserved stream name (SYSDTA, SYSLST, SYSCMD, SYSOUT, SYSIPT, SYSOPT or stream name beginning with '$') has been used.
Check stream name and retry.

SDP0511 `Assignment invalid for stream '(&00)'`

Meaning
The specified stream could not be assigned :
- assignment to father stream creates a cycle over the previously assigned streams.
- ...

Response
- assign the stream to another father
- correct error cause and retry.

SDP0512 `Server no longer active, S-stream reset to *DUMMY. /TRANSMIT-BY-STREAM command ignored, processing continues`

Meaning
The server status is no longer valid at transmission time.
The server has cancelled the information related to this stream.
No transmission is therefore possible any longer on this stream.

Response
Check server environment and assign stream in a valid environment for this stream.

SDP0513 `System stream '(&00)' cannot be deleted`

SDP0514 `Stream '(&00)', assigned to another one, cannot be deleted`

Meaning
The stream cannot be deleted because it has been assigned to another stream (son stream). Deleting this stream would leave the son stream without any destination.

Response
Please assign the son stream to another destination or *dummy (the son stream can be found with /show-stream-assignment).
Then repeat /delete-stream.

SDP0515 `Stream not created at primary level`

Meaning
Streams can only be deleted at primary level. This avoids that the results of stream deletion depend of the current environment.

Response
/delete-stream can only be input when all procedures are exited or system-file-contexts of all calling procedures are *same from primary level up.

SDP0516 `Stream '(&00)' does not exist, processing continues`

SDP0517 `SPECIFIED STREAM '(&00)' DOES NOT EXIST`

- Warranty: Y

Meaning
The specified stream has not been created in the current context (procedure or dialog).

Response
The stream must be created by /ASSIGN-STREAM command.

SDP0518 `NO MATCH FOR SPECIFIED WILDCARD PATTERN, PROCESSING CONTINUES`

SDP0519 `NO MATCH FOR WILDCARD PATTERN`

SDP0520 `Specified variable '(&00)' invalid`

Meaning
The variable must be of type structure.

Response
Verify the variable type.

SDP0522 `Transmitted data incompatible with format processed by server`

Meaning
The server could not process the current request for one of the following reasons :
- the name of the transmitted data is not expected by the server
- some data are missing in the transmitted variables
- the attributes of the transmitted data do not match the attributes expected by the server.

SDP0524 `Server no longer active, S-stream reset to *DUMMY`

Meaning
The server status is no longer valid at transmission time.
The server has cancelled the information related to this stream.
No transmission is therefore possible any longer on this stream.

Response
Check server environment and assign stream in a valid environment for this stream.

SDP0530 `Server '(&00)' does not exist`

SDP0531 `Warning returned by server`

Meaning
Warning from server.
More information stored in RET-CONTROL-VAR-NAME variable if specified at /ASSIGN-STREAM and /TRANSMIT-BY-STREAM commands.

Response
Check contents of RET-CONTROL-VAR-NAME variable.

`SDP0532` `Error returned by server`

Meaning
Error from server.
More information stored in RET-CONTROL-VAR-NAME variable if specified at /ASSIGN-STREAM and /TRANSMIT-BY-STREAM commands.

Response
Check contents of RET-CONTROL-VAR-NAME variable.

`SDP0533` `Server linkage error`

`SDP0534` `INTERNAL ERROR RETURNED BY SERVER`

Meaning
Server aborted after unexpected event or missing or lack of system resources.

`SDP0535` `WARNING RETURNED BY SERVER AT DELETION OF STREAM '(&00)', PROCESSING CONTINUES`

`SDP0536` `Error returned by server, delete rejected for stream '(&00)'`

`SDP0537` `Internal error returned by server, delete rejected for stream '(&00)'`

`SDP0538` `Recursive call of stream server rejected`

Meaning
The server for structured streams assigned by /ASSIGN-STREAM ...TO=*VARIABLE(...) has detected a recursive call in error situation and aborts the recursion by cancelling every next operation.
The primary call terminates with error.

Response
The recursion can be caused by the current user environment.
The server uses MIP and SDF-P services which probably operate on standard streams (SYSINF, SYSMSG) which are in turn assigned to this server. By eliminating the cause of the original error, the recursive call will be avoided.

`SDP1006` `INTERNAL ERROR IN VARIABLE HANDLER. CONTACT SYSTEM ADMINISTRATOR`

`SDP1007` `No variable declared`

Meaning
The variable pool is empty.

`SDP1008` `VARIABLE/LAYOUT '(&00)' DOES NOT EXIST`
◆ Warranty: Y

`SDP1010` `VARIABLE '(&00)' HAS NO VALUE`

`SDP1014` `NO STRUCTURE OR LAYOUT DECLARATION INITIATED`

SDP1015 `STRUCTURE CLOSED INTERNALLY`

Meaning
Not all structures declared with /BEGIN-STRUCTURE have been closed by means of /END-STRUCTURE by the end of the procedure.

SDP1017 `WARNING: VARIABLE '(&00)' ALREADY EXISTS`

SDP1018 `VARIABLE '(&00)' ALREADY EXISTS BUT WITH OTHER ATTRIBUTES`

◆ Warranty: Y

Response
Change the name of the variable.

SDP1019 `ARRAY BOUND OR LIMIT OUT OF RANGE`

SDP1020 `UPPER LIMIT OF ARRAY OR LIST ELEMENTS OF VARIABLE '(&00)' REACHED`

SDP1022 `JOB VARIABLE '(&00)' NOT ACCESSIBLE`

Meaning
Possible reasons:
The required password was not provided.
Only read access is allowed.

SDP1023 `JOB VARIABLE '(&00)' ALREADY EXISTS`

Meaning
An attempt has been made to create an existing job variable as a container.

Response
Check the STATE operand of the declaration and use the value STATE=OLD or STATE=ANY.

SDP1024 `JOB VARIABLE '(&00)' DOES NOT EXIST`

Response
Specify STATE=NEW or STATE=ANY or create the job variable.

SDP1026 `STRING TOO LONG FOR JOB VARIABLE '(&00)'`

Meaning
The string has more than 256 characters.

SDP1027 `VALUE FOR JOB VARIABLE '(&00)' IS NOT A STRING`

Meaning
The value of a variable with a job variable as container must be a string.

SDP1029 `VALUE TYPE NOT ALLOWED FOR VARIABLE 'SYSSWITCH'`

Meaning
The value must be of type BOOLEAN.

SDP1030 `CONTAINER / VARIABLE-CONTAINER '(&00)' DOES NOT EXIST`

- Warranty: Y

Meaning
- Variables must exist before they can be used as containers.
- VARIABLE-CONTAINER has been closed.

SDP1031 `ATTRIBUTES OF CONTAINER DO NOT MATCH THOSE OF DECLARED VARIABLE`

SDP1032 `VARIABLE '(&00)' ALREADY EXISTS`

SDP1033 `Scope of container is smaller than scope of variable '(&00)'`

SDP1034 `TYPE OF CONTAINER '(&00)' DOES NOT MATCH TYPE OF VARIABLE`

SDP1035 `THIS ELEMENT CANNOT BE DECLARED IMPLICITLY`

SDP1036 `VARIABLE TYPE AND VALUE TYPE DO NOT MATCH`

SDP1037 `Variable '(&00)' cannot have or get a value`

Meaning
The variable (&00) is the name of a structure or of an array or of a list.
A simple value (i.e.: a string, an integer or a boolean) or a name of variable containing a simple value was expected.
If the variable (&00) must be supplied as parameter to a built-in function expecting the name of a structure, of an array or of a list, the variable name (&00) must be enclosed between quotes (e.g.: '(&00)').

SDP1038 `/FREE-VARIABLE NOT POSSIBLE FOR VARIABLE 'SYSSWITCH'`

SDP1039 `'#' MISSING OR MISUSED`

Meaning
A '#' was used in variable name and the variable is no array or list.
The '#' was forgotten and the variable is an array or a list.

SDP1040 `VARIABLE '(&00)' MUST BE A VARIABLE OR LIST OF TYPE 'STRING'`

Meaning
The variable has a job variable container and must be of type 'string'.
/SHOW-VARIABLE needs a list of type 'string' as an output variable.
/READ-VARIABLE needs a list of type 'string' as an input variable.
/EXEC-CMD needs a list of type 'string' as the TEXT-OUTPUT variable.
/CALL-PROCEDURE needs a list of type 'string' as variable.

SDP1041 `STRUCTURE/ARRAY/LIST '(&00)' DOES NOT EXIST`

SDP1042 `AGGREGATE NODE '(&00)' DOES NOT EXIST`

SDP1043 `VARIABLE '(&00)' MUST NOT BE AN ARRAY ELEMENT OR A LIST ELEMENT`

SDP1044 `POOL ID INVALID, VARIABLES POOL DOES NOT EXIST. CONTACT SYSTEM ADMINISTRATOR`

SDP1045 `VARIABLE OR ELEMENT OF VARIABLE '(&00)' CANNOT BE DECLARED IMPLICITLY`

SDP1046 `NO IMPLICIT DECLARATION ALLOWED`

SDP1047 `STRUCTURE '(&00)' NOT COMPLETE. MEMORY SPACE SHORTAGE`

Meaning
There was no more memory space available during declaration of the structure.

SDP1048 `LAYOUT OF VARIABLE '(&00)' DOES NOT EXIST`

SDP1049 `LAYOUT ALREADY EXISTS BUT WITH OTHER ATTRIBUTES`

SDP1050 `LAYOUT DOES NOT EXIST`

SDP1052 `AGGREGATE ELEMENT NOT PRESENT`

Meaning
The first, last or next element of an aggregate was requested but not found.

SDP1054 `JOB VARIABLE ERROR: JVS ERROR CODE '(&00)' WHILE ACCESSING JOB VARIABLE '(&01)'. IN SYSTEM MODE: /HELP-MSG JVS(&00)`

SDP1056 `SYNTAX ERROR IN SET-VARIABLE COMMAND`

Meaning
Something is wrong in the syntax of the SET-VARIABLE command :
- the name of the variable is incorrect,
- or the value of the WRITE-MODE operand is incorrect
- or ...

Response
Correct the syntax of the command.

SDP1057 `NAME '(&00)' TOO LONG`

Meaning
The variable exists but is not accessible.
Possible reasons:
- The name is too long as a result of index evaluation (arrays).
- The name of the container is longer than the name of the variable.

SDP1058 `NOT ENOUGH MEMORY SPACE FOR VALUE BUFFER`

Meaning
There is no more memory space available for the value.

SDP1059 `NOT ENOUGH MEMORY SPACE FOR NAME BUFFER AND VALUE BUFFER`

Meaning
There is no more space available for name and value.

SDP1060 `WARNING: VARIABLE OR LAYOUT '(&00)' IMPORTED`

SDP1061 `ERROR WHILE IMPORTING LAYOUT OF VARIABLE '(&00)'`

SDP1063 `STRUCTURE NOT OPENED`

Meaning
/END-STRUCTURE is illegal because there is no open structure.

SDP1064 `LAYOUT OF VARIABLE '(&00)' NOT CLOSED`

Meaning
Possible reasons:
- The layout has not been closed before the first access.
- One or more /END-STRUCTUREs are missing.

SDP1065 `STEM '(&00)' OF AGGREGATE ELEMENT IS NOT A STRUCTURE`

SDP1066 `VARIABLE '(&00)' CANNOT BE DECLARED WITH A CONTAINER`

Meaning
A static structure declared with TYPE=STRUCT(*BY-SYSCMD) must never have a container.

SDP1067 `STRUCTURE NOT DECLARED`

Meaning
Possible reasons:
- The name of the layout has already been assigned.
- A /BEGIN-STRUCTURE has been entered without declaration of the structure with TYPE=STRUCT(*BY-SYSCMD).
- A /BEGIN-STRUCTURE command is supernumerary.

SDP1068 `LAYOUT DECLARATION NOT POSSIBLE`

Meaning
A structure or layout is still open.
A /BEGIN-STRUCTURE without operands is expected to start the element declarations of the structure declared with DEF=*BY-SYSCMD.

Response
The structure or layout must be closed with /END-STRUCTURE.

SDP1069 `STRUCTURE NOT CLOSED`

Meaning
Possible reasons:
One /END-STRUCTURE command is missing.
The assignment to the structure element is ignored, because the structure has not been closed.

Response
Close the structure first with the required number of /END-STRUCTURE commands, then try the assignment again.

SDP1070 `STRUCTURE NAME UNKNOWN`

Meaning
An invalid name was given in the /END-STRUCTURE command.

SDP1071 `FURTHER STRUCTURE DECLARATION NOT POSSIBLE`

Meaning
A structure or layout with TYPE=STRUCT(*BY-SYSCMD) has not been closed,
so it is not possible to declare another structure.

SDP1072 `OPERAND 'WRITE-MODE=(&00)' INVALID IN /SET-VARIABLE COMMAND`

Meaning
WRITE-MODE=EXTEND or WRITE-MODE=PREFIX is only possible for lists.
WRITE-MODE=MERGE is only possible for structures.

SDP1073 `THE TYPES OF TARGET AND SOURCE OF THE /SET-VARIABLE COMMAND DO NOT MATCH`

SDP1074 `/DECLARE PARAMETER ONLY ALLOWED IN PROCEDURE HEAD`

SDP1075 `WARNING: /SHOW-VARIABLE FOR OUTPUT LIST IGNORED`

Meaning
The contents of the output list are not shown in the output list itself.
E.g. /SHOW-VARIABLE A,OUTPUT=*LIST(A) is not possible.

SDP1076 `INVALID VALUE OF OPERAND 'LIMIT'`

Meaning
The list limit specified is zero or negative.
The UPPER-BOUND operand in an array declaration is smaller than LOWER-BOUND.

SDP1077 `INPUT LIST EQUAL TO OUTPUT LIST IS NOT ALLOWED. COMMAND REJECTED`

SDP1078 `NO ASSIGNMENT ALLOWED FOR THIS VARIABLE`

Meaning
READ-VAR: The variable is an array or a list.

SDP1079 `CONVERSION ERROR WHILE READING THE INPUT`

Meaning
The types of the value and of the variable into which the value is to be read are not compatible.

SDP1080 `INVALID INPUT FORMAT ENTERED FOR OPERAND '*BY-INPUT' IN /READ-VARIABLE COMMAND`

SDP1081 `MISSING INPUT RECORD`

Meaning
The input file or input list of the /READ-VARIABLE command does not contain enough values.

SDP1082 `FILE '(&00)' NOT AN ISAM FILE`

SDP1083 `LIST OF AGGREGATES NOT ALLOWED`

SDP1084 `WARNING: INITIALIZATION OF LAYOUT ELEMENTS IGNORED`

SDP1085 `NO SDF-P COMMANDS PERMITTED IN NOT STRUCTURED PROCEDURES`

SDP1086 `SCOPE OF LAYOUT SMALLER THAN SCOPE OF VARIABLE '(&00)'`

SDP1087 `VARIABLE '(&00)' ON THE LEFT SIDE OF THE ASSIGNMENT MUST BE AN AGGREGATE`

SDP1088 `ELEMENT NAME '(&00)' IN INVALID CONTEXT`

Meaning
/DECLARE-ELEMENT with data type 'composed name' is only possible for dynamic structures.

SDP1089 `OPERAND '*PROMPT' OF /DECLARE-PARAMETER IN COMBINATION WITH 'TRANSFER-TYPE=BY-REFERENCE' NOT ALLOWED`

SDP1090 `JOB VARIABLES NOT POSSIBLE AS CONTAINERS`

SDP1091 `ADDITION OF ELEMENTS TO STATIC STRUCTURES NOT POSSIBLE`

Meaning
The structure must not be enlarged:
no further /DECLARE-ELEMENT is allowed, neither implicitly nor explicitly.

SDP1092 `MULTIPLE DECLARATION OF STRUCTURE ELEMENTS NOT POSSIBLE`

SDP1093 `OK, VARIABLE EXISTS, STRUCTURE IS ALREADY CLOSED`

SDP1094 `THE CREATED ELEMENT NAME IS TOO LONG (> 255)`

Meaning
/SET-VAR <name> = <expr>,MERGE may create element names which are too long
A recursive assignment may create an element name which is too long.

SDP1095 `WARNING: STRUCTURE IS EMPTY`

SDP1096 `VARIABLE '(&00)' MUST BE A LIST OF TYPE STRING OR ANY CONTAINING ONLY STRING VALUES`

SDP1097 `ERROR IN PROMPT OPERAND`

Meaning
The prompt value is not of type string or exceeds size limit.

SDP1098 `/DELETE-VARIABLE not allowed for the variable '(&00)'`

Meaning
/DELETE-VARIABLE not possible for :
- SYSSWITCH variable
- procedure parameter
- element of complex variable.

SDP1099 `Cannot free list element '(&00)'`

Meaning
Only the first and last elements of a list may be suppressed by the command /FREE-VARIABLE.

SDP1100 `Creation of gaps not allowed for a list in the variable '(&00)'`

Meaning
New list elements may be created only at the end of the list, provided that no gaps are created:
if the size of the list is 'n', only the element 'n+1' may be created.

Response
Use the /SET-VARIABLE command with WRITE-MODE = *PREFIX to add new list elements before the head of the list.

SDP1101 `SYNTAX ERROR IN VARIABLE NAME`

SDP1102 `Task variable '(&00)' has been deleted`

SDP1103 `Value of constant variable '(&00)' cannot be modified`

Meaning
The value of the constant variable (&00) cannot be modified by means of the SET-VARIABLE, READ-VARIABLE or FREE-VARIABLE commands.
The variable declaration may be deleted by means of DELETE-VARIABLE.

SDP1104 `VARIABLE '(&00)' MUST BE A LIST OF ELEMENTS WITH SIMPLE VALUES`

SDP1105 `Constant variable already declared with another value`

Meaning
Multiple declarations of the same constant variable is allowed provided the values are identical.

SDP1106	`Error while reading on SYSDTA`
SDP1107	`VARIABLE '(&00)' CAN NOT BE SORTED. IT IS A LIST OF ELEMENTS WITH MIXED TYPE VALUE`
SDP1120	`Variable '(&00)' must be a list of simple types or a list of structures` **Meaning** The input variable must be a list of simple types (string, integer or boolean) or a list of structures.
SDP1121	`Variable '(&00)' is not a list of structures` **Meaning** Names of structure elements can not be specified in the DISPLAYED-ELEMENTS operand if the input variable is not a list of structures.
SDP1122	`Variable '(&00)' must be a list of dynamic structures` **Meaning** An element SELECTION-CODE will be added to the structure. This is only possible if TO-VARIABLE is a list of dynamic structures.
SDP1123	`Value too long for variable '(&00)'` **Meaning** Value for string variables are limited to 4096 characters.
SDP1124	`Variable '(&00)' is not initialized`
SDP1131	`INDEX EVALUATION ERROR`
SDP1132	`VARIABLE NAME '(&00)' OR SUBNAME TOO LONG` **Meaning** The name of a variable must not be more than 255 characters in length. A subname of a variable must not be more than 20 characters in length.
SDP1133	`INDEX OF VARIABLE '(&00)' OUT OF RANGE` **Meaning** Only values between -2**31 and 2**31-1 may be used.
SDP1134	`VARIABLE NAME RESERVED FOR TPR APPLICATIONS`
SDP1135	`NO ARRAY ELEMENTS ALLOWED HERE`
SDP1136	`NO LIST ELEMENTS ALLOWED`
SDP1137	`NO LIST DECLARED`

SDP1138 `RESERVED KEYWORD USED`

Meaning
This name must not be used as a variable name or a function name.
The following names are reserved:
- AND, LE, OFF, DIV, LT, ON, EQ, MOD, OR, FALSE, NE, TRUE, GE,
- NO, OR, GT, NOT, YES.

SDP1139 `Parameter already declared`

Meaning
Two declaration of the same parameter are not allowed.

SDP1140 `Not a simple variable name '(&00)'`

Meaning
The variable name (&00) may not contain a '.' nor a '#'.

SDP1141 `Attributes of variable '(&00)' to be declared not allowed in SDF-P basic version`

Meaning
Only variables of TYPE=*ANY and MULTIPLE-ELEMENTS=*NO can be automatically declared in SDF-P basic version.

Response
Please, contact system administrator to start subsystem SDF-P.

SDP1201 `Variable container '(&00)' already exists but with other attributes`

SDP1202 `WARNING: Variable Container '(&00)' already exists`

SDP1203 `Variable Container '(&00)' does not exist`

SDP1204 `Contained variable '(&00)' cannot be processed by this version of SDF-P`

SDP1205 `Variable Container '(&00)' cannot be processed by this version of SDF-P`

SDP1206 `Variable container '(&00)' is corrupted`

SDP1207 `Contained variable '(&00)' does not match the specified structure`

SDP1208 `Version *INCREMENT not allowed for Variable Container '(&00)'`

Meaning
Version *INCREMENT cannot be specified when the Variable Container has been opened with LOCK-ELEMENT = *YES.

SDP1209 `Value of SCOPE parameter not allowed for OPEN-VARIABLE-CONTAINER in procedure head`

SDP1210 `Variable Container has been closed. Information not available`

SDP1211 `Variable '(&00)' already declared but refer to a closed or a no more visible variable-container`

Response
The variable declaration may be deleted by means of /DELETE-VARIABLE.

SDP1300 `Procedure compiler version '(&00)' started`

SDP1301 `Procedure compiler terminated normally`

SDP1302 `Procedure compiler terminated abnormally`

SDP1303 `Command '(&00)' cannot be processed by the current SDF-P version`

Meaning
The command (&00) comes from a procedure compiled in a higher SDF-P version and cannot be executed by the current version.

Response
Upgrade to the latest SDF-P version.

SDP1304 `Compiled procedure corrupted at command '(&00)'`

SDP1305 `Error when writing result of compilation`

SDP1306 `Internal error in compiler component. Contact system administrator`

SDP1307 `*SAME cannot be specified for incompatible FROM-FILE and TO-FILE operands`

Meaning
The value *SAME has been specified for operand LIBRARY and/or ELEMENT and/or VERSION. This is not allowed if FROM-FILE is not also a library-element.

SDP1308 `Not structured procedures cannot be compiled`

SDP1401 `Compiled procedure '(&00)' corrupted`

SDP1402 `Procedure '(&00)' is not a S-procedure. Only elements of type 'J' supported`

SDP1403 `Procedure '(&00)' cannot be processed by this version of SDF-P`

SDP1404 `Invalid operand TYPE specified. Only '*STD' is processed by this version of SDF-P`

SDP1405 `This procedure cannot be processed by this version of SDF-P`

Meaning
An element with the specified name of type SYSJ has been found in the library. SDF-P cannot choose which procedure must be executed. This functionality is only supported from SDF-P-BASYS V02.0B.

SDP1406	Library element ’(&00)’ not found
SDP2000	Warning: Not all list elements could be treated successfully
SDP2001	None of the list elements could be treated successfully
SDP2002	Error when treating list element ’(&00)’. Processing continues
SDP2003	Error when treating list element ’(&00)’. Processing aborted

Glossary

This glossary provides definitions of the main terms used in the SDF-P procedure concept. Terms given in italics in the definitions are themselves defined in the glossary.

background procedure

Procedure that runs in the background independently of the calling job and which contains a separate job number (TSN). Both S procedures and non-S procedures can run as background procedures

branching

Control structure in *S procedures*, in which command sequences are executed regardless of the result of a condition. Branching is initiated by a start command and terminated with a termination command. Branching is also referred to as *IF blocks*.

called procedure

Procedure called from another procedure and subordinate to that procedure. The called procedure is executed in full before it returns control to the *calling procedure*.

calling procedure

Procedure that calls a subordinate procedure. The calling procedure relinquishes control to the subordinate procedure while the latter is being executed.

command block

Part of an *S procedure* that brings together related parts of procedures into a logical unit. A command block starts with a start command and ends with a termination command. In between are the commands to be executed in this block. Command blocks include *loops, branching* and simple command blocks.

control flow command

Commands in *S procedures* that control *procedure execution*. Control flow commands include branch commands as well as paired start and termination commands that start and terminate command blocks. Control flow commands in SDF-P are: BEGIN-BLOCK, BEGIN-PARAMETER-DECLARATION, CYCLE, ELSE, ELSE-IF, END-BLOCK, END-FOR, END-IF, END-WHILE, END-PARAMETER-DECLARATION, EXIT-BLOCK, FOR, GOTO, IF, IF-BLOCK-ERROR, IF-CMD-ERROR, REPEAT, UNTIL, WHILE.

control structure

Part of *S procedures* for controlling *procedure execution*. Control structures include *loops, branching* and simple *command blocks*. They are each started and terminated with paired control flow commands. Branch commands also belong to *control flow commands*.

ELSE branch

Part of an *IF block* containing an alternative command sequence which is executed if the condition in the IF command is not met. An ELSE branch is started and terminated by the ELSE command.

error handling

In *S procedures,* this is based on the fact that BS2000 commands supply a defined return code. This return code allows SDF-P to determine whether there was an error when the command was executed. Error handling is automatically triggered if a return code indicating an error is returned when a command is executed. Error handling itself is carried out in what are known as error handling blocks. There are two types of such blocks: IF-BLOCK-ERROR and IF-CMD-ERROR. In the event of an error in *procedure execution*, SDF-P branches to the next IF-BLOCK-ERROR or IF-CMD-ERROR command.

FOR block

Loop in *S procedures* in which a command sequence is executed for as long as values are assigned to a run variable and, optionally, a defined condition is met. The FOR block begins with the command FOR and ends with the command END-FOR. The command FOR contains the run variable.

foreground procedure

Procedure executed under the control of the job in which it was called; no new job is created.

formal procedure parameter

Variable in a *procedure* to which the current parameter specified in the *procedure call* is transferred. Formal procedure parameters are declared in the *procedure head*.

IF block

Conditional *branching* in *S procedures*. An IF block is started with the IF command and terminated with the END-IF command. The condition for branching is specified in the IF command, and an alternative condition in the ELSE-IF command. An IF block always consists of a *THEN branch*, executed if the condition is met, and optionally an *ELSE branch*, executed if the condition is not met.

language elements in S procedures

These are commands, statements, data records, variables, functions and expressions.

loop

Control structure in *S procedures* that can be executed several times as a function of a condition. A loop is started by a start command and ended with a termination command.

non-S procedure

Procedure in BS2000 not created in line with SDF-P rules. Non-S procedures can be executed as *foreground* and *background procedures*.

procedure

Frequently used sequences of commands, statements and data records stored in a *procedure container*. These command sequences can be executed with a single command in interactive or batch mode.

procedure attributes

These are set in the SET-PROCEDURE-OPTIONS command and affect calling, execution and *error handling* of the *procedure*. The procedure attributes include: command for the procedure call, system file environment, length of the procedure records, type of logging, interruptibility of the procedure, type of error handling, escape characters, type of variable declaration, and job variable replacement. Procedure attributes can be modified with the MODIFY-PROCEDURE-OPTIONS command.

procedure body

Part of an *S procedure* that determines the execution of the procedure. The procedure body follows the procedure *head*. It comprises a series of commands, statements and data. The S procedure can be controlled via the various types of *command blocks* that support SDF-P, e.g. via *branching*, *loops* or error handling blocks.

procedure call
Initiation of the *procedure start*, in which the procedure to be started is named and if necessary is assigned *procedure parameters*. When calling *S procedures*, the commands of the *procedure head* are executed first and the *procedure attributes* set before the commands of the *procedure body* are analyzed and processed.

procedure container
File, library element or list variable in which a *procedure* is stored.

procedure environment
This comprises all system data and system attributes that affect execution of the procedure, e.g. task variables, *procedure attributes*, variables of the *calling procedure*.

procedure execution
The execution part of the *procedure start*, in which the commands are executed in accordance with BS2000 rules. The expressions within the command are replaced first. The command is then analyzed and executed.

procedure head
Part of an *S procedure* in which the global *procedure attributes* are defined and the *procedure parameters* declared. The procedure head is at the start of an S procedure.

procedure interpreter
This checks the *procedure head,* executes it and then analyzes the *procedure body*. When executing the procedure, the procedure interpreter identifies the *control flow commands* and executes them.

procedure interruption
Interruption of the procedure run of a *procedure* which was called interactively as a foreground procedure. A distinction is made between interruptions from system level and procedure-internal interruptions. Within the procedure, the procedure run can be interrupted at any time with the HOLD-PROCEDURE command. From the system level, a procedure can be interrupted with the K2 function key.

procedure line
Data record of the *procedure container*, containing the data, commands and statements for the procedure. Command and statement sequences may extend over several procedure lines.

procedure nesting

This refers to calling a procedure from another procedure, where each *procedure container* may contain just one *procedure*.

procedure parameter

In SDF-P, a general term for current parameters and *formal parameters*. It is used as a synonym for current and formal parameters when it is not necessary to distinguish between them. Procedure parameters are identified by the following attributes: parameter name, start value (if specified), data type and type of parameter transfer.

procedure record

Processing unit of a *procedure*, comprising commands, statements or data processed at the same time by the system during execution. Several procedure records consisting of commands or statements can be written in a *procedure line*, separated by a semicolon.

procedure start

This includes calling, analyzing and executing the *procedure*.

procedure termination

Procedures can be terminated with the following *procedure termination commands*: the SDF-P command EXIT-PROCEDURE, the END-PROCEDURE command, and the CANCEL-PROCEDURE command.

procedure termination command

Command with which *procedure execution* can be stopped at any point. The procedure termination command can be used to provide the procedure caller with error information.

REPEAT block

Loop in *S procedure*s, in which a command sequence is executed for as long as a defined condition is no longer met. The REPEAT block is executed at least once. It starts with the REPEAT command and ends with the UNTIL command.

SDF-P

Procedure language that expands the BS2000 command language into a programming language in which structured programming is possible in the same way as in higher-level programming languages. A significant feature of procedures under SDF-P is execution by *command blocks*.

S procedure

Structured procedure in BS2000, corresponding to the SDF-P procedure format. The main parts of an S procedure are the *procedure head* and the *procedure body*. Related parts of a procedure can be brought together in *command blocks*. *Error handling* in S procedures is block-oriented.

THEN branch

Part of an *IF block* executed if the condition in the IF command is met. The THEN branch is started and executed by the IF command.

WHILE block

Loop in *S procedures* in which a command sequence is executed for as long as a defined condition is no longer met. If the condition is not met at the time of first execution, the command sequence is never executed. The WHILE block starts with the WHILE command and ends with the END-WHILE command.

Related publications

The manuals are available as online manuals, see *http://manuals.fujitsu-siemens.com*, or in printed form which must be paid and ordered separately at *http://FSC-manualshop.com*.

[1] **BS2000/OSD-BC**
Introductory Guide to DMS

[2] **BS2000/OSD-BC**
DMS Macros

[3] **BS2000/OSD-BC**
Commands, Volumes 1 - 5
User Guide

[4] **BS2000/OSD-BC**
Commands, Volume 6, Output in S Variables and SDF-P-BASYS
User Guide

[5] **JV** (BS2000/OSD)
Job Variables
User Guide

[6] **AID** (BS2000/OSD)
Advanced Interactive Debugger
Debugging of ASSEMBH programs
User Guide

[7] **BS2000/OSD-BC**
Executive Macros
User Guide

[8] **BS2000/OSD-BC**
Introductory Guide to Systems Support
User Guide

[9] **HIPLEX MSCF** (BS2000/OSD)
BS2000 Processor Networks
User Guide

[10] **SDF** (BS2000/OSD)
SDF Management
User Guide

[11] **LMS** (BS2000/OSD)
Subroutine Interface
User Guide

[12] **IMON** (BS2000/OSD)
Installation Monitor
User Guide

[13] **SECOS** (BS2000/OSD
Security Control System
User Guide

[14] **BS2000/OSD-BC**
System Installation
User Guide

[15] **BS2000/OSD-BC**
System Messages
User Guide

[16] **SDF-A** (BS2000/OSD)
User Guide

[17] **SDF-CONV** (BS2000/OSD)
User Guide

[18] **POSIX** (BS2000/OSD)
POSIX Commands
User Guide

[19] **FHS** (BS2000/OSD, TRANSDATA)
Dialog Extension for TIAM and SDF-P
User Guide

[20] **SDF** (BS2000/OSD)
Introductory Guide to the SDF Dialog Interface
User Guide

[21] **MSGMAKER** (BS2000/OSD)
Creating and Editing BS2000 Message Files
User Guide

[22] **DSSM / SSCM** (BS2000/OSD)
Subsystem Management in BS2000/OSD
User Guide

[23] **CALENDAR** (BS2000/OSD)
User Guide

[24] **BS2000/OSD-BC**
System Exits
User Guide

[25] **BS2000/OSD**
Softbooks English
CD-ROM

Internet address
http://manuals.fujitsu-siemens.com

Index

A
abbreviation 231
ABEND 294
account number
  requesting 348
ACCOUNT( ) 348
activation
  error handling 84
ADD-CJC-ACTION 293
AID sequences 293
  compatibility 293
alias 545
alphanum-name (data type) 546
alternative branch
  initiating 621
analyze string 423, 426
analyze variable 381, 453
append list element 375
array 138, 142, 168, 174
  declaring 142
array element name 152
array index
  requesting 349
array size 478
ARRAY-INDEX( ) 349
assign S variable stream 563
ASSIGN-STREAM 563
asterisk 257
attribute definition, jobs 634

B
back branch
  limiting 697
background procedure 120
base term 249, 250
basics 135
batch job 631
  attribute definition 634
  define priority 636, 640
  monitor (job variable) 636
  repeat 638
  runtime logs 642
  start 637
batch mode 291
BEGIN block 92
BEGIN-BLOCK 568
  execute as a subprocedure 681
  jump to 681
BEGIN-PARAMETER-DECLARATION 572
BEGIN-PROCEDURE 293
BEGIN-STRUCTURE 573
BIFDEF macro 305
BIFDESC macro 307
BIFMDL1 macro 309
BIFMDL2 macro 311
block error handling
  initiating 674
block name 58
BOOLEAN 156, 485
Boolean constants 251
Boolean expression
  translating 518
Boolean value
  converting 351
BOOLEAN( ) 351
branch command 54, 100
branch destination 102
  random 104
branch tag 58
  creating 289

branches 54, 100
built-in function 223

C
C literal
  checking 404
  converting 383
C string 252
calendar 366, 370, 371, 450, 511
CALL command, replacing 289
call counter 363
CALL-PROCEDURE 681
CALL-PROCEDURE command 293, 576
  adapting 289
cancel 78
CANCEL-PROCEDURE 293
cat (suffix for data type) 557
catalog entry 406
  checking 406
catalog ID 482
  home pubset 389
  requesting 389
cat-id (data type) 546
changes 21
character 294
  converting 352, 401
  significant 294
character set, S tag 55
character string 252
CHARACTER-TO-INTEGER( ) 352
check
  C literal 404
  catalog entry 406
  expression 417
  file size 413
  function 354
  library element 421
  library name 419
  logging 444
  variable declaration 411
  variable initialization 415
  variable name 428
  whether SDF is loaded 424
  X literal 430
CHECK-DATA-TYPE( ) 354
CLIEXPR macro 314
CLIGET macro 320
CLISET macro 323
close variable container 581
CLOSE-VARIABLE-CONTAINER 581
command
  INCLUDE-BLOCK 681
  logging 112
  REPEAT-CMD 717
  REPEAT-STMT 721
  restrictions 293
  separation 52
  user-defined 295
command block 92
  terminating 622, 653
command error handling
  initiating 676
command length 52
command line 63
command return code 71, 241, 448, 451, 486, 488, 559
  error classes 559
  saving 724
command separator 294
command sequence
  conditional 625, 672
  starting 576, 686
command source, outputting 504
command-rest (data type) 546
comment 276
comment commands 53
comment syntax 53
comments 53
commutative law 269
compare strings 534
comparison
  Boolean values 261
  numeric 261
compiled procedure
  call (performance aspect) 275
COMPILE-PROCEDURE 582
compiler 74
  starting 582

compl (suffix for data type) 552
complex variable level
  requesting 453
composed-name 150
composed-name (data type) 546
composed-variable-name 150
concatenation operator 264
constructor (string) 555
continuation character 51, 53, 58
continuation handling 53, 732
continuation lines 52, 60
control flow command 132
control structure 132
conversion function 240, 351, 352, 383, 385, 386, 387, 399, 401, 446, 485, 514, 515, 518, 523, 532
conversion functions 513
convert
  Boolean value 351
  C literal 383
  character 352, 401
  expression 386, 399, 485
  integer 402
  string 513, 514
  uppercase letters 446
  variable 532
  X literal 385, 538
corr (suffix for data type) 557, 558
COUNTER( ) 363
counter, function calls 363
create
  end-of-file condition 66
c-string (data type) 546
CURRENT-TYPE( ) 364
CYCLE 103, 587

D
data
  logging 112
  reading in 59
data line 59, 63
data type 90, 156
  alphanum-name 546
  cat-id 546
  command-rest 546
  composed-name 546
  c-string 546
  date 546
  device 546
  dynamically changing 136
  filename 547
  fixed 137, 546
  integer 548
  name 548
  partial-name 549
  posix-filename 549
  posix-pathname 549
  product-version 550
  structured-name 550
  text 550
  time 550
  vsn 550
  x-string 551
  x-text 551
data types in SDF 542, 546
  suffixes 543
date
  outputting 366
  outputting particular 368
date (data type) 546
DATE( ) 366
DATE-VALUE( ) 368
day of the week
  outputting 370
DAY( ) 370
days difference
  outputting 371
debugging utility 64
declaration
  explicit 139
  implicit 136, 139
  variables 87, 136
DECLARE-CONSTANT 589
DECLARE-ELEMENT 594
DECLARE-PARAMETER 288, 572, 600
DECLARE-VARIABLE 607
default catalog ID
  requesting 482

default pubset 482
default value
  transferring 228
DELETE-STREAM 616
DELETE-VARIABLE 618
device (data type) 546
dialog
  guided 78
  unguided 78
division 258
DO command 289, 294
documentation
  procedure 53

E
ELAPSED-DAYS( ) 371
element name 152
  outputting 754
ELSE 620
ELSE branch
  initiating 620
ELSE-IF 621
end of block 102
end of line 77
END-BLOCK 622
END-FOR 623
END-IF 625
endless loops, preventing 283
end-of file condition
  creating 66
end-of-file condition 59, 63, 68
end-of-line comment 276
end-of-line comments 54
END-PARAMETER-DECLARATION 627
END-STRUCTURE 628
END-WHILE 629
Enter file 291
ENTER-PROCEDURE 631
environment information 236, 348, 366, 368, 370, 389, 390, 396, 413, 450, 457, 458, 480, 481, 506, 510, 511
  calendar 371
EOF condition 66
  generating 732
equals sign 294, 295
  deleting 290
error classes 486
  command return code 559
  Subcode1 559
error code
  requesting 448
error condition
  data lines 73
error handling 69
  activating 84, 704
  inserting 290
error handling block 72, 290
error message 241
error situation
  terminating 293
error transfer 73, 116, 121
escape character 55, 58, 86
evaluate user switch 526
execute subprocedure 681
EXECUTE-CMD 648
EXIT-BLOCK 102, 653
EXIT-JOB 294
EXIT-PROCEDURE 117, 295, 656
expansion
  variables 137
explicit command call
  outputting 373
explicit declaration 139
EXPLICIT-CALL( ) 373
expression
  checking 417
  complex 249
  converting 386, 399, 485
  evaluation 266
  regular 471
  simple 249
  types 265
expression replacement 51, 55, 78
EXTEND-SDF-LIST( ) 375
extract field 377
EXTRACT-FIELD( ) 377

F
FHS 201
  as output server 201
file information 406
file size
  checking 413
filename (data type) 547
fill character 379
fill string 379
FILL( ) 379
FIRST-VARIABLE-NAME( ) 381
fixed (data type) 546
FOR 659
FOR block 96
  initiating 659
  terminating 623
FOR loop 623, 659
  increment 661
foreground procedure 105
FREE-VARIABLE 664
FROM-C-LITERAL( ) 383
FROM-X-LITERAL( ) 385
function
  call (performance aspect) 274
  predefined 56, 223, 347, 724
function call 223, 255
function group 233
function result 232

G
gen (suffix for data type) 557
global index 555
GOTO 104, 670
guided dialog 78

H
HASH-STRING( ) 386
HASH-VALUE( ) 387
home pubset, catalog ID 389
HOME-CAT-ID( ) 389
host name, requesting 390
HOST( ) 390
hyphen 53

I
IF 672
IF block 620, 621
  initiating 672
  terminating 625
IF-BLOCK-ERROR 674
IF-CMD-ERROR 676
IF-CMD-ERROR block 71
implicit declaration 139, 159
IMPORT-VARIABLE 678
improve performance 271
include procedure 692
INCLUDE-CMD 683
INCLUDE-PROCEDURE 145, 274, 599, 686
index 555
INDEX( ) 391
initial value 90, 156, 415
  defining 607
input
  data 59
  destination 175
  variable 175
input file 59
input lines
  structure 76
input parameter 225
input source 177
installation 793
installation files 795
INSTALLATION-PATH( ) 396
INTEGER 156, 485
integer 250
  converting 402
integer (data type) 548
INTEGER( ) 399
INTEGER-TO-CHARACTER( ) 401
INTEGER-TO-X-LITERAL( ) 402
interactive mode 287
interrupt 78
interruptibility 84
INTR 294
ISAM files 51
IS-CATALOGED-FILE( ) 406
IS-CATALOGED-JV( ) 409

IS-C-LITERAL( ) 404
IS-DECLARED( ) 411
IS-EMPTY-FILE( ) 413
IS-INITIALIZED( ) 415
IS-INTEGER( ) 417
IS-LIBRARY( ) 419
IS-LIBRARY-ELEMENT( ) 421
ISP commands 287
IS-SDF-LIST( ) 423
IS-SDF-P( ) 424
IS-SDF-STRUCTURE( ) 426
IS-VARIABLE-NAME( ) 428
IS-X-LITERAL( ) 430

J
job
    define attributes 634
    monitoring with job variables 636
    runtime logs 642
    start 637
job attributes 120
job class
    requesting 432
job control 656
job information 432, 434, 466, 522, 525, 526
job monitoring 433
job name
    requesting 434
job number 522
job switch
    replacing 290
job variable 55, 56, 57, 185, 458
    monitor job 636
    requesting 409, 436
job variable function 433, 436
job variable replacement 87, 288
    setting 288
JOB-CLASS( ) 432
JOB-MONJV( ) 433
JOB-NAME( ) 434
JV( ) 436

K
keyword 347
keyword parameters 108, 227, 578, 688
    mixing 109

L
layout scope
    requesting 438
LAYOUT-SCOPE( ) 438
length of procedure lines 51
LENGTH( ) 440
library element
    checking 421
library name
    checking 419
lifetime 157
LIMIT( ) 442
list 62, 138, 140, 169, 174
    declaring 140
list element
    appending 375
    names 152
    selecting 490
list header 140
list size 478
    requesting 442, 492
list variable 49, 138
    reading to 176
    sorting 779
log
    batch job 642
    procedure 642
log contents 113
logging 84, 111, 282
    checking 444
    modifying 283
    S procedure 579, 689
logging a command 112
logging data 112
logging mode 114
LOGGING-MODE( ) 444
LOGOFF 294
LOGON 294
loop 60, 63, 96, 791

loop block 587
loop pass 790
  terminating 587
low (suffix for data type) 552
lowercase letters
  translating 523
LOWER-CASE( ) 446

M
macro
  BIFDEF 305
  BIFDESC 307
  BIFMDL1 309
  BIFMDL2 311
  CLIEXPR 314
  CLIGET 320
  CLISET 323
MAINCODE( ) 448
man (suffix for data type) 557, 558
mandatory (suffix for data type) 558
message 451
message class 448
message code 448, 451
message text 451
  outputting 451
messages
  filter 695, 738
  suppress 87
metasyntax of SDF 542
minus sign 257
mixing procedure lines 62
MOD 259
MODIFY-PROCEDURE-OPTIONS 692
MODIFY-PROCEDURE-TEST-OPTIONS 295, 697
modulo operation 259
MONJV, requesting 433
MONJV program
  requesting 458
MONTH( ) 450
MSG( ) 451
multiple declaration 166
multiplication 257

N
name
  specifying new 460
name (data type) 548
name of month, outputting 450
nesting, command block 54
nesting level, requesting 455
NEXT-VARIABLE-NAME( ) 453
non-S procedures 57, 287, 295, 697
non-S procedures (converting)
  abnormal end of procedure 291
  end of procedure 287, 291
  procedure head 287, 291
non-S tag 55
non-SDF-P commands 293
notational conventions for SDF 542
number 250

O
odd (suffix for data type) 557
OPEN-VARIABLE-CONTAINER 700
operand value
  checking 354
operator 256
  arithmetic 256
  logical 263
  priority 266
optimization capabilities 271
output
  date 366
  day of the week 370
  days difference 371
  destination 178
  explicit command call 373
  message text 451
  name of month 450
  path name 396
  source 178
  source of command 504
  structure value 468
  substring 493
  variable attributes 769
  variable elements 727

output structure
  SHOW-STREAM-ASSIGNMENT 753
  SHOW-VARIABLE-ATTRIBUTES 774
  SHOW-VARIABLE-CONTAINER-ATTR 778

P

parameter
  symbolic 55
parameter name 89
parameter transfer 91, 106
parentheses 269
partial-filename (data type) 549
passing parameters
  performance aspect 275
password specification
  job variable 636
  user ID 635
path name
  outputting 396
path-compl (suffix for data type) 552
pattern
  searching 536
PAUSE 294
plus sign 256
positional parameters 107, 227, 578, 688
  mixing 109
posix-filename (data type) 549
posix-pathname (data type) 549
priority
  define batch job 636, 640
PROCEDURE 293
procedure
  analyzing 131
  appropriate use of compiled procedures 75
  background 291
  call 681
  call (performance aspect) 274
  called in background 121
  calling in the background 120, 631
  compiled procedure 275
  compiling 74
  dialog 287
  exiting 117
  generating 136
  identifying compiled procedures 133
  improve performance 271
  logs 642
  optimize 271
  protecting 126
  resuming 782
  runtime security 137
  starting 631
  terminating 49, 121, 287
  tracing 282
procedure attributes 288
  defaults 734
  modifying 692
  setting 81, 288, 734
procedure call 82
  adapt 291
  converting 289
  in the background 118
  in the foreground 105
  permissible 105
procedure compiler 74
procedure container 49, 106, 686
procedure control
  command compatibility 293
procedure creation 136
procedure execution 132, 697
  continuing 587
  normal 112
procedure head 81
  settings 734
procedure information 373, 444, 455, 467, 483
procedure interpreter 131
procedure interruption 125, 159, 161, 284
procedure line 51, 58, 59, 61, 290
  first character 52
  is read 63
  length 51
procedure logging 782
  modifying 697
procedure nesting 122
procedure parameter 184, 287, 288
  declaring 288, 600
  evaluation 109
  initializing 288

procedure parameter declaration
  starting 600
  terminating 627
procedure record 83
procedure termination 656
procedure termination command 49
procedure test runs 113
processor name, requesting 457
PROCESSOR( ) 457
PROC-LEVEL( ) 455
product structure 18
product-version (data type) 550
PROG-MONJV( ) 458
PROG-NAME( ) 459
program
  call (performance aspect) 277
  protecting 126
program information 459
program manager
  address 312
program monitoring 458
program name
  requesting 459
program unloading
  S procedure 579, 689
programs
  protecting explicitly 127
  protecting implicitly 127
PUTVAR 330

Q
quotes (suffix for data type) 558

R
RAISE-ERROR 704
range of columns
  when searching 392
read password 112
read protection 112
reading in data 59
reading to a list variable 176
README file 24
READ-VARIABLE 705
register 12, assigning 312
relational operator 260
REMARK 294
RENAME( ) 460
REPEAT 716, 717, 721
REPEAT block 99
  initialize 717, 721
  initiating 716
  terminating 790
REPEAT-CMD (command) 717
REPEAT-STMT (command) 721
replace substring 463
REPLACE( ) 463
request
  account number 348
  array index 349
  catalog ID 389
  complex variable level 453
  default catalog ID 482
  error code 448
  host name 390
  job class 432
  job name 434
  job variable 409, 436
  layout scope 438
  list size 492
  maximum list size 442
  MONJV 433
  MONJV program 458
  nesting level 455
  processor name 457
  program name 459
  runtime priority 466
  SDF-P version 467
  statement spin-off 483
  string length 440
  subcode1 486
  subcode2 488
  SYSCMD assignment 495
  SYSDTA assignment 497
  SYSLST assignment 500
  SYSOUT assignment 502
  system identification 499
  system information 506
  system parameters 506

request (cont.)
- system sequence number 477
- task mode 510
- TIAM device type 481
- TIAM station name 480
- time 511
- TSN 522
- user identification 525
- variable attribute 528
- variable element name 381
- variable size 478
- variable type 364

resources 121
return code 332, 448, 486, 488
- generating 704

return to calling level 656
RUN-PRIORITY( ) 466
runtime priority
- requesting 466

runtime security
- procedure 137

S

S procedure
- define attributes 734
- define procedure call 734
- error handling 734
- escape character 734
- format 734
- implicit variable declaration 734
- improve performance 271
- interrupt 734
- job variable replacement 734
- logging 579, 689
- program unloading 579, 689
- test mode 579, 689

S procedures 51, 53, 59, 287, 288, 290, 293, 295
- asynchronous 631
- creating 49

S tag 54
S variable stream 187
- assigning 190, 563
- deleting 194, 616
- outputting 193
- SYSINF 189
- SYSMSG 189
- SYSVAR 189

S variables 135, 184, 185
- implicit declaration 734
- in FHS-TIAM programs 207
- transmitting 192, 783

save variable container 725
SAVE-RETURNCODE 70, 72, 724
scope
- CURRENT 159
- PROCEDURE 159
- TASK 158

screen, output variable elements to 727
SDF interface. commands 287
SDF statement 483
SDF syntax analysis (performance aspect) 272
SDF-P commands 541
- restrictions 295

SDF-P version, requesting 467
SDF-P-VERSION( ) 467
SDF-STRUCTURE-VALUE( ) 468
search
- direction 473
- for string 276
- in a range of columns 392
- string 471

search direction 473
search pattern 536
search string 391
SEARCH-LIST-INDEX( ) 471
select list elements 490
select, variable elements 727
SELECT-VARIABLE-ELEMENTS 727
semicolon 58, 132, 293, 294
SEND-STATEMENT 733
sep (suffix for data type) 557
SESSION-NUMBER( ) 477
SET-JOB-STEP 72, 294
SET-LOGON-PARAMETERS 294
SET-PROCEDURE-OPTIONS 734
SET-VARIABLE 740
SHOWSSA 333
SHOW-STREAM-ASSIGNMENT 750

SHOW-STRUCTURE-LAYOUT 179, 754
SHOW-VAR-CONTAINER-ATTRIBUTES 776
SHOW-VARIABLE 758
SHOW-VARIABLE-ATTRIBUTES 769
SIZE( ) 478
SKIP commands 294
SKIP-COMMANDS 104
slash 52, 131, 258, 290, 732
software configuration 796
SORT-VARIABLE 779
specify new name 460
startup behavior 120
statement 52
statement record
  transferring 733
statement spinoff
  requesting 483
statements
  mark 290
static structure 137
STATION( ) 480
STATION-TYPE( ) 481
STD-CAT-ID( ) 482
STMT-SPINOFF( ) 483
STRING 156, 485
string
  analyzing 423, 426
  checking 471
  comparing 534
  converting 513, 514, 538
  filling 379
  search 276
  searching 391
string analysis 440
string comparison 261
string function 233, 354, 460, 468, 490, 492, 493
string length, requesting 440
string literals 252
string processing 375, 377, 463
STRING( ) 485
structure 138, 143, 174, 573
  declaring 143
  dynamic 144, 664
  static 137, 145, 573
structure declaration
  interruption 146
  terminating 628
structure element 594
  declaring 594
  name 153
structure layout 147, 438, 574, 754
  declaring 573, 599
  outputting 179
structure size 478
structure value, outputting 468
structured output
  ASSIGN-STREAM 197
  EXECUTE-CMD 196
  method 196
structured-name 150
structured-name (data type) 550
structured-variable-name 150
structures 170
subcode1
  error classes 559
  requesting 486
SUBCODE1( ) 486
subcode2, requesting 488
SUBCODE2( ) 488
SUBLIST( ) 490
SUBLIST-NUMBER( ) 492
substring
  outputting 493
  replacing 463
SUBSTRING( ) 493
subtraction 257
suffixes for data types 543, 552
suppress SDF-P message 87
symbolic parameters
  substitute values for 578, 688
syntax 56
syntax description 542
SYSCMD assignment, requesting 495
SYSCMD exits 312
  activation 312
  command conversion 312
SYSCMD( ) 495
SYSDTA assignment, requesting 497

SYSDTA( ) 497
SYSFILE environment 82
SYSFILE information 495, 497, 500
SYS-ID( ) 499
SYSINF 189
SYSINF assignment 564
SYSLST (system file)
    output to printer 642
SYSLST assignment, requesting 500
SYSLST( ) 500
SYSMSG 189
SYSMSG assignment 564
SYSOUT 189
SYSOUT (system file)
    output to printer 642
SYSOUT assignment, requesting 502
SYSOUT( ) 502
SYSPARAM 155
SYSSWITCH 154, 155
system exits 312
system files 237
    delete 642
system ID 499
system identification 499
    requesting 499
system information 477, 499, 504
    requesting 506
system information, request 506
system output, controlling 648
system parameters, request 506
system sequence number, requesting 477
SYSTEM-CALL 504
SYSTEM-CALL( ) 504
SYSTEM-INFORMATION( ) 506
SYSVAR 189
SYSVAR assignment 564

T
tag 54, 293, 670
    compatibility 293
task information 348, 510
task mode, requesting 510
task sequence number 522
TASK-MODE( ) 510
temp-file (suffix for data type) 557
terminating error situations 294
test function 404, 411, 415, 417, 419, 421, 423, 424, 426, 428, 430, 536
test mode, S procedure 579, 689
text (data type) 550
TIAM device type, requesting 481
TIAM station 457, 480, 481
TIAM station name, requesting 480
time, requesting 511
time (data type) 550
TIME( ) 511
TO-C-LITERAL( ) 513
TO-X-LITERAL( ) 514
TRACE-PROCEDURE 782
transferring errors 73, 116, 121
translate Boolean expression 518
translate lowercase letters 523
TRANSLATE( ) 515
TRANSLATE-BOOLEAN( ) 518
TRANSMIT-BY-STREAM 783
TRANSVV 336
TSN, requesting 522
TSN( ) 522
TYPE 294

U
under (suffix for data type) 552
unguided dialog 78
uninterruptibility 126
UNTIL 790
uppercase letters, converting 446
UPPER-CASE( ) 523
usage (performance aspect) 273, 276
user (suffix for data type) 558
user ID 406
user identification, requesting 525
user information 482
user switch 526
    evaluating 526
USER-IDENTIFICATION( ) 525
USER-SWITCH( ) 526

V
value, defining constant 589
value assignment
  variable 167, 705, 740
variable 62
  analyzing 381, 453
  complex 135, 137, 138, 139, 168, 594, 740, 758
  converting 532
  creating 176
  declaring 87
  deleting 173, 618
  expansion 137
  importing 678
  inputting 175
  removing 175
  scope 157
  simple 135, 138, 139, 168, 594, 740
  value assignment 167, 705, 740
variable access 364
variable attribute 235, 236, 438, 442, 528, 589, 594, 607
  output 769
  requesting 528
variable container 162, 174
  closing 581
  displaying open ones 776
  opening 700
  saving 725
variable contents
  deleting 173, 664
  output 758
  sort 779
variable declaration 136, 597, 607
  checking 411
  explicit 137
  with constant value 589
variable element name, requesting 381
variable elements, output and select 727
variable initialization, checking 415
variable name 150, 254, 364
  checking 428
  specifying 175
  syntax 150
variable output 178
variable size
  requesting 478
variable type 138
  requesting 364
VARIABLE-ATTRIBUTE( ) 528
variables 273
VARIABLE-TO-STRING( ) 532
VERIFY( ) 534
vers (suffix for data type) 558
visibility 157
vsn (data type) 550

W
WAIT-EVENT 104, 295
WHILE 791
WHILE block
  initiating 791
  terminating 629
wild(n) (suffix for data type) 553
WILDCARD( ) 536
wild-constr (suffix for data type) 555
with (suffix for data type) 552
with-constr (suffix for data type) 555
with-low (suffix for data type) 552
without (suffix for data type) 557
without-cat (suffix for data type) 557
without-corr (suffix for data type) 557
without-gen (suffix for data type) 557
without-man (suffix for data type) 557
without-odd (suffix for data type) 557
without-sep (suffix for data type) 557
without-user (suffix for data type) 558
without-vers (suffix for data type) 558
with-under (suffix for data type) 552
with-wild(n) (suffix for data type) 553
words
  reserved 154

X

X literal
    checking 430
    converting 385
X string 252
X-LITERAL-TO-INTEGER( ) 538
x-string (data type) 551
x-text (data type) 551