

Synergy Language Reference Manual

Version 9.3



Printed: December 2009

The information contained in this document is subject to change without notice and should not be construed as a commitment by Synergex. Synergex assumes no responsibility for any errors that may appear in this document.

The software described in this document is the proprietary property of Synergex and is protected by copyright and trade secret. It is furnished only under license. This manual and the described software may be used only in accordance with the terms and conditions of said license. Use of the described software without proper licensing is illegal and subject to prosecution.

© Copyright 1997–2009 by Synergex

Synergex, Synergy, Synergy/DE, and all Synergy/DE product names are trademarks of Synergex.

MS-DOS, ActiveX, Windows, Windows Server, and Win32 are registered trademarks of Microsoft Corporation. Windows Vista is a trademark of Microsoft Corporation.

Adobe, Acrobat, and Reader are registered trademarks of Adobe Systems Incorporated.

All other product and company names mentioned in this document are trademarks of their respective holders.

DCN SL-00-9301

Synergex
2330 Gold Meadow Way
Gold River, CA 95670 USA

<http://www.synergex.com>

phone 916.635.7300

fax 916.635.6549

Contents

Preface

About this manual	xxix
Manual conventions	xxix
Other useful publications	xxx
Product support information	xxx
Synergex Professional Services Group	xxx
Comments and suggestions	xxxi

1 Welcome to Synergy Language

Before You Begin Using Synergy Language	1-2
System Philosophy	1-3
What is Synergy Language?	1-3
Program Organization	1-5
Modularity and structured programming	1-5
Bound programs	1-6
System Components	1-8
Compiler	1-8
Linker	1-8
Librarian	1-8
Runtime	1-8
External subroutine, intrinsic function libraries, and APIs	1-9
Indexed Sequential Access Method (ISAM)	1-9
Source-level debugger	1-9
C language interfacing	1-9
Basic Language Elements	1-10
Character set	1-10
Data types	1-10
Statements	1-10
Preprocessor and compiler directives	1-18

Identifiers	1-20
Textual literals	1-22
Variables	1-24
Variable references	1-29
Other System Features	1-35
Data encryption	1-35
Prototyping	1-37
Profiling	1-38
Messaging	1-38
Synergy Language Files	1-42
File types	1-42
File specifications	1-43

2 Defining Data

Data Types	2-2
Rules for Passing One Type to Another	2-15
Where Data Types Can Be Used	2-18
Field definitions	2-20
Parameter definitions	2-33
Return values	2-38
Using Data	2-39
Where fields can be defined	2-39
Data organization	2-39
Data layout	2-43
Lifetime of Data	2-47
Scope	2-47
Characteristics	2-47
Memory management	2-48

3 Expressions

What Is an Expression? 3-2

Truth values 3-2

Operands 3-3

Operators 3-4

Assignment operations 3-4

Alpha operations 3-5

Numeric operations 3-7

Relational operations 3-11

Boolean operations 3-12

Bitwise operations 3-13

Object operators 3-14

Order of evaluation 3-20

More examples of expressions 3-24

Assignment Statements 3-26

Moving alpha data to an alpha destination 3-26

Moving alpha data to a numeric destination 3-27

Moving numeric data to an alpha destination 3-28

Moving decimal/packed data to a decimal/packed destination 3-33

Moving decimal/packed data to an implied-decimal/implied-packed destination 3-34

Moving decimal/packed data to an integer destination 3-34

Moving implied-decimal/implied-packed data to an implied-decimal/implied-packed destination 3-35

Moving implied-decimal/implied-packed data to a decimal/packed destination 3-36

Moving implied-decimal/implied-packed data to an integer destination 3-36

Moving integer data to a decimal/packed destination 3-37

Moving integer data to an implied-decimal/implied-packed destination 3-38

Moving integer data to an integer destination 3-38

Assigning an object handle to another object handle 3-39

Compile-Time Expressions and Optimizations 3-41

Optimizations 3-42

4 Synergy Language Statements

Introduction to Statements 4-6

Statement format 4-6

Data division 4-8

Procedure division 4-8

Compound Statements 4-9

Statement Syntax 4-10

ACCEPT – Receive a character from a specified channel 4-11

ADDHANDLER – Register a method as an event handler for an event 4-13

BEGIN-END – Begin and end a compound statement 4-15

CALL – Call an internal subroutine 4-16

CASE – Select a statement from a set of statements 4-18

CLASS-ENDCLASS – Define a class 4-22

CLEAR – Clear a variable 4-26

CLOSE – Close a channel 4-28

COMMON-ENDCOMMON – Define a shared data record 4-30

DATA – Add a local stack variable declaration 4-33

DECR – Decrement a numeric variable 4-35

DELETE – Delete a record from an ISAM file 4-37

DETACH – Disconnect program from terminal 4-39

DISPLAY – Send output to a device or file 4-40

DO FOREVER – Repeatedly execute a statement 4-44

DO-UNTIL – Repeatedly execute a statement 4-45

END – End a routine 4-46

ENDPARAMS – End parameter list 4-47

ENUM – Declare an enumeration 4-48

EXIT – Terminate a BEGIN-END block 4-51

EXITLOOP – Terminate a loop 4-53

EXITTRY – Exit a TRY statement 4-54

EXTERNAL FUNCTION-ENDEXTERNAL – Declare an external function 4-55

FIND – Find a record 4-57

FLUSH – Flush all buffers 4-61

FOR-DO – Repeatedly execute a statement 4-62

FOR-FROM-THRU – Repeatedly execute a statement 4-64

FOR-UNTIL-DO – Repeatedly execute a statement (alternate) 4-66

FOREACH – Iterate sequentially through elements in a collection 4-68

FORMS – Process ASCII control codes	4-70
FRETURN – Return a value to the calling routine	4-71
FUNCTION-ENDFUNCTION – Define a function	4-72
GET – Receive data from a channel	4-75
GETS – Get sequential binary data	4-77
GLOBAL-ENDGLOBAL – Define a global data section	4-80
GOTO – Transfer program control	4-84
GROUP-ENDGROUP – Define a group	4-86
IF – Conditionally execute a statement	4-90
IF-THEN-ELSE – Conditionally execute one of two statements (alternate)	4-92
IMPORT – Import prototypes	4-94
INCR – Increment a numeric variable	4-96
INIT – Initialize a named data structure to its default values	4-98
LITERAL-ENDLITERAL – Define a literal	4-99
LOCASE – Convert uppercase characters to lowercase	4-101
LPQUE – Queue a file for printing by the system	4-102
MAIN-ENDMAIN – Explicitly define a main routine	4-105
MERGE – Combine two or more files	4-107
METHOD-ENDMETHOD – Define a method	4-112
MRETURN – Return control from a method	4-116
NAMESPACE-ENDNAMESPACE – Define a namespace	4-117
NEXTLOOP – Terminate current loop iteration	4-118
NOP – Perform no operation	4-119
OFFERROR – Disable runtime errors traps	4-120
ONERROR – Enable trapping of runtime errors	4-121
OPEN – Open a channel	4-124
PROC-END – Begin and end the procedure division	4-135
PROPERTY-ENDPROPERTY – Declaring a property	4-136
PURGE – Purge a channel	4-140
PUT – Write fixed-length data	4-141
PUTS – Write sequential fixed-length data	4-143
READ – Read a specified record	4-145
READS – Read the next sequential record	4-150
RECORD-ENDRECORD – Define a data record	4-156
RECV – Receive a message	4-160
REMOVEHANDLER – Unregister a registered event handler	4-162
REPEAT – Repeatedly execute a statement	4-163

RETURN – Return control from a subroutine	4-164
SEND – Send a message	4-165
SET – Assign a value to a set of variables	4-167
SLEEP – Suspend program execution	4-168
SORT – Sort a file	4-169
STOP – Terminate program execution	4-175
STORE – Store a record to an ISAM file	4-177
STRUCTURE-ENDSTRUCTURE – Define how data is laid out	4-179
SUBROUTINE-ENDSUBROUTINE – Define an external subroutine	4-182
THROW – Throw an exception	4-185
TRY-CATCH-FINALLY-ENDTRY – Handle exceptions	4-186
UNLOCK – Release a record lock on a channel	4-189
UPCASE – Convert lowercase characters to uppercase	4-191
USING – Conditionally execute a statement	4-192
USING-RANGE – Conditionally execute a statement with a range	4-194
WHILE – Repeatedly execute a statement	4-196
WRITE – Write a record to a file	4-198
WRITES – Write the next sequential record to a channel	4-202
XCALL – Call an external subroutine or function	4-204
XRETURN – Return control to a calling routine	4-205
Input and Output Statement Qualifiers	4-207
ALLOC – Pre-allocate a number of blocks	4-208
BKTSIZ – Set bucket size	4-209
BLKSIZ – Set block size	4-210
BUFNUM – Set number of process local I/O buffers	4-211
BUFSIZ – Set buffer size	4-212
CONTIG – Allocate contiguity of a file	4-213
DEQ – Set number of blocks to add	4-214
DIRECTION – Set direction of a READS	4-215
FDL – Specify an FDL or XDL string or file specification	4-216
GETRFA – Return a record's RFA	4-218
GUIWND – Open a new application window	4-221
KEYNUM – Specify a key of reference	4-222
LOCK – Control record locking	4-223
MASK – Specify activation characters	4-225
MATCH – Define how a record is located	4-227
NOFILL – Prevent the data area from being padded with blanks	4-229

- NUMREC – Preallocate a number of records 4-230
- OPTIONS – Specify runtime options 4-231
- POSITION – Define the file position 4-233
- RECSIZ – Specify a record size for a file 4-234
- RECTYPE – Define the type of records to write to a file 4-236
- RFA – Locate record with the specified RFA 4-237
- SHARE – Control file access 4-239
- TEMPFILE – Define an intermediate file for output mode 4-241
- WAIT – Specify wait time for an I/O statement to finish processing 4-242
- Record Locking 4-243
 - Optimistic locking technique 4-244
- Record File Addresses (RFAs) 4-246
 - Static RFAs 4-246
 - Global RFAs (GRFAs) 4-247

5 Preprocessor and Compiler Directives

- Types of Directives 5-2
 - Preprocessor directives 5-2
 - Compiler directives 5-2
- Directive Syntax 5-3
 - .ALIGN – Align the data location counter to a boundary 5-4
 - .DEFINE – Define an identifier 5-7
 - .IDENT – Include a module identifier string 5-13
 - .IF-.ELSE-.ENDC – Perform conditional processing 5-16
 - .IFDEF-.ELSE-.ENDC – Perform conditional processing 5-18
 - .IFNDEF-.ELSE-.ENDC – Perform conditional processing 5-20
 - .IFT, .IFF, .ITF – Control conditional processing 5-22
 - .INCLUDE – Include external source code 5-25
 - .LIST – Control the listing of source code 5-35
 - .NODEBUG – Ignore breakpoints in debug mode 5-36
 - .NOLIST – Control the listing of source code 5-37
 - .NOPROTO-.PROTO – Turn prototype generation off and on 5-38
 - .PAGE – Advance program listing to top of new page 5-39
 - .REGION-.ENDREGION – Specify a collapsible block of code 5-40
 - .START – Control the listing of source code 5-41

- .TITLE – Specify the listing page header 5-43
- .UNDEFINE – Remove definition of replacement identifier 5-44

6 Error Handling

- Error Trapping 6-2
 - I/O error lists 6-3
- Structured Exception Handling 6-4
 - Customizing error information 6-4
- Exception Classes 6-6
 - Synergy exception class hierarchy 6-6
 - Synergy exception class error mapping 6-8

7 Understanding Routines

- Writing Subroutines and Functions 7-2
 - Subroutines 7-2
 - Functions 7-4
 - ^VAL functions 7-6
- System-Supplied Subroutines and Functions 7-9
 - System-supplied subroutines 7-9
 - System-supplied functions 7-9
 - Data reference operations 7-10
- Argument Type Mapping 7-12
- How Arguments and Parameters Are Handled 7-14
 - Argument mapping during routine calls 7-15

8 Understanding Objects and Classes

- Namespaces 8-2
 - What is a namespace? 8-2
 - Creating a namespace 8-2
 - Importing a namespace 8-2
- Classes 8-4
 - What is a class? 8-4
 - Creating a class 8-4

Class Members	8-5
Methods, subroutines, and functions	8-5
Class fields, records, groups, and structures	8-7
Properties and indexers	8-8
Delegates	8-9
Enumerations	8-9
Nested classes	8-9
Operators	8-10
Constructors and destructors	8-10
Instantiation and Destruction	8-12
Instantiating and using a class	8-12
Destroying a class instance	8-13
Inheriting Class Functionality and Data	8-15
Accessibility	8-16
Casting	8-17
Boxing	8-19
Sample Object Code	8-21

9 System-Supplied Subroutines, Functions, and Classes

System-Supplied Routines	9-9
^A – Access data as alpha	9-10
%ABS – Return the absolute value of an expression	9-11
ACCHR – Define additional activation characters	9-12
ACESC – Define additional activation characters	9-16
^ADDR – Obtain the address of a data element	9-17
^ARG – Access a parameter as alpha	9-19
^ARGA – Access a parameter as alpha	9-20
^ARGDIM – Return the dimension number of an argument	9-21
^ARGN – Access a parameter as numeric	9-23
^ARGNUM – Return the parameter position	9-24
^ARGTYPE – Return the data type of a parameter	9-25
ASCII – Convert a decimal value into an ASCII character	9-26
ASCR5 – Convert ASCII characters into RAD50 characters	9-28
ASTRST – Restore the contents of work areas used as the result of an AST	9-29
ASTSAV – Save the contents of work areas used as the result of an AST	9-30
%ATRIM – Return a string stripped of trailing blanks	9-31

`^B` – Interpret a string as a binary number 9-32
`BEGFL` – Position to first record 9-34
`%BIN` – Return a binary string representation of a numeric value 9-35
`%BIT_IS_CLR` – Evaluate if a bit is clear 9-37
`%BIT_IS_SET` – Evaluate if a bit is set 9-39
`%BKSTR` – Return the position of the last occurrence of a substring 9-41
`BREAK` – Issue a break to the channel 9-42
`BTOD` – Convert binary values to decimal data 9-43
`%CHAR` – Return an ASCII character for a numeric expression 9-45
`CHNOPN` – Check whether a channel is in use 9-46
`%CHOPEN` – Determine if an I/O channel is open 9-48
`CMDLN` – Return the command line that started your program 9-49
`%CNV_IP` – Convert native integer data to portable form 9-53
`%CNV_PI` – Convert portable integer data to native form 9-54
`COMPRESS` – Compress a data string 9-55
`COPY` – Copy file contents to another file 9-57
`%CPUTIME` – Return the accumulated CPU time 9-59
`CPUTM` – Return the accumulated CPU time 9-60
`CREMBX` – Create a mailbox 9-61
`CVTIIV` – Pass an integer value to a field 9-64
`CVTIZV` – Pass an integer value to a zoned numeric value 9-65
`^D` – Access data as decimal or implied-decimal 9-66
`DACHR` – Disable activation characters 9-67
`DAESC` – Disable activation characters 9-68
`DATA_DECRYPT` – Decrypt ciphertext data 9-69
`DATA_ENCRYPT` – Encrypt plaintext data 9-71
`DATA_SALTIV` – Generate a salt and/or initialization vector 9-73
`^DATATYPE` – Return the data type 9-75
`%DATE` – Return the date 9-77
`%DATECOMPILED` – Return the compilation date and time 9-79
`%DATETIME` – Return the date and time 9-80
`DBL$` subroutines 9-82
`DBL$DEVCLT` – Return the class and type of a device 9-86
`%DBL$EXECUTE_IMAGE_ROUTINE` – Execute a routine in a shared image 9-88
`DBL$SETKRF` – Set the key of reference for the next operation on an ISAM file 9-90

DBL\$SNDOPR – Send a message to the system operator	9-91
%DECML – Convert an ASCII character to its decimal value	9-92
^DEFINED – Determine if a field or parameter has been defined	9-94
DELET – Delete the specified file	9-95
DELMBX – Mark a permanent mailbox for deletion	9-96
^DESCR – Pass an argument by descriptor	9-97
DFLAG – Set runtime option flags	9-98
DTOB – Convert decimal data to binary values	9-101
EMPBUF – Write out modified I/O buffers	9-103
ENDFL – Position the file pointer after the last record of a file	9-104
ENVRN – Identify the current operating system and machine	9-105
%ERLIN – Return the line number of the statement that caused an error	9-107
%ERNUM – Return the number of the most recently trapped error	9-108
%ERR_TRACEBACK – Generate error traceback	9-109
ERRMOD – Return the module in which the most recent error occurred	9-110
%ERROR – Return the number of the most recently trapped error	9-111
ERTXT – Return an error message	9-113
EXEC – Execute a non-Synergy Language program	9-115
EXITE – Exit the routine with a specific error	9-116
EXITERROR – Exit the routine with a specific error	9-118
^F – Access variable as implied-decimal data type	9-119
%FALSE – Return a value of zero	9-120
FATAL – Chain to another program if a fatal error occurs	9-121
FILL – Fill a variable with a pattern	9-123
FILNM – Return the file specification of the open channel	9-124
FLAGS – Set runtime option flags	9-126
FORK – Split the current process	9-131
FREE – Release record locks on a channel	9-133
%FSTAT – Return the value of the last floating point call	9-134
FSTAT – Return the value of the last floating point call	9-137
FXSUBR – Dispatch to a floating-point function	9-138
GETCM – Load the contents of a common area into a variable	9-139
GETDFN – Return the default file specification	9-140
GETFA – Return file attributes	9-141
GETFNM – Return the file specification of the open channel	9-147
GETLOG – Translate an environment variable	9-148
GETRFA – Retrieve the current position in a file	9-150

GLINE – Get a line of input 9-153

GTPPN – Return information about the current process 9-155

%HEX – Return a hexadecimal string representation of a numeric value 9-157

^I – Access data as integer 9-158

%IMPLIED – Convert an expression to zoned numeric implied-decimal 9-159

INITPORT – Initialize a serial port 9-160

%INSTR – Search for a substring within a string 9-165

%INT – Return the whole number in a numeric expression 9-168

%INTEGER – Convert an expression to an integer value 9-169

ISAMC – Create an ISAM file 9-170

ISCLR – Clear an ISAM file 9-179

%ISINFO – Return numeric ISAM file status and key information 9-181

%ISINFOA – Return alpha ISAM file status and key information 9-185

ISKEY – Retrieve ISAM key information 9-189

ISSTS – Return status information about an ISAM file 9-192

JBNAM – Provide the image name of the calling program 9-195

%JBNO – Return the job number for the current process 9-196

%JPERIOD – Return the Julian period value for a date 9-198

%KEYVAL – Extract the value of an index file's key from a record 9-201

KILL – Terminate the calling program 9-203

%LEN – Return the length of the data area 9-204

%LINE – Return the current line number 9-205

LOCALIZE – Localize currency 9-206

^M – Map a structure to a data area 9-208

%MEM_PROC – Implement memory management functions 9-210

MODNAME – Get the name of a routine 9-218

MONEY – Specify the currency symbol 9-220

%NDATE – Return a numeric date 9-221

%NUMARGS – Return the number of arguments passed to a routine 9-223

^O – Interpret a string as an octal number 9-224

%OCT – Return an octal string representation of a numeric value 9-226

OPENELB – Make an ELB's subroutines available to executing program 9-228

%OPTION – Determine if a system option is set 9-230

%PACKED – Convert a numeric expression to packed decimal data 9-232

%PAINT – Return the code for the current paint character 9-233

PAINT – Redefine the paint character 9-234

PAK – Compress numeric data 9-236

PARSE	– Extract components of a file specification	9-239
^PASSED	– Determine whether an argument was passed	9-242
^PASSTHRU	– Access a parameter as its calling type	9-243
POSRFA	– Restore the saved position in an ISAM file	9-244
PURGE	– Delete previous versions of a file	9-247
PUTCM	– Put the contents of a variable into a common area	9-248
QSORT	– Perform an in-memory sort	9-249
R5ASC	– Convert a RAD50 word into ASCII	9-251
RANDM	– Generate a random number	9-253
RCFLG	– Control the receive option flags	9-255
RCVID	– Change the default name referenced by RECV	9-258
%RDLEN	– Return the length of the last record read	9-260
%RDTRM	– Return the terminating character of the last record read	9-261
%RECNUM	– Return the number of the last record processed	9-262
^REF	– Pass an argument by reference	9-263
RENAM	– Rename a file	9-264
%RFA	– Return a record's address	9-267
%RND	– Round a value to the nearest whole number	9-268
%ROUND	– Round a value to the nearest whole number	9-269
%RSIZE	– Return the size of the last record read	9-270
RSTAT	– Return information about the last record read	9-272
RSTATD	– Return numeric information about the last record read	9-274
%RTERM	– Return the terminating character of the last record read	9-275
RUNJB	– Issue a system command or start a concurrent process	9-277
%RVSTR	– Search for the last occurrence of a substring within a string	9-281
S_BLD	– Construct a formatted string	9-283
S_DETAB	– Expand tabs	9-286
S_MATCH	– Match a string	9-288
S_PARSE	– Parse a string and return information about it	9-290
S_WRAP	– Wrap a string	9-294
SDMS_SEL	– Set file type access behavior	9-296
SERIAL	– Return the registration string of the current runtime	9-298
%SET_PRIORITY_CLASS	– Set the priority class for a process	9-299
SETCTL	– Modify the operation of control characters	9-301
SETDFN	– Set the default filename extension	9-303
SETLOG	– Set the value of an environment variable	9-304
SHELL	– Execute the system command processor or create a subprocess	9-306

^SIZE	– Return the size of an expression	9-310
SORT	– Provide a callable interface to DBLSORT	9-312
SPAWN	– Execute a command or program	9-314
%STRING	– Convert a value to a string	9-318
STTY	– Control terminal settings	9-319
%SUCCESS	– Determine if low-order bit is on or off	9-320
%SYN_ATEXIT	– Register a routine to be called when a service shuts down	9-321
%SYN_CHARTOSTR	– Convert C-style character pointer to alpha	9-322
%SYN_ESCAPE_HANDLE	– Escape characters in a string in a handle	9-324
%SYN_FREECHN	– Get the next available channel	9-327
%SYN_GETDIR	– Get the current directory path	9-328
%SYN_GETSTATE	– Retrieve a Synergy state option	9-329
SYN_REPORT EVENT	– Write entries to the event log	9-330
%SYN_SETDIR	– Set the current directory path	9-331
%SYN_SETSTATE	– Set a Synergy state option	9-332
%SYN_SYSERRTXT	– Get the formatted system error message	9-334
%SYN_UNAME	– Get the current username	9-335
%SYN_UNESCAPE_HANDLE	– Unescape escape codes in a string in a handle	9-336
%SYN_XNAME	– Get a function name from its segment address	9-337
%SYSERR	– Return the status code from the last system call	9-338
%SYSID	– Return the ID of the system where the source is being compiled	9-339
TFLSH	– Flush the terminal buffer	9-341
TIME	– Return the current system time	9-343
%TIMEZONE	– Return the time zone of the current machine	9-344
%TNMBR	– Return the terminal number	9-346
TRACEBACK	– Print the current traceback	9-349
%TRIM	– Return the length of a variable without trailing blanks	9-350
%TRIMZ	– Return the length of a variable without trailing blanks	9-352
%TRUE	– Return a value of one	9-354
%TRUNC	– Truncate a numeric expression	9-355
%TRUNCATE	– Truncate a numeric expression	9-356
TT_NAME_TO_NUMBER	– Convert a terminal name to its number	9-357
TT_NUMBER_TO_NAME	– Convert a terminal number to its name	9-358
TTBRDCST	– Enable a program to trap broadcast messages	9-359
TTCHAR	– Return type, lines, and width of a file	9-361

TTFLGS – Set terminal-oriented flags	9-364
TTMBX – Associate a mailbox with a channel opened to a terminal device	9-370
%TTNAME – Return the terminal name	9-371
%TTSTS – Determine whether input is pending	9-372
UNPAK – Decompress numeric data	9-374
%UNSIGNED – Return the unsigned binary value of a numeric expression	9-375
^VAL – Pass an argument by value or declare a value type function	9-376
^VARIANT – Return the value of an internal compiler variable	9-378
%VERSN – Return the Synergy Language version number and operating system	9-379
VMCMD – Execute a DCL command	9-380
VMMSG – Get the text of an OpenVMS system message	9-382
%VMS – Indicate if a program is being compiled on OpenVMS	9-384
%WAIT – Suspend program execution	9-385
%WKDAY – Return the day of the week	9-387
^X – Interpret a string as a hexadecimal number	9-389
%XADDR – Return the address of a routine	9-391
XARGS – Return information about arguments passed to external routine	9-393
XSTAT – Get return value from last routine called	9-394
XSUBR – Invoke an external routine with the specified arguments	9-397
^XTRNL – Provides the value of a link-time symbol	9-400
%ZONED – Convert expression to decimal in zoned numeric representation	9-402
System-Supplied Classes	9-403
System.String	9-404
System.Object	9-406
System.Exception	9-408
System.Array	9-410
Synergex.SynergyDE.SynException	9-412
System.Collections.ArrayList	9-414
Synergex.SynergyDE.Collections.ArrayList	9-417
Synergex.SynergyDE.Select.Select	9-420
Synergex.SynergyDE.Select.From	9-425
Synergex.SynergyDE.Select.Where	9-427
Synergex.SynergyDE.Select.OrderBy	9-437

Synergex.SynergyDE.Select.Event 9-438
Synergex.SynergyDE.Select.AlphaEnumerator and
RestrictedAlphaEnumerator 9-439

10 Synergy ActiveX API

Introduction to the Synergy ActiveX API 10-2

Requirements 10-2

Callback routines 10-2

Workbench support 10-3

UI Toolkit support 10-3

Microsoft Windows support 10-3

Properties 10-4

ActiveX data types 10-6

ActiveX Diagnostic utility 10-7

Licensing considerations 10-7

Debugging 10-8

Sample program 10-9

Synergy ActiveX API Routines 10-12

AX_BIND – Bind a callback subroutine to an ActiveX event 10-13

%AX_CALL – Call a method within an ActiveX control 10-15

%AX_CREATE – Create an ActiveX container 10-16

%AX_DELETE – Delete an ActiveX object 10-18

AX_GET – Get a property value for an ActiveX object 10-19

%AX_GETINT – Get an integer property value for an ActiveX object 10-20

AX_HIDE – Hide an ActiveX container 10-21

%AX_LOAD – Load an ActiveX control into a container 10-22

AX_NEWVAL – Modify an ActiveX callback routine's parameter 10-24

AX_SET – Set a property in an ActiveX object 10-26

AX_SHOW – Show an ActiveX container 10-28

AX_TIMEOUT – Set the time-out value for an ActiveX control 10-29

%WIN_PROC – Process system-level Windows messages 10-30

%WIN_STOP – Suspend Windows system processing 10-31

11 Synergy DLL API

Introduction to the Synergy DLL API 11-2

Synergy DLL API Routines 11-3

%DLL_CALL – Call a DLL routine 11-4

%DLL_CLOSE – Close a Dynamic Link Library 11-9

%DLL_OPEN – Open a Dynamic Link Library 11-10

12 Synergy Floating Point API

Introduction to the Synergy Floating Point API 12-2

Sample program 12-2

Synergy Floating Point API Routines 12-4

FP_ADD – Add floating-point numbers 12-5

FP_ARCCOS – Return the arc cosine of a floating-point number 12-6

FP_ARCSIN – Return the arc sine of a floating-point number 12-7

FP_ARCTAN – Return the arc tangent of a floating-point number 12-8

FP_CMP – Compare floating-point numbers 12-9

FP_COS – Return the cosine of a floating-point number 12-10

FP_DIV – Divide floating-point numbers 12-11

FP_EXPE – Exponentiate a floating-point number 12-12

FP_FROM_NUM – Convert to floating-point from implied-decimal 12-13

FP_LOG10 – Return the base 10 logarithm of a floating-point number 12-14

FP_LOGE – Return the logarithm of a floating-point number 12-15

FP_MUL – Multiply floating-point numbers 12-16

FP_POW – Raise a floating-point number to a floating-point power 12-17

FP_SIN – Return the sine of a floating-point number 12-18

FP_SQRT – Get the square root of a floating-point number 12-19

FP_SUB – Subtract floating-point numbers 12-20

FP_TAN – Return the tangent of a floating-point number 12-21

FP_TO_NUM – Convert from floating-point to implied-decimal 12-22

13 Synergy HTTP Document Transport API

Introduction to the Synergy HTTP Document Transport API 13-2

What is HTTP? 13-2

What is HTTPS? 13-2

Using the HTTP document transport API 13-3

Transmitting data securely 13-5

Error messages 13-11
Sample programs 13-13

Synergy HTTP Document Transport API Routines 13-14

%HTTP_CLIENT_GET – Send a document retrieval request to a server 13-15
%HTTP_CLIENT_POST – Send a request to post a document to a server 13-19
%HTTP_METHOD – Specify a method that prompts for a password 13-23
%HTTP_SERVER_CREATE – Create an HTTP or HTTPS server class
object 13-24
%HTTP_SERVER_DELETE – Free memory for the HTTP or HTTPS
server 13-26
%HTTP_SERVER_RECEIVE – Receive a request from a client 13-27
%HTTP_SERVER_SEND – Send a response to a client 13-30

14 Synergy Symbol Table API

Introduction to the Synergy Symbol Table API Routines 14-2

Requirements 14-2
Access list 14-2
Search list 14-2
Sample program 14-3

Synergy Symbol Table API Routines 14-7

%NSPC_ADD – Add an entry to a symbol table 14-8
%NSPC_ATOS – Map entry in symbol table to search index 14-9
NSPC_CLOSE – Close a symbol table 14-10
NSPC_DELETE – Delete an entry within a symbol table 14-11
%NSPC_FIND – Find an entry in a symbol table 14-12
NSPC_GETDATA – Retrieve an entry's data from a symbol table 14-13
NSPC_MOVE – Move an entry within a symbol table 14-14
%NSPC_OPEN – Open a symbol table 14-15
NSPC_PUTDATA – Modify an entry's data in a symbol table 14-17
NSPC_RESET – Reset a symbol table 14-18
%NSPC_STOA – Map search index to entry in symbol table 14-19

15 Synergy Routine Call Block API

Introduction to the Synergy Routine Call Block API 15-2

Making calls using a routine call block 15-2

Synergy Routine Call Block API Routines 15-5

%RCB_CALL – Call a routine using a routine call block 15-6

%RCB_CREATE – Create a routine call block 15-8

RCB_DELARG – Delete an argument from a routine call block 15-10

RCB_DELETE – Delete a routine call block 15-11

RCB_INSARG – Insert argument into an existing routine call block 15-12

RCB_SETARG – Set an argument in the routine call block 15-15

RCB_SETARGS – Set multiple arguments in a single call 15-19

RCB_SETFNC – Specify a routine to call 15-20

16 Synergy Socket API

Introduction to the Synergy Socket API 16-3

System requirements 16-3

Assumptions 16-3

Differences between Synergy Language sockets and Berkeley Sockets 16-3

Calling the socket functions 16-4

Handling lost sockets 16-5

Synergy socket function errors 16-5

Synergy socket programming examples 16-10

Synergy Socket API Routines 16-14

%SS_ACCEPT – Accept a connection on a stream socket 16-15

%SS_BIND – Bind a name to an unbound socket 16-16

%SS_CLOSE – Close a socket 16-17

%SS_CONNECT – Connect sockets or establish peer address 16-18

%SS_FATAL – Notify server that program is terminating abnormally 16-19

%SS_GETHOSTBYADDR – Get the host name corresponding to an Internet address 16-20

%SS_GETHOSTBYNAME – Get the Internet address corresponding to a host name 16-21

%SS_GETHOSTNAME – Get the name of the local host 16-22

%SS_GETPEERNAME – Return the name of the peer socket 16-23

%SS_GETSERVBYNAME – Return the port number given the service name 16-24

%SS_GETSERVBYPOR – Return the service name given the port number 16-25

%SS_GETSOCKNAME – Return the name associated with a socket 16-26

%SS_HTONL – Convert long integers from host to network byte order 16-27

%SS_HTONS – Convert short integers from host to network byte order 16-28

%SS_INET_ADDR – Convert a dotted text form Internet address 16-29

%SS_INET_NTOA – Convert a numeric Internet address 16-30

%SS_LISTEN – Listen for connections 16-31

%SS_NTOHL – Convert long integers from network to host byte order 16-32

%SS_NTOHS – Convert short integers from network to host byte order 16-33

%SS_RECV – Receive bytes from a socket 16-34

%SS_RECVBUF – Receive bytes from a socket into a buffer 16-36

%SS_RECVFROM – Receive bytes from a datagram socket 16-38

%SS_SELECT – Wait for data 16-40

%SS_SEND – Send bytes to a connected peer or established peer address 16-41

%SS_SENDBUF – Send bytes from a socket buffer to a connected peer 16-43

%SS_SENDTO – Send bytes through a datagram socket 16-45

%SS_SOCKET – Return a socket descriptor 16-47

17 Synergy Windows Printing API

Introduction to the Synergy Windows Printing API 17-2

Synergy Windows Printing API Routines 17-4

Recommended calling sequence 17-4

Sample programs 17-7

%WPR_EXECUTE – Execute printing 17-15

%WPR_GETDEVICE – Retrieve printer settings 17-20

%WPR_INFO – Set and retrieve extended device capabilities 17-27

%WPR_PRINT – Perform print operation to selected printer 17-36

%WPR_SETDEVICE – Set printer settings 17-42

18 Synergy XML API

Synergy XML API 18-4

XML documents 18-4

DOM trees 18-6

XML API routines 18-7

Parsing and processing XML data 18-13

Assembling Synergy data in XML format	18-16
Synergy XML API defines	18-17
%XML_ATTR_COPY – Copy from one XML attribute to another	18-19
%XML_ATTR_CREATE – Create an XML attribute	18-20
%XML_ATTR_DELETE – Release an XML attribute	18-21
%XML_ATTR_GETNAME – Get the name of an XML attribute	18-22
%XML_ATTR_GETVALUE – Get the value of an XML attribute	18-23
%XML_ATTR_GETVALUEHANDLE – Return the handle for an XML attribute's value	18-24
%XML_ATTR_SETNAME – Set the name of an XML attribute	18-26
%XML_ATTR_SETVALUE – Set the value of an XML attribute	18-27
%XML_ATTR_SETVALUEHANDLE – Assign a handle as an XML attribute's value	18-28
%XML_ATTRLIST_ADD – Add an attribute to an XML attribute list	18-30
%XML_ATTRLIST_ADDLIST – Add a list of attributes to an XML attribute list	18-31
%XML_ATTRLIST_CLEAR – Clear XML attribute list contents	18-33
%XML_ATTRLIST_COPY – Copy attributes from one XML attribute list to another	18-34
%XML_ATTRLIST_COUNT – Count the attributes in an XML attribute list	18-35
%XML_ATTRLIST_CREATE – Create an XML attribute list	18-36
%XML_ATTRLIST_DELETE – Release an XML attribute list	18-37
%XML_ATTRLIST_FIND – Find an attribute in an XML attribute list by name	18-38
%XML_ATTRLIST_INSERT – Insert an attribute at a specific position in a list	18-39
%XML_ATTRLIST_ITEM – Find an attribute in an XML attribute list by position	18-41
%XML_ATTRLIST_REMOVE – Remove an attribute from an XML attribute list	18-42
%XML_DOC_ADDDECLARATION – Add a declaration to an XML document	18-43
%XML_DOC_CREATE – Create an XML document instance	18-45
%XML_DOC_DELETE – Release an XML document instance	18-46
%XML_DOC_GETDECLARATION – Get the declaration for an XML document	18-47
%XML_DOC_GETROOT – Return the root element of an XML document	18-48
%XML_DOC_REMOVEDDECLARATION – Remove a declaration from an XML document	18-49

%XML_DOC_SETCOMMENT – Set a comment at the XML document level	18-50
%XML_DOC_SETDECLARATION – Set the declaration for an XML document	18-51
%XML_DOC_SETROOT – Set the root element of an XML document	18-52
%XML_DOC_TOFILE – Write contents of an XML document to a file	18-53
%XML_DOC_TOSTRING – Write contents of an XML document to an XML string	18-54
%XML_ELEM_ADDATTRIBUTES – Add attributes in a list to an XML element	18-55
%XML_ELEM_ADDCHILD – Add a child to an XML element	18-57
%XML_ELEM_ADDCHILDREN – Add child elements to an XML element	18-58
%XML_ELEM_ATTRIBUTES – Return attributes for an XML element	18-60
%XML_ELEM_CHILDREN – Get children of an XML element	18-61
%XML_ELEM_COPY – Copy from one XML element to another	18-62
%XML_ELEM_CREATE – Create an XML element instance	18-63
%XML_ELEM_DELETE – Release an XML element instance	18-64
%XML_ELEM_GETATTRIBUTE – Get the value of an attribute for an XML element	18-65
%XML_ELEM_GETELEMENTSBYNAME – Get children of an XML element by tag	18-66
%XML_ELEM_GETNAME – Get the tag name of an XML element	18-67
%XML_ELEM_GETTEXT – Get the text value of an XML element	18-68
%XML_ELEM_GETTEXTHANDLE – Return handle containing text for an element	18-69
%XML_ELEM_REMOVEATTRIBUTE – Remove an attribute from an XML element	18-71
%XML_ELEM_REMOVEATTRIBUTES – Remove all attributes from an XML element	18-72
%XML_ELEM_REMOVECHILD – Remove the first child specified from an element	18-73
%XML_ELEM_REMOVECHILDREN – Remove all child elements from an element	18-74
%XML_ELEM_SETATTRIBUTE – Add an attribute to an XML element	18-75
%XML_ELEM_SETNAME – Set the tag name of an XML element	18-76
%XML_ELEM_SETTEXT – Set the text value of an XML element	18-77
%XML_ELEM_SETTEXTHANDLE – Assign a handle as the text for an XML element	18-78

%XML_ELEM_TOSTRING – Give the string representation for an XML element	18-80
%XML_ELEMLIST_ADD – Add an element to an XML element list	18-81
%XML_ELEMLIST_ADDLIST – Add a list of elements to an XML element list	18-82
%XML_ELEMLIST_CLEAR – Clear XML element list contents	18-84
%XML_ELEMLIST_COPY – Copy an XML element list to another list	18-85
%XML_ELEMLIST_COUNT – Count the elements in an XML element list	18-86
%XML_ELEMLIST_CREATE – Create an XML element list	18-87
%XML_ELEMLIST_DELETE – Release an XML element list	18-88
%XML_ELEMLIST_FIND – Find an element in an XML element list by name	18-89
%XML_ELEMLIST_FINDTAG – Find an element in an XML element list by tag	18-90
%XML_ELEMLIST_INSERT – Insert an XML element at a specific position in a list	18-91
%XML_ELEMLIST_ITEM – Find an element in an XML element list by position	18-93
%XML_ELEMLIST_REMOVE – Remove an element from an XML element list	18-94
%XML_OPTION – Set an XML option value	18-95
%XML_PARSER_CREATE – Create an XML parser instance	18-97
%XML_PARSER_DELETE – Release an XML parser instance	18-98
%XML_PARSER_ERROR – Get the last XML parser error	18-99
%XML_PARSER_PARSEFILE – Parse an XML file	18-101
%XML_PARSER_PARSESTRING – Parse an XML string	18-102
%XML_STRING_APPEND – Append a Synergy alpha string to an XML string	18-103
%XML_STRING_APPENDDDYN – Append one XML string to another	18-104
%XML_STRING_APPENDHANDLE – Append Synergy handle contents to an XML string	18-105
%XML_STRING_CLEAR – Clear the contents of an XML string	18-106
%XML_STRING_COPY – Copy one XML string to another	18-107
%XML_STRING_CREATE – Create an XML string instance	18-108
%XML_STRING_DELETE – Release an XML string instance	18-109
%XML_STRING_GETHANDLE – Get the memory handle of an XML string	18-110
%XML_STRING_GETSIZE – Get the size of an XML string	18-111

19 Synergy Windowing API

General Windowing Concepts	19-2
External subroutines and their functions	19-2
Window IDs and names	19-2
Window I/O	19-3
Screen updates	19-3
Window areas	19-3
Window placement and occlusion	19-4
Overlay windows	19-4
Terminal I/O	19-4
Attributes	19-5
Colors and the color palette	19-5
Windowing subroutines on Windows platforms	19-10
Windowing API and UI Toolkit	19-10
Window definitions	19-11
Hardware scrolling	19-11
User data sets	19-12
The Windowing Routines	19-13
Attributes	19-13
W_AREA – Manipulate the processing and display areas	19-16
W_BRDR – Control window borders, titles, and their attributes	19-26
W_CAPTION – Retrieve or load the application window caption	19-33
W_DISP – Control window display and input functions	19-34
W_EXIT – Exit the windowing system	19-47
W_FLDS – Manipulate field sets within a window	19-48
W_INFO – Get information about the windowing system	19-59
%W_INFO – Get information about the windowing system	19-68
W_INIT – Initialize the windowing system	19-73
W_PROC – General window processing	19-75
W_RESTORE – Restore a window that was previously compressed	19-93
W_SAVE – Save a window in compressed form	19-95
W_UPDT – Update the physical screen	19-97
Error Processing in the Windowing System	19-98
Trapping errors	19-98
Window errors	19-100

Converting Your Code to Use the Windowing API 19-101

Converting your code 19-101

Updating the screen 19-102

Examples 19-102

20 Synergy .NET Assembly API

Introduction to the Synergy .NET Assembly API 20-2

Requirements 20-2

Callback routines 20-3

Error handling 20-3

Using .NET assemblies referenced by the Synergy .NET assembly API 20-4

Workbench support 20-5

UI Toolkit support 20-5

Debugging 20-5

Sample programs 20-6

Synergy .NET Assembly API Methods 20-9

DotNetAssembly – Load an assembly 20-10

DotNetObject – Create an instance of a .NET type 20-12

DotNetObject.Call – Invoke methods of an instance 20-13

DotNetObject.SetProperty – Set the value of a property on an instance 20-15

DotNetObject.GetProperty – Query the value of a property 20-17

DotNetObject.SetField – Set the value of a field on an instance 20-19

DotNetObject.GetField – Query the value of a field 20-21

DotNetObject property indexer set method 20-23

DotNetObject property indexer get method 20-24

DotNetDelegate – Create an instance of a delegate 20-25

DotNetObject.AddDelegateToEvent – Associate a delegate with an event 20-27

DotNetObject.RemoveDelegateFromEvent – Remove a delegate from an event 20-28

21 Synergy Language C Interface

Introduction to the C Interface 21-3

Passing Arguments to C Routines from Synergy Language 21-5

C Routine Processing 21-6

Runtime Interface Functions 21-8

Example using the Synergy Language C interface 21-8

dbl_exec – Execute a routine within a Synergy Language program 21-11

dblerror – Issue a trappable Synergy Language error 21-12

dblfatal – Issue a nontrappable Synergy Language error 21-13

get_chn_sts – Get the status of a channel 21-14

get_errnum – Convert an error name to a Synergy Language error number 21-15

get_gbl_ptr – Get the address of a named global data section 21-16

get_seg_ptr – Get the address of a routine's control segment 21-17

get_xarg – Get a subroutine argument's descriptor 21-18

get_xarg_cstr – Get a subroutine argument as a null-terminated C string 21-19

get_xarg_fval – Get a subroutine argument as a floating-point value 21-20

get_xarg_val – Get a subroutine argument as an integer value 21-21

put_descr – Perform a Synergy Language store operation 21-22

put_xarg – Load data into a subroutine argument 21-23

put_xarg_fval – Load a floating-point value into an argument 21-24

put_xarg_str – Load a string into a subroutine argument 21-25

put_xarg_val – Load a value into an argument 21-26

Building a Dynamic Library with C Routines 21-27

Appendix A: Maximums

Appendix B: ASCII Character Set

Appendix C: Terminal Keys with Special Functions

Glossary

Index

Preface

About this manual

This manual is your complete reference guide to Synergy Language. It is written for those users who are already familiar with programming, although not necessarily with Synergy Language.

The information in this manual applies to all operating systems on which Synergy Language is available, unless otherwise noted. You will be able to create system-independent programs using this manual. For additional information about system-specific characteristics, refer to the *Professional Series Portability Guide*.

Manual conventions

Throughout this manual, we use the following conventions:

- ▶ In code syntax, text that you type is in `Courier` typeface. Variables that either represent or should be replaced with specific data are in *italic* type.
- ▶ Optional arguments are enclosed in *[italic square brackets]*. If an argument is omitted and the comma is outside the brackets, a comma must be used as a placeholder, unless the omitted argument is the last argument in a subroutine. If the comma is inside the brackets and an argument is omitted, the comma may also be omitted.
- ▶ Arguments that can be repeated one or more times are followed by an ellipsis...
- ▶ A vertical bar (|) in syntax means to choose between the arguments on each side of the bar.
- ▶ Data types are **boldface**. The data type in parentheses at the end of an argument description (for example, **(n)**) documents how the argument will be treated within the routine. An **a** represents alpha, a **d** represents decimal or implied-decimal, an **i** represents integer, and an **n** represents numeric (which means the type can be **d** or **i**).

WIN

Items or discussions that pertain only to a specific operating system or environment are called out with the name of the operating system.

Other useful publications

- ▶ Synergy Language release notes (**REL_DBL.TXT**)
- ▶ *Synergy Language Tools*
- ▶ *Environment Variables and System Options*
- ▶ *Getting Started with Synergy/DE*
- ▶ *Professional Series Portability Guide*

Product support information

If you cannot find the information you need in this manual or in the publications listed above, you can call the Synergy/DE™ Developer Support department at (800) 366-3472 (in North America) or (916) 635-7300. To purchase Synergy/DE Developer Support services, contact your Synergy/DE account manager at the above phone numbers.

Before you contact us, make sure you have the following information:

- ▶ The version of the Synergy/DE product(s) you are running.
- ▶ The name and version of the operating system you are running.
- ▶ The hardware platform you are using.
- ▶ The error mnemonic and any associated error text (if you need help with a Synergy/DE error).
- ▶ The statement at which the error occurred.
- ▶ The exact steps that preceded the problem.
- ▶ What changed (for example, code, data, hardware) before this problem occurred.
- ▶ Whether the problem happens every time, and whether it is reproducible in a small test program.
- ▶ Whether your program terminates with a traceback, or whether you are trapping and interpreting the error.

Synergex Professional Services Group

If you would like assistance implementing new technology or would like to bring in additional experienced resources to complete a project or customize a solution, Synergex™ Professional Services Group (PSG) can help. PSG provides comprehensive technical training and consulting services to help you take advantage of Synergex's current and emerging technologies. For information and pricing, contact your Synergy/DE account manager at (800) 366-3472 (in North America) or (916) 635-7300.

Comments and suggestions

We welcome your comments and suggestions for improving this manual. Send your comments, suggestions, and queries, as well as any errors or omissions you've discovered, to doc@synergex.com.

1

Welcome to Synergy Language

This chapter provides general information about Synergy Language and the structure of Synergy Language programs. It introduces the statements that you will use to build programs and describes the basic elements of the language.

Before You Begin Using Synergy Language 1-2

Suggests how new and existing users of Synergy Language should approach this manual.

System Philosophy 1-3

Describes the theory behind Synergy Language and discusses Synergy Language as the foundation for Synergy/DE.

Program Organization 1-5

Explains how Synergy Language programs are organized and where you can define your data; discusses the use of structured, modular programming in Synergy Language; defines “binding” and describes the features of binding and bound applications.

System Components 1-8

Lists and briefly describes the primary components of Synergy Language.

Basic Language Elements 1-10

Describes the following elements of Synergy Language: the ASCII character set; data types; the Synergy Language statements, organized by type; identifiers; textual literals; variables, including path specifications and arrays; and variable references.

Other System Features 1-35

Discusses data encryption, strong prototype validation, profiling, and the Synergy Language messaging facilities.

Synergy Language Files 1-42

Describes the types of files used by Synergy Language.

Before You Begin Using Synergy Language

Thank you for choosing Synergy Language, the most efficient, portable business language on the market.

If you're a new user of Synergy Language...

If you are a new user of Synergy Language, we suggest you read the remainder of this chapter before you begin programming. You may also want to thumb through [chapter 4, “Synergy Language Statements,”](#) and [chapter 9, “System-Supplied Subroutines, Functions, and Classes,”](#) to familiarize yourself with the available statements, subroutines, and functions and where they are located in the manual.

If you've used a previous version of Synergy Language...

If you're already experienced with a previous version of Synergy Language, please refer to the [Updating Synergy/DE](#) manual and your Synergy Language online release notes (**REL_DBL.TXT**). These documents list the changes we have made to Synergy Language since previous versions. You can then refer to the *Synergy Language Reference Manual* for more information about the changes in which you are especially interested. Several changes may affect your existing code.

System Philosophy

Synergex believes that computer programmers should be able to spend their time creatively designing applications without having to work around unnatural constraints imposed by the programming language. The structured programming elements of the Synergy Language enable programmers to express solutions quickly in straightforward, logical code. Like most programmers, we think you'll find that structured programming improves program efficiency, accuracy, and longevity.

Synergex also believes that the operational characteristics of a programming language should remain consistent across operating environments. Synergy Language enables you to program on one system and then move your programs to other systems easily: you need only recompile and relink to run the programs on the new system. The only time you'll need to change a program is if you designed your original applications expressly for a particular operating system and relied on facilities only available on that system.

Synergy™ ISAM files that contain noninteger data are portable across operating systems. (Note that on OpenVMS, we use the RMS ISAM facility.) If you choose to use integer data, your ISAM files are portable among two groups of machines; see [%CNV_IP on page 9-53](#) and [%CNV_PI on page 9-54](#) for information about moving between the groups. Synergy Language thus enables you to reach a greater market without having to maintain a different version of program sources for each operating system.

What is Synergy Language?

Synergy Language is at the heart of a complete application development environment: Synergy/DE. This environment provides the tools you need to build a successful business application with greater speed and flexibility. It includes the following:

- ▶ Professional Series
 - ▶ Synergy Language (described in this manual)
 - ▶ Workbench
 - ▶ UI Toolkit (including Composer on Windows)
 - ▶ Repository
 - ▶ ReportWriter
- ▶ *x/*Series
 - ▶ *x/*ServerPlus and *x/*NetLink
 - ▶ *x/*Server
- ▶ Connectivity Series
 - ▶ *x/*ODBC
 - ▶ SQL Connection

Welcome to Synergy Language

System Philosophy

You can write state-of-the-art applications using Synergy Language and UI Toolkit; maintain your data structures and fields with Repository; design your own columnar, ad-hoc reports with ReportWriter (or integrate it into your own applications); access business logic and data remotely with *x/ServerPlus*, *x/NetLink*, and *x/Server*; access Synergy data from ODBC-enabled applications with *x/ODBC*; and access RDBMS data with SQL Connection. You can develop these applications using Workbench, a comprehensive visual development environment. All of these tools work together to make your job easier and more productive.

Synergex used the ANS DIBOL 1992 standard (Document X3J12/91-017) as a guideline for this version of Synergy Language.

Program Organization

A Synergy Language program contains one or more imports, structures, namespaces, classes, and routines. A Synergy Language routine contains two main parts: a data division and a procedure division. The data division defines most of the data structures that will be used in the program. The procedure division contains the executable statements and defines the processing algorithms, but you can also define local stack variables there if desired. The PROC statement indicates the end of the data division and the start of the procedure division.

You can define data externally in the S/DE Repository, and reference it in your program's data division using the .INCLUDE compiler directive. (Refer to the *Repository User's Guide* for more information about Repository.)

Modularity and structured programming

Synergy Language encourages structured, modular programming, which divides a large program into smaller subroutines, functions, classes, and methods. The main routine is the main entry point into the program. Subroutines, functions, and methods are either called from the main routine or from other subroutines, functions, and methods. An external subroutine begins with the SUBROUTINE statement, a function begins with the FUNCTION statement, and a method inside a class begins with the METHOD statement. (See [FUNCTION-ENDFUNCTION on page 4-72](#), [SUBROUTINE-ENDSUBROUTINE on page 4-182](#), and [METHOD-ENDMETHOD on page 4-112](#) for more details.) An internal subroutine begins with a label, and its code is part of the same program as the routine that calls it.

A Synergy Language program may contain multiple subroutines, functions, or methods and one main routine. Each main routine, subroutine, function, and method must end with an END, ENDMETHOD, ENDSUBROUTINE, ENDFUNCTION, or ENDMETHOD statement.

An explicit main routine looks like this:

```
main
    Data division statements (optional)
    .
    .
    .
proc
    Procedure division statements
    .
    .
    .
endmain
```

In most cases the MAIN statement is optional. (A main routine without "MAIN" is an *implicit main*.) However, if the first thing in the data division is a STRUCTURE declaration, or if the main routine immediately follows an IMPORT statement, the MAIN keyword is required.

A subroutine or function looks like this:

```
subroutine mysub or function myfunc
    Data division statements (optional)
    .
    .
    .
proc
    Procedure division statements
    .
    .
    .
endsubroutine or endfunction
```

A method looks like this:

```
class myclass
    method mymethod ,void
        Data division statements (optional)
        .
        .
        .
    proc
        Procedure division statements
        .
        .
        .
    endmethod
endclass
```

Bound programs

Binding is a method of grouping multiple main routines into one executable program. On Synergy standard, you can combine several Synergy Language main routines into a single, bound program that can be executed using the **dbf** command. Binding and bound applications have the following features:

- ▶ A bound application has only one entry point; the rest of the main routines are considered “super subroutines.”
- ▶ Binding reduces chaining overhead (which on OpenVMS is quite high).
- ▶ Binding reduces the size of the executable file, because only one copy of the utility subroutines is used, instead of a separate copy for each individual main routine. (Note that ELBs and shared images also provide this feature.)
- ▶ All commons, global records, and global literals are loaded into one data area.

- ▶ All class definitions are combined into one table.
- ▶ Binding reduces class reference overhead after a chain on classes that have already been referenced.

Commons, global data sections, and classes must individually match other commons, global data sections, and classes of the same name. Only one GLOBAL COMMON and one GLOBAL DATA SECTION, INIT is allowed in a bound program.

See “[Binding](#)” in the “Building and Running Synergy Language Programs” chapter of *Synergy Language Tools* for instructions on creating a bound program.

System Components

The primary components of Synergy Language are as follows:

- ▶ Compiler
- ▶ Linker
- ▶ Librarian
- ▶ Runtime
- ▶ External subroutine and intrinsic function libraries
- ▶ Indexed Sequential Access Method (ISAM)
- ▶ Source-level debugger
- ▶ C language interfacing
- ▶ Windowing routines

Compiler

The compiler translates your source files containing Synergy Language statements into object files containing system-level information. The [“Building and Running Synergy Language Programs”](#) chapter of *Synergy Language Tools* explains how to compile programs on your operating system.

Linker

The linker combines the object files created by the compiler to produce executable code files. The [“Building and Running Synergy Language Programs”](#) chapter of *Synergy Language Tools* contains information about linking object files on your operating system.

Librarian

The librarian creates and maintains object libraries. See the [“Building and Running Synergy Language Programs”](#) chapter of *Synergy Language Tools* for information about the librarian.

Runtime

The Synergy runtime executes the actions requested by your Synergy Language programs. Instructions for invoking the runtime are provided in the [“Building and Running Synergy Language Programs”](#) chapter of *Synergy Language Tools*.

External subroutine, intrinsic function libraries, and APIs

Synergy Language includes a collection of external subroutines, intrinsic functions, and APIs. (The external subroutines and intrinsic functions are described in [chapter 9, “System-Supplied Subroutines, Functions, and Classes.”](#)) The APIs are described in [chapter 10](#) through [chapter 19](#).

Indexed Sequential Access Method (ISAM)

ISAM provides a powerful file structure that allows keyed as well as sequential record processing. Synergy Language includes a segmented, multiple-key ISAM system that reuses released data space, eliminating the need for frequent ISAM file reorganization. (Synergy ISAM file structure and utilities are described in the “[Synergy DBMS](#)” chapter of *Synergy Language Tools*.)

Source-level debugger

The symbolic source-level debugger enables programs to run in a debugging mode, which is useful during program development or when a problem is discovered in a production program. This productivity tool helps you to better understand program behavior. (See the “[Debugging Your Synergy Programs](#)” chapter in *Synergy Language Tools* for more information about the debugger.)

C language interfacing

The Synergy Language C interface enables Synergy Language routines to call C language routines and vice versa. (For more information on accessing C language routines from your Synergy Language programs, see [chapter 21, “Synergy Language C Interface.”](#))

Basic Language Elements

Character set

The basic unit of Synergy Language is the ASCII character. ([Appendix B](#) contains the complete ASCII character set.) ASCII characters are combined into sequences that represent the elements of Synergy Language. Certain characters, called delimiters, separate adjacent language elements in a statement. Note that identifier names and keywords are case insensitive in Synergy Language.

This manual refers to the following ASCII character subsets:

Numeric: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9

Alphabetic: Uppercase characters **A** through **Z** and lowercase characters **a** through **z**

Alphanumeric: Both numeric and alphabetical characters

Alpha: All ASCII characters except the nonprintable characters (for example, carriage return, line feed, and form feed)

Data types

Synergy Language supports all of the data types listed in [“Data Types” on page 2-2](#).

Statements

Synergy Language provides six types of statements:

- ▶ Declarative
- ▶ Data manipulation
- ▶ File manipulation
- ▶ Control
- ▶ Input/output
- ▶ Interprogram communication

This section is merely intended as an introduction to Synergy Language statements. You will find more complete information in [chapter 4, “Synergy Language Statements.”](#)

In this manual, we refer to the statement name as a “keyword.”

Declarative statements

Declarative statements declare the variables and definitions that the routine can access. Many, but not all, declarative statements appear in the data division or class definition of your program. The declared variables can be local to the routine, global to the routine, or used to exchange data with other routines.

CLASS-ENDCLASS

CLASS defines a class within the current namespace or within another class.

COMMON-ENDCOMMON

COMMON defines a shared data record and its associated fields. A GLOBAL COMMON statement allocates a shared data area. An EXTERNAL COMMON statement references a data area that was allocated by a GLOBAL COMMON statement.

DATA

DATA defines a local stack variable in the procedure division.

EXTERNAL FUNCTION-ENDEXTERNAL

EXTERNAL FUNCTION declares a user-defined or system-supplied external function.

FUNCTION-ENDFUNCTION

FUNCTION indicates the beginning of a user-defined function.

GLOBAL-ENDGLOBAL

GLOBAL specifies that all RECORD definitions that follow the GLOBAL statement are stored in a global named data area instead of the local data area. ENDGLOBAL designates the end of a global data section.

GROUP-ENDGROUP

GROUP defines a data structure within another data structure. A group member can be a field or another group. ENDGROUP designates the end of a group.

IMPORT

IMPORT imports the prototypes in the specified namespace to enable you to use any members of that namespace.

LITERAL-ENDLITERAL

LITERAL defines a data structure and its associated fields that cannot be modified during program execution.

MAIN-ENDMAIN

MAIN indicates the beginning of a main routine.

METHOD-ENDMETHOD

METHOD indicates the beginning of a method.

NAMESPACE-ENDNAMESPACE

NAMESPACE creates a namespace, which enables you to group related entities together for use in a Synergy application.

PROC-END

PROC indicates the end of the data division and the beginning of the procedure division. END designates the end of the procedure division.

PROPERTY-ENDPROPERTY

PROPERTY indicates the beginning of a property, which must contain a get or set accessor method.

RECORD-ENDRECORD

RECORD defines a data structure and its associated data fields. ENDRECORD designates the end of a record.

STRUCTURE-ENDSTRUCTURE

STRUCTURE defines how data is laid out, without allocating any data space.

SUBROUTINE-ENDSUBROUTINE

SUBROUTINE indicates the beginning of an external subroutine.

Data manipulation statements

Data manipulation statements directly change data.

CLEAR

CLEAR initializes one or more variables to either blanks (for alpha variables) or zero representation (for numeric variables).

DECR

DECR decrements a numeric variable by 1.

INCR

INCR increments a numeric variable by 1.

INIT

INIT initializes a named data structure to its declared default values.

LOCASE

LOCASE converts all uppercase characters in an alphabetic variable to their corresponding lowercase characters.

SET

SET initializes a set of variables of the same data type to a single value.

UPCASE

UPCASE converts all lowercase characters in an alphabetic variable to their corresponding uppercase characters.

File manipulation statements

File manipulation statements access records within a file or group of files.

MERGE

MERGE merges the contents of two or more sorted files into a single file. Synergy Language supports an in-place merge and an *n*-way merge.

SORT

SORT orders a file by specified key fields and places the sorted result in either a new file or the original file.

Control statements

When a routine is being executed, control begins with the statement that follows the PROC statement and flows sequentially to each successive statement. Control statements alter the order of execution by directing control to another statement, transferring control to another routine, ending processing of the current routine, or returning control to a referencing routine. This type of statement is used to create loops and internal functions within a program.

BEGIN-END

The BEGIN and END statements define a compound statement, which is logically considered to be a single statement.

CALL

CALL transfers control to an internal subroutine until a corresponding RETURN statement is executed. Synergy Language provides both an unconditional form and a computed form, which transfers control based on the result of an expression.

CASE

CASE executes one of a set of statements based on the value of a selection expression.

DO FOREVER

DO FOREVER executes a statement repeatedly until a condition occurs to transfer control outside of the statement. (DO FOREVER works just like the REPEAT statement.)

DO-UNTIL

DO-UNTIL executes a statement repeatedly until the specified condition becomes true. The DO loop is always executed at least once.

EXIT

EXIT terminates execution of the current BEGIN-END compound statement. Control is transferred to the END statement of the current compound statement.

EXITLOOP

EXITLOOP terminates execution of the current loop. Control is transferred to the statement that immediately follows the loop.

EXITTRY

EXITTRY exits a TRY statement.

FOR

FOR executes a statement repeatedly under the control of a specified variable. FOR has three forms:

- ▶ **FOR-DO**
- ▶ **FOR-FROM-THRU**
- ▶ **FOR-UNTIL-DO**

FRETURN

FRETURN specifies the return value of a user-defined function and returns control to the calling program.

GOTO

GOTO transfers program control to the specified label. Synergy Language provides both an unconditional form and a computed form, which transfers control based on the result of an expression.

IF

IF executes a statement based on the result of an expression.

IF-THEN-ELSE

The IF-THEN-ELSE sequence executes one of two statements based on the result of an expression.

MRETURN

MRETURN returns control from a method to the calling program.

NEXTLOOP

NEXTLOOP terminates the current iteration of a loop and moves to the start of the next iteration. (NEXTLOOP performs like the EXIT statement within the BEGIN-END block of a loop structure.)

NOP

NOP performs no operation. It is ignored.

OFFERROR

OFFERROR disables any runtime error trapping established by an ONERROR statement.

ONERROR

ONERROR enables program trapping of one or more runtime errors.

REPEAT

REPEAT executes a statement repeatedly until control is explicitly transferred outside of the statement. (REPEAT works just like the DO FOREVER statement.)

RETURN

RETURN completes the processing of an internal subroutine and returns control to the statement that follows the CALL that invoked the subroutine.

SLEEP

SLEEP suspends execution of a routine for the specified number of seconds.

STOP

STOP terminates execution of the current program. Synergy Language provides two forms of this statement: an unconditional form, which returns control to the controlling system, and a chaining form, which passes control to another program.

THROW

THROW throws an exception.

TRY-CATCH-FINALLY-ENDTRY

The TRY-CATCH-FINALLY-ENDTRY block handles exceptions.

USING

USING and USING-RANGE execute one statement from a list of statements based on the result of an expression.

WHILE

WHILE executes a statement repeatedly based on the result of an expression. If the initial result is false, the WHILE statement never executes.

XCALL

XCALL transfers execution control to an external subroutine.

XRETURN

XRETURN completes the processing of an external subroutine and returns control to the statement that follows the XCALL statement in the calling routine.

Input/output statements

Input/output statements control the exchange of data between the executing program and the input/output devices attached to the computer system. Such devices include printers, terminals, mass storage devices (like disks and floppies), and any other data transmission devices attached to your computer system.

ACCEPT

ACCEPT receives the next sequential character from an input device. (Note that ACCEPT acts differently on OpenVMS; refer to the “[Developing for OpenVMS](#)” section of the *Professional Series Portability Guide* for more information.)

CLOSE

CLOSE completes any pending I/O activity and disassociates a Synergy Language channel from a device or a file.

DELETE

DELETE removes a record from an ISAM file.

DISPLAY

DISPLAY outputs a sequence of one or more characters to a character-oriented device or file.

FIND

FIND locates the position of a record within a file.

FLUSH

FLUSH flushes all buffers associated with a channel.

FORMS

FORMS outputs special form-positioning characters to a character-oriented device or file.

GET

GET reads a fixed-length record from a specified point in a file. This is a binary operation that does not interpret the ASCII characters that are read.

GETS

GETS reads the next sequential fixed-length record from a file. This is a binary operation that does not interpret the ASCII characters that are read.

OPEN

OPEN associates an internal Synergy Language channel with an I/O device or file.

PURGE

PURGE ends I/O activity on a Synergy Language channel. PURGE is equivalent to CLOSE, except that PURGE deletes a file opened in output (**O**) mode.

PUT

PUT outputs a fixed-length record to a specified point in a file. This is a binary output operation, not a record operation, as is the WRITE statement.

PUTS

PUTS outputs a fixed-length record, updating the next sequential record in the file. This is a binary output operation, not a record operation, as is the WRITE statement.

READ

READ obtains a record from a specified point in a file.

READS

READS reads the next sequential record from a file.

STORE

STORE adds a new record to an ISAM file.

UNLOCK

UNLOCK releases any record locks the current program may have on a specified channel.

WRITE

WRITE updates an existing record in a file.

WRITES

WRITES updates the next sequential record.

Interprogram communication statements

Interprogram communication statements pass and obtain data from interactive processes.

LPQUE

LPQUE queues a file for printing by the system.

RECV

RECV receives a message sent by a previously executed SEND statement (either in the current program or some other Synergy Language program).

SEND

SEND transmits a message to the current or a different program. The message is saved until a RECV statement retrieves it.

Preprocessor and compiler directives

Preprocessor directives are instructions for the preprocessor and are implemented before any code is compiled. Compiler directive statements instruct the Synergy compiler and are evaluated only when a program is compiled. Directives are not included in the resulting executable program. See [page 5-1](#) for more information.

.ALIGN

.ALIGN aligns the data location counter to a boundary.

.DEFINE

.DEFINE defines a replacement identifier and optionally assigns replacement text that will be substituted for that identifier. The replacement identifier can be used anywhere in the routine. (See [.DEFINE on page 5-7](#) for more information about text replacement.)

.IDENT

.IDENT defines a routine identifier string to be included in a routine's object file.

.IF-.ELSE-.ENDC

The .IF-.ELSE-.ENDC conditional block specifies that compilation of the statements within the block depends on the truth value of an expression.

.IFDEF-.ELSE-.ENDC

The .IFDEF-.ELSE-.ENDC conditional block specifies that compilation of the statements within the block depends on the declaration of a variable or replacement identifier.

.IFNDEF-.ELSE-.ENDC

The .IFNDEF-.ELSE-.ENDC conditional block specifies that compilation of the statements within the block depends on the nondeclaration of a variable or replacement identifier.

.IFT, .IFF, .IFTF

.IFT, .IFF, and .IFTF are used within conditional blocks to provide greater control over conditional processing. .IFT tests whether the original condition of the current conditional block was true. .IFF tests whether the original condition was false. .IFTF allows processing regardless of whether the original condition was true or false.

.INCLUDE

.INCLUDE causes the compiler to read source code from another source file or from the Repository. Included statements are compiled as if they were part of the current routine at the point where the .INCLUDE statement appears.

.LIST and .NOLIST

.LIST and .NOLIST control the listing of source code.

.NODEBUG

.NODEBUG prevents the debugger from stopping at breakpoints, watchpoints, or anything else that causes the program to pause.

.NOPROTO-.PROTO

.NOPROTO turns off prototype generation for a block of source code in a program, and .PROTO turns it back on.

.PAGE

.PAGE ends the current listing page and begins a new listing page.

.START

.START optionally sets or unsets compilation flags that control the information printed in the program listing. See [.START on page 5-41](#) for a complete list of compilation flags.

.TITLE

.TITLE changes the title in the listing page header.

.UNDEFINE

.UNDEFINE removes the declaration of a replacement identifier that was established by a previous .DEFINE directive.

Identifiers

Identifiers are used as names for namespaces, classes, methods, subroutines, functions, parameters, records, fields, and properties. Identifiers begin with an alpha character, optionally followed by a sequence of alphanumeric characters, underscores (_), and dollar signs (\$) that identify a statement, variable, or routine to the compiler. For example, **Read**, **a10**, **namelast**, and **TAX\$RATE_single** are all valid identifiers. Destructor method identifiers prefix a tilde (~) to the name of the class.

Identifiers are not case sensitive. For example, **NAME_FIRST** is the same as **name_first** and **NAME_first**.

An identifier section is one of the sections in a nested identifier which may include namespace, class, or member name. For example, the following identifier has three identifier sections:

```
mynamespace.myclass.myfield
```

The maximum length of an identifier section is 30 characters on Windows and UNIX and 31 characters on OpenVMS.

All identifiers within a scope, such as classes and structures in a namespace or member identifiers in a class, must be distinct, independent of kind, except where the names are identical and resolved via overloading, such as two routines with the same name but different parameters. A scope is defined to be the smallest of one of the following groupings that has the identifier declared as one of its members, not including inherited members:

- ▶ Namespace
- ▶ Class
- ▶ Structure
- ▶ Record
- ▶ Group
- ▶ CATCH block
- ▶ Local data block

Thus, you cannot have two fields with the same name, or a field and a method with the same name in the same class, but you can have two methods with the same name in the same class if their method signatures are different. You also can't have two classes with the same name or a class and structure with the same name in the same namespace.

Two method signatures are considered to match if their identifier, number of arguments, and argument types match exactly. Two subroutine or function signatures are considered a match if their identifiers match exactly.

Keywords

Keywords (like READ, END, and .DEFINE) have predefined meanings in Synergy Language and identify statements and compiler directives. They must be spelled exactly as shown in this manual.

Replacement identifiers

A replacement identifier defines a text string that is to be substituted for the identifier throughout the program. A replacement identifier is defined by the .DEFINE statement.

Variable names

Variable names are identifiers that refer to data records, groups, and fields. A variable name can have a maximum of 30 characters. Although variable names appear in a program listing exactly as you typed them, the compiler translates the names to uppercase. For example, the compiler recognizes both of the following identifiers as the identifier ABCDE:

```
aBcdE  
abcDe
```

Statement labels

You can label any statement in the procedure division with a symbolic name. Several of the program control statements can reference this label, enabling you to transfer control to a specific statement during program execution. For example, you might use a label to transfer control to an error-trapping routine or when using a CALL statement.

The statement label must be the first nonblank item in the statement, followed by a comma. A statement label can have a maximum of 30 characters on Windows and UNIX and 31 characters on OpenVMS.

Here's an example:

```
      call lb102          ;Unlabeled, references label lb102  
      .  
      .  
      .  
lb102,                   ;Labeled  
      incr flg  
      return
```

A statement label must be unique within the routine.

Routine names

Routine names identify which main routine, external subroutine, or function is being referenced. Like all other identifiers, a routine name can have a maximum of 30 characters on Windows and UNIX and 31 characters on OpenVMS.

Data reference operation names

A data reference operation name identifies a data reference operation. See [“Data reference operations” on page 7-10](#) for more information.

Textual literals

A textual literal is either a sequence of characters enclosed between matching delimiters (single or double quotation marks) or a sequence of numeric characters, which may be preceded by a plus or minus sign. A literal represents a specific value that is defined at compile time and cannot be changed during execution.

Alpha literals

An alpha literal is a sequence of characters enclosed in matching single or double quotation marks. An alpha literal can be up to 255 characters long.

Here are some examples of alpha literals:

- ▶ `'This is an ALPHA literal'`
- ▶ `"as is this one ....."`

If an alpha literal contains either a single or double quotation mark and you're using the same character as the delimiter, you must use two successive characters to represent a single embedded mark. The examples below illustrate embedded delimiters: two successive quotation marks are required to specify one quotation mark within the literal.

- ▶ `"An" "is embedded"`
- ▶ `"quotation mark at end""`

If the embedded character is different from the ones delimiting the literal, you don't need to double the embedded character (as shown in the examples below).

- ▶ `"An ' is embedded"`
- ▶ `' "Quotation marks are present"...'`

Sometimes it's more convenient to split an alpha literal into smaller physical parts. For example, you might do this if you're using the literal to initialize an entire record and you want to append comments to each field. The Synergy compiler concatenates several alpha literals if they are separated only by blanks and tabs. (If sections of the alpha literal are on continuation lines, each line must begin with the continuation character [&].)

For example:

```
"This is "      "an alpha "      "literal"
```

and

```
"This is "  
&"an alpha "  
&"literal"
```

are functionally the single alpha literal

```
"This is an alpha literal"
```

The blanks and tabs between the separate alpha literals are not part of the concatenated literal. (See [“Continuation lines” on page 4-6](#) for a warning about splitting alpha literals across two or more physical lines.)

Alpha literals are case sensitive.

Decimal literals

A decimal literal is any consecutive sequence of numeric characters, which may be preceded by a plus (+) or minus (–) sign. A decimal literal can consist of a maximum of 28 digits. The compiler strips nonsignificant leading zeros to a minimum of one digit. For example, the literal 00003 is stored as 3. The literal 000 is stored as 0.

Here are some examples of decimal literals:

- ▶ 273949830000000000
- ▶ 1
- ▶ -391

Implied-decimal literals

An implied-decimal literal is a sequence of not more than 28 numeric characters, which may be preceded by a plus or minus sign. A maximum of 28 digits can appear before the decimal point, and a maximum of 28 can occur after it, but the two combined cannot exceed 28 total digits. The decimal point cannot be the first or the last character in an implied-decimal literal; at least one leading and trailing digit must appear before and after the decimal point.

Here are some examples:

- ▶ 1234.567
- ▶ 123456789012345678.0123456789
- ▶ +18.10
- ▶ 0.928456989
- ▶ -518.0

The compiler strips nonsignificant leading and/or trailing zeros prior to processing.

Integer literals

You cannot explicitly write an actual integer literal in a source line in the same way that you can specify alpha, decimal, and implied-decimal literals. However, if you include a decimal literal in an arithmetic expression with an integer variable, the compiler builds an integer literal into the code.



You can also use %INTEGER with a numeric literal argument to create an integer literal.

Error literals

An error literal has the format \$ERR_*mnemonic*, where *mnemonic* is an error identification mnemonic. One error literal is defined for every trappable runtime error. (For example, \$ERR_DIVIDE specifies the “Divide by zero” error.)

The entire error literal is treated as a decimal literal, and you can use error literals with ONERROR statements, in I/O error lists, and anywhere else in your program that you can use decimal literals. Here’s an example:

```
onerror($ERR_IOFAIL, $ERR_DIGIT) proc_err1,  
      & ($ERR_EOF) proc_err2  
read(CHN, data, rec_id) [$ERR_IOFAIL=proc_err1]
```

The “[Error Messages](#)” chapter of *Synergy Language Tools* contains a complete list of Synergy Language error mnemonics, numbers, and messages.

Variables

A variable is an identifier that refers to a data location in your program. Before using a variable in statements in the procedure division, you must first define it in the data division. The data division statement that defines the name and data type of the variable also determines the characteristics of the variable’s data area: size, possible initial values, and position relative to the other variables.

Shared variables (common records, global records, and global literals) can be declared by any routine in a program and can be accessed by any routine in a program. A shared variable name must be unique. Variables that are not shared are local to the routine in which they are declared.

Variable path specifications

A path uniquely references a named record or a field within a data structure (as specified by the RECORD or GROUP statement). Variable path specifications have the following format:

[struct_name.][struct_name....]field_name

Paths have the following characteristics:

- The path can contain any number of data structure names as long as it creates a unique variable reference. Use a period (without any spaces) as the delimiter between data structure names or between a structure name and the field name. Let's use the following group as an example:

```
record a
  group b
    fld      ,a2
  group c
    nam      ,a20
    city     ,a10
  endgroup
endgroup
```

You can reference the field **city** with the following paths:

```
city
a.city
b.city
c.city
a.b.city
a.c.city
b.c.city
a.b.c.city
```

Note that you don't have to use the whole path name to reference a field.

- You can use the same field name more than once in GROUP and named RECORD statements. However, when you reference a nonunique field, make sure the path specification is unique. For example, for the following data area:

```
record inhouse
  acct      ,d5
  name      ,a30
  address   ,a40
  zip       ,d10

record client
  acct      , [100]d5
  group customer, [100]a
    name    ,a30
    group bldg ,a
      group address,a
        street ,a40
        zip    ,d10
      endgroup
    endgroup
  group contact,a
    name     ,a30
```

Welcome to Synergy Language

Basic Language Elements

```
        group address,a
            street ,a40
            zip    ,d10
        endgroup
    endgroup
endgroup
```

the following paths are valid:

```
contact.name
contact.address.street
customer[5].bldg.address.street
client.accnt
inhouse.accnt
inhouse.name
```

while the following paths are invalid because they are not unique:

```
name
customer[1].name
address.street
accnt
```

If you specify one of these paths, the compiler generates a “Path specification is ambiguous” error (AMBIGUOUS).

- If you do not define structures with unique paths to all fields, you may end up with fields that you cannot access directly. For example, the fields in the first address group in the following group specification can never be accessed directly:

```
group customer    ,a
    name          ,a30
    group address,a
        street    ,a40
        zip       ,d10
    endgroup
    group contact,a
        name      ,a30
        group address,a
            street ,a40
            zip    ,d10
        endgroup
    endgroup
endgroup
```

Because partial path names can access fields, the second address group's **street** field can be referenced in any of the following ways:

```
customer.contact.address.street  
customer.contact.street  
contact.street
```

The path specification

```
customer.address.street
```

generates a “Path specification is ambiguous” compiler error, because it could reference either the first **address** group or the second. The only way to reference the data in the first **address** group is by ranging. See [“Ranged references” on page 1-32](#) for information.

- ▶ If no other delimiters are present, the period character in a valid path specification takes precedence over any logical or Boolean operator. For example, if your group looks like this:

```
group var1      , a  
  group and    , d4  
    var2      , d4  
  endgroup  
endgroup
```

and you reference it with the following path:

```
var1.and.var2
```

the Synergy compiler interprets this reference as a path, not a Boolean expression. (To use it as a Boolean expression, add a space before and after the **.AND.** operator.)

- ▶ The data type of the variable reference is the data type of the rightmost element in the path.

Object paths

You can refer to the current instance of a class within the class using the **this** keyword. For example,

```
this.mymethod()
```

Real arrays

When referencing a real array, you must specify each declared dimension. For example, if you declare a variable as follows:

```
record  
  brk      , [3,4]d1
```

the references **brk[1,2]** and **brk[2,2]** are valid because each dimension is referenced.

In contrast, **brk[1]** is *not* valid, and the compiler generates an “Incorrect number of dimensions” error (INVNUMDIM), because it specifies only one of the two dimensions.

The reference **brk[]**, however, *is* valid and refers to the entire scope, or contents, of the dimensioned array as a single element. The maximum size of a scope reference is 65,535. If the array is larger than 65,535 bytes, the scope size is modulo 65,535.

If your dimension specification is larger than the declared size of its corresponding array dimension, you can reference outside the defined area up to the end of the data area, except for class data fields and when **-qstrict** or **-qcheck** is specified. We recommend using **-qcheck** in all development environments and **-qstrict** for all production builds.

Here's an example that references a real array:

```
record demo
  alpha          , [3,2]d2,      12 ,34,
  &              56 ,78,
  &              98 ,76
  beta, [2,4]a3,      "JOE" , "JIM" , "TED" , "SAM" ,
  &                  "LOU" , "NED" , "BOB" , "DAN"
```

Data is referenced as follows:

Variable reference	Data obtained
alpha[1,2]	34
alpha[3,1]	98
alpha[3]	INVNUMDIM error
alpha[3,3]	JO
alpha[4,1]	JO
alpha[]	123456789876
beta[]	JOEJIMTEDSAMLOUNEDBOBDAN
beta[1,3]	TED
beta[2,4]	DAN

Variable references

You can specify variables in a simple, subscripted, or ranged form. Each form is described below.

Simple references

A simple reference consists of the name of a data area, without any subscripting or range specifications. If it's not defined as an array, the reference refers to the entire data area. Otherwise, the simple reference refers to the first element of the array. Here are some examples of simple references:

- ▶ amount
- ▶ a_result
- ▶ Y73
- ▶ days[m,1]
- ▶ cust[i,j].name
- ▶ var[]
- ▶ HELP

Subscripted references

A subscripted reference refers to an individual element within a list of like elements. The variable name is followed by the subscript value, which is enclosed in parentheses. The subscript value can be any numeric expression greater than zero and defines which element in the list is being accessed. For example, a subscript value of 1 represents the first element in the list.

Any variable reference can be subscripted, provided the data area is accessible to the routine. If the data area is inaccessible, or if the subscript value is less than one, an "Invalid subscript specified" error (\$ERR_SUBSCR) is generated. Note that you cannot subscript beyond the defined size of class data fields and records or when the **-qcheck** compiler option is specified.

To understand how subscripted variables work, consider the following example.

Assume the following data division statements:

```
record demo
  alpha      ,3d4,      3, 9876, 4321
  beta       ,a6,      "ABCDEF"
  gamma      ,d3,      545
  delta      ,4a2,      "LM", "NP", "RS" , "TV"
  epsilon    ,3d3.2 ,   6.75, 1.23, 8.00
  group theta ,[3]a
    one       ,d3,      123
    two       ,a3,      "abc"
endgroup
```

Data is allocated as follows (shaded boxes are not part of the data space):

Variable	Data area (characters)					
alpha(1)	0	0	0	3		
alpha(2)	9	8	7	6		
alpha(3)	4	3	2	1		
beta	A	B	C	D	E	F
gamma	5	4	5			
delta(1)	L	M				
delta(2)	N	P				
delta(3)	R	S				
delta(4)	T	V				
epsilon(1)	6	7	5			
epsilon(2)	1	2	3			
epsilon(3)	8	0	0			
theta[1].one	1	2	3			
theta[1].two	a	b	c			
theta[2].one	1	2	3			
theta[2].two	a	b	c			
theta[3].one	1	2	3			
theta[3].two	a	b	c			

Here are some sample variable references:

Variable reference	Data obtained	Explanation
demo	000398764321ABCDEF 545LMNPRSTV6751238 00123abc123abc123abc	Refers to the entire data area.
alpha	0003	Refers to the first element of the array.
alpha(1)	0003	Refers to the first element of the array.
alpha(3)	4321	Refers to the third element of the array.
alpha(4)	ABCD	Refers to the fourth group of four characters after the beginning of alpha's data area (because a single alpha data item is four characters).
alpha(5)	EF54	Refers to the fifth group of four characters after the beginning of alpha's data area.
beta(2)	545LMN	Refers to the second group of six characters after the beginning beta's data area (because a single beta data item is six characters).
gamma (alpha(1))	PRS	Refers to the third group of three characters after the beginning of gamma's data area (because the variable alpha(1) has a value of 3 and a single gamma data item is three characters).
delta(4)	TV	Refers to the fourth element of the array.
delta(-3)	\$ERR_SUBSCR	The subscript value is less than one.
epsilon(3)	8.00	Refers to the third element of the array. (The decimal point is shown here for clarity; it is not stored in the data area.)
epsilon(4)	123	Refers to the fourth group of three characters after the beginning of epsilon's data area.
theta[1].two(2)	123	Refers to the second set of three characters after the first array element in the group.
theta[3].two(2)	\$ERR_SUBSCR	Refers to data outside of the data area accessible to the program.

Alpha(4) and **alpha(5)** are both valid references because they don't extend beyond the total data space allocated by the data division statements. **Theta[3].two(2)**, on the other hand, attempts to refer past the end of the data area, which is not possible.

Notice that **beta** and **gamma** can be subscripted even though they are simple variable references.

Ranged references

A ranged reference is a variable reference followed by a range specification, either in absolute or relative form, that is enclosed in parentheses. In either case, the specification indicates starting and ending character positions relative to the beginning of the variable's data space. The ranged variable's value is the sequence of characters between the starting and ending positions, inclusive. If the range in a variable reference goes beyond the first variable, Synergy Language continues to the next variable to obtain data.

You can range real arrays. Here's an example:

```
brk[1,2] (1,23)
```

You cannot range subscripted arrays. The following generates an error:

```
array(4) (1,23)
```

If the variable being ranged is implied-decimal, it is interpreted as decimal. Note that you cannot range beyond the defined size of class data fields and records or when the **-qcheck** compiler option is specified.

Absolute ranging

The absolute form of a range specification indicates the starting and ending positions of a character sequence using two numeric expressions separated by a comma.

Absolute ranging has the following format:

```
variable(start_pos,end_pos)
```

Here's an example:

```
record abs
  alpha      ,d8,      10203405
  beta       ,a13,     "ABCDEFGHJKLM"
  gamma      ,[3,2]a3,  "ABC", "DEF",
&            "GHI", "JKL",
&            "MNO", "PQR"

  group theta ,[3]a
    one       ,d3,      123
    two       ,a3,      "abc"
  endgroup
```


Variable reference	Data obtained
alpha(2,4)	020
alpha(8,13)	5ABCDE
beta(3,6)	CDEF
beta(alpha(2,3),alpha(7,8))	BCDE
beta(alpha(2,3)+2,5)	DE
beta(13,14)	MA
gamma[1,2](2,3)	EF
gamma[2,2](3,6)	LMNO
alpha(6,3)	\$ERR_SUBSCR
theta[1].two(2,8)	bc123ab

Note that the ability to access characters that extend past (or in front of) the specified field has been deprecated. For backward compatibility, the Synergy Runtime will continue to support this type of access. However, we recommend that you compile and run with **-qcheck** and make the appropriate changes to avoid future memory access violations.

Relative ranging

The relative form of a range specification indicates the starting or ending character position and length of a character sequence range using two numeric expressions separated by a colon. The value that is referenced depends on the sign of the length. For positive lengths, the specified position is the starting point, and the value is the character sequence of the specified length that begins at that position. For negative lengths, the position is the ending point, and the value is the character sequence that begins the specified number of characters before that position.

Relative ranging has the following format:

variable(position:length)

Here's an example of a data division:

```
record abs
  alpha      ,d8,          10203405
  beta       ,a13,         "ABCDEFGHJKLMN"
  gamma      , [3,2]a3,    "ABC ", "DEF ",
&            "GHI ", "JKL ",
&            "MNO ", "PQR "
```

Welcome to Synergy Language

Basic Language Elements

```
group theta      , [3]a
  one      , d3 ,    123
  two      , a3 ,    "abc"
endgroup
```

Variable reference	Data obtained
alpha(3:1)	2
alpha(3:2)	20
alpha(6:-2)	34
beta(13:2)	MA
gamma[1,2](1:2)	DE
gamma[1,1](1:12)	ABCDEFGHIJKL
gamma[3,2](1:12)	PQR123abc123
beta(2:-15)	\$ERR_SUBSCR
theta[2].one(3:10)	3abc123abc

Note that the ability to access characters that extend past (or in front of) the specified field has been deprecated. For backward compatibility, the Synergy Runtime will continue to support this type of access. However, we recommend that you compile and run with **-qcheck** and make the appropriate changes to avoid future memory access violations.

Other System Features

Data encryption

With the number of security regulations on the rise, data encrypting has become a critical requirement for most application developers. As of version 9.3, Synergy Language enables you to encrypt and decrypt sensitive data using industry standard cipher techniques. This is accomplished on an on-demand basis using a set of Synergy subroutines that encrypt and decrypt a specified data region. You determine which data is encrypted as well as how it is encrypted.

The data encryption feature interfaces with a third-party library, OpenSSL, to provide SSL support.

Setting up your system for encryption

Follow these steps to set up your system to use encryption. For details on which version of OpenSSL is required for your operating system, see “[OpenSSL Requirements](#)” in the *Installation Configuration Guide*.

1. Install OpenSSL. See Synergex KnowledgeBase article [100001979](#) for the latest information on where to obtain OpenSSL for the various operating systems and details on installation.
 - ▶ For Windows, go to <http://www.openssl.org/related/binaries.html> and follow the link to download the version for your platform.
 - ▶ For UNIX, obtain the OpenSSL security kit specific to your operating system and version from your operating system vendor, if one is available. For some platforms, you may have to download and build the shared libraries. See Synergex KnowledgeBase article [100001970](#) for details.
 - ▶ For OpenVMS, use the operating system-supplied libraries, HP SSL, available from http://h71000.www7.hp.com/openvms/products/ssl/ssl_download.html.
2. Ensure that the OpenSSL shared libraries are in the correct location or have been added to the correct path.
 - ▶ On Windows, if OpenSSL is installed in the default location (the Windows system directory), you don't need to set anything. If you install to another location, set OPENSSL_LIB to the location of the libraries. If OPENSSL_LIB is not set, PATH is used to find the OpenSSL libraries.
 - ▶ On UNIX, whether you install the vendor-supplied OpenSSL libraries or build the shared libraries yourself, you will need to set the library path (LIBPATH on AIX, SHLIB_PATH on HP-UX, and LD_LIBRARY_PATH on all other systems).



If you build the OpenSSL libraries yourself, you will need to build the shared versions of the libraries.

- ▶ On OpenVMS, during installation, SYNSSLIB is set in `SYSS$MANAGER:SYNERGY_STARTUP.COM` to the full path and filename of **SYNSSLIB.EXE**, the SSL runtime support file.

Encrypting and decrypting data

The `DATA_ENCRYPT` and `DATA_DECRYPT` routines encrypt and decrypt data using the data and password buffers that you pass to them, and the `DATA_SALTIV` routine generates a salt and/or initialization vector (IV) that you can use with the other two routines to further enhance security. Supported encryption routines include 3DES (Triple DES) running in triple CBC mode and AES running in CBC mode. (For syntax information, see [DATA_ENCRYPT on page 9-71](#), [DATA_DECRYPT on page 9-69](#), and [DATA_SALTIV on page 9-73](#).)

Ciphers process data in blocks. If the length of the data isn't a multiple of the cipher block size, `DATA_ENCRYPT` and `DATA_DECRYPT` use PKCS padding to pad it. Bytes of padding (between 1 byte and the number of bytes in a cipher block) are always added to the data when it is encrypted and then removed when it is decrypted. When using PKCS padding, the unencrypted data can be any size, but the encrypted data is always a multiple of cipher block size; PKCS padding does not allow in-place encryption even when the data is a multiple of the cipher block size.

To encrypt and decrypt data without changing its size (or for in-place encryption), you can suppress padding, but the data *must* be a multiple of cipher block size. To suppress padding, pass a 0 value to the `DATA_ENCRYPT` and `DATA_DECRYPT` routine's *pad* argument. If you suppress padding to encrypt a piece of data, you must also suppress padding when you decrypt the same piece of data, or the decryption will fail with an "Argument specified with wrong size" error (`$ERR_ARGSIZ`).



In-place encryption is not allowed unless you explicitly suppress padding.

Cipher block sizes are as follows:

3DES	8 bytes
AES	16 bytes

You can determine the size of the encrypted data by passing the *len* argument to `DATA_ENCRYPT` with no destination argument, which applies the following formulas. (*Padding* is either 1 or 0.)

3DES	$8 * (\%TRUNC(data\_size / 8) + padding)$
AES	$16 * (\%TRUNC(data\_size / 16) + padding)$



Only non-key data should be encrypted in a record destined for an ISAM file, because encrypted key data will not index properly. Also, access to encrypted data in a Synergy file is not supported by `xrODBC`. Make sure you mark encrypted fields as "Excluded by ReportWriter" in the repository.

To implement Synergy data encryption,

1. Decide what type of encryption you want to use and which fields should be encrypted. For example, you might choose to use AES-128 encryption to encrypt a field in your customer master record. You can allow for padding by inserting a “pad” field directly after the field and then declaring an overlay field to encompass both the original field and the pad field. Make sure when encrypting that the source doesn’t include the pad field. If the field is a multiple of the cipher block size, you may also suppress padding.
2. If you want to maximize the effectiveness of the cipher by including a random salt and/or IV, call the `DATA_SALTIV` routine and save the resulting value(s).
3. Before writing data to your master file using the `STORE` and `WRITE` statements, call the `DATA_ENCRYPT` routine with a user password and the salt and/or IV you generated in [step 2](#), if applicable.
4. After reading data from your master file using the `READ` and `READS` statements, call the `DATA_DECRYPT` routine with the same user password, salt, and/or IV that you used to encrypt the data.
5. Convert your master file (if one existed before) to the new record structure. (Note that you must construct a program or routine to perform this conversion; no utility exists to convert the file automatically.)

Prototyping

As of version 9.1, Synergy Language provides strong prototype validation of system-supplied routines at compile time. This means that the compiler compares each system-supplied routine used in your program to its prototype to ensure it has the correct number and type of arguments, correct return type, and so on. (If you need to relax some of the strong prototype validation, use the **-qrelaxed** option(s) when you compile.)

Object-oriented Synergy code that you write is always strongly prototyped. If desired, you can also choose to implement strong prototyping for your non-object-oriented Synergy code using the Synergy Prototype utility. Doing so will ensure that calls match their routine definitions, which enables you to detect more problems at compile time rather than waiting until runtime.

For nonprototyped routines in the same compilation unit, the compiler will automatically perform weak prototype checking. This means the compiler will issue warnings (rather than errors) based on the local definitions that are available at compile time. The amount of checking that takes place depends on the information available to the compiler. (If you need to relax some of the weak prototype validation, use the **-qrelaxed:local** option when you compile.)

See “[The Synergy Prototype Utility](#)” in the “General Utilities” chapter of Synergy Language Tools for more information on prototyping.

Profiling

An often-overlooked developer tool is the Synergy Language Profiler. The profiler provides information that enables you to analyze the resources required by your application. It can help you significantly improve application performance by identifying the areas that are not as efficient as they could be.

Two profiling techniques are available:

- ▶ Routine profiling, which shows which routines are being accessed and what resources are associated with each routine, gives you a picture of a program's execution process.
- ▶ Line profiling, which shows you which lines in a routine are being executed the most, enables you to identify potential bottlenecks within a routine.

You can also exclude specific routines from the profiler trace by specifying an exclusion list. By default, the profiler uses an exclusion list for the UI Toolkit input routines (WND:tkexclude.txt).

Profiling is enabled either by setting DBLOPT #40, #41, or #52 and using the profiling compiler option (**-u** on Windows and UNIX or **/profile** on OpenVMS) or by setting DBLOPT #42. (Refer to “[The Synergy Language Profiler](#)” in the “General Utilities” chapter of *Synergy Language Tools* for the distinctions between these system options and under what circumstances you would want to use each one.) When you execute your program, profiling information is stored in a file called **profile.dat**. You can then run the profile or propline utility (depending on whether you are using routine profiling or line profiling) to produce a report of the results.

Messaging

You can send messages between routines using the SEND and RECV statements.

Synergy Language provides two types of message facilities to process these messages: the local message facility (which is the default) and the Synergy message manager.

Using these message facilities, your program can communicate with other programs that are currently running or that will be running soon. For example, a program can send messages that affect how processing will be performed by programs to which it subsequently chains.

Both message facilities maintain local, group, and global queues. If you only use the local message facility, messages are only available to the current job stream. Routines are in the same job stream if they are called as subroutines or functions or if they are chained to. On OpenVMS, routines that are chained to are only considered to be in the same job stream if the program is bound. Once the runtime terminates, message queues are reinitialized.

If the local message facility is sufficient for your needs, we recommend using it rather than the Synergy message manager, which can incur additional processing overhead.

Differences between the message facilities

Local message facility

The local message facility is maintained by the runtime. Once the runtime terminates, all messages in the local message facility are lost.

Synergy message manager

Unlike the local message facility, the Synergy message manager (which is the Synergy Language daemon on UNIX) enables you to send messages between processes. The Synergy message manager is a detached program that retains and maintains your messages. Even after the runtime terminates, the messages still exist, and you can run another program and receive those messages.

UNIX

To use the Synergy message manager, you must set system option #7 with the DBLOPT environment variable. (See the “[System Options](#)” chapter in *Environment Variables and System Options* for more information.) If you don’t set option #7, Synergy Language uses the local message facility instead of the Synergy message manager for SEND and RECV statements.

See [SEND on page 4-165](#) and [RECV on page 4-160](#) for more information about sending and receiving messages.

VMS

The Synergy message manager is used by default. It writes errors to the log file **DBLDIR:dblmsgctl.log**.

To disable the messaging facilities, set system option #47.

The message manager can buffer up to 32000 messages.

WIN

Synergy Language uses only local queues. There is no Synergy Message Manager on Windows.

Synergy Language messages and message queues

Each Synergy Language message is identified by a combination of the terminal number to which it is directed and a message ID of one to six characters (or one to thirty-nine characters on OpenVMS). If a SEND statement doesn’t specify a terminal number, the resulting message’s terminal identification includes the number of the terminal that initiated the program.

Local messages

UNIX

Each terminal running a Synergy Language program has its own local message queue into which messages are inserted. Messages are retrieved from this queue in “first in, first out” order, according to message ID. Each message in a terminal’s local queue is called a local message.

By default, if you don’t specify a terminal number in a SEND statement, the message is inserted into the local message queue unless system option #1 or #13 is set. These options are explained in more detail below.

Group and global messages

Sometimes you’ll want to send a message that has an ID but that is not directed to any particular terminal. You can use either group messages or global messages for this purpose. If you are sending to or from a detached process, you need to use a group or global queue and the Synergy message manager.

A global message can be received by any program that explicitly specifies the correct message ID in its RECV statement. Synergy Language provides one global message queue for your system. Any message directed to terminal number 255 is inserted into the global queue and can be retrieved by referencing the message’s ID.

A group message can be received by any program specifying the correct message ID and running under the same group number as the sending program. Synergy Language provides one group message queue for each group (which consists of a unique project and programmer number combination for each account). Any message directed to terminal number 254 is inserted into the group queue.

If you specify system option #13 with the DBLOPT environment variable, a SEND statement that does not specify a terminal number sends the message to terminal number 254 (the group message queue) by default. If you specify system option #1, the message goes to terminal number 255 (the global message queue) by default. Option #13 supersedes option #1. Therefore, if you specify both options, option #1 is ignored. See [#1](#) and [#13](#) in the “System Options” chapter of *Environment Variables and System Options* for more information.

If you need to send a message to a program that will be running in the future, send it to a group or global message queue using the Synergy message manager. Programs whose terminal number is negative (for example, detached processes or batch jobs) use a global message queue by default.

VMS

Group messages are the same as global.

Sending a message to terminal -1 is the same as sending to the global queue.

Receiving messages

Unless the DFLAG subroutine's runtime option flag 1 is set, the RECV statement begins searching for a requested message ID in the local queue for that process. If no message ID is specified, RECV assumes that the message ID is [SELF]. If RECV can't find a message that matches the specified ID, it searches first in the group message queue and finally in the global message queue for the first message with the specified ID.

If RECV can't find the message in any of these queues, it returns a "message not found" status. You can also use the DFLAG subroutine to indicate that your program should wait for the requested message to become available if no message exists. If you set DFLAG's runtime option flag 2, your program waits for the message and does not signal the "message not found" status. (See [DFLAG on page 9-98](#) for more information.)

By default, the RECV statement searches all three message queues in the order described above. However, you can use the RCFLG subroutine to disable the searching of any queue for a particular program. RCFLG enables you to change the searching algorithm of the Synergy message manager by setting one or more of its receive option flags. See [RCFLG on page 9-255](#) for more information.

Special message IDs

Both message facilities recognize two special message IDs: [SELF] and [INIT].

A [SELF] ID in a SEND or RECV statement is interpreted as one of the following:

- ▶ The name specified by the main routine's MAIN statement.
- ▶ If MAIN wasn't used, the first six characters of the main routine's source filename.

WIN, UNIX

An [INIT] ID in a SEND statement causes the message facilities to discard all messages currently contained in the local message queue. When using the Synergy message manager, you can use [INIT] to restore the environment to a known state after a Synergy Language program terminates abnormally.

Synergy Language Files

File types

Synergy Language uses four types of files: ISAM, relative, sequential, and stream.

VMS

For more information about all file types, refer to your *Guide to OpenVMS File Applications* (which is part of your OpenVMS systems documentation set).

ISAM files

An ISAM file consists of a series of data records referenced by key. An ISAM file is created by the ISAMC subroutine or **bldism** utility on Windows and UNIX, or by CREATE/FDL on OpenVMS. For more information, see the “[Synergy DBMS](#)” chapter of *Synergy Language Tools*.

Relative files

A relative file consists of a series of fixed-length data records referenced by their relative position within the file. A relative file is created when you specify the relative (**R**) submode when opening the file for output or update mode. Data records can be accessed by record number or sequentially.

Synergy Language relative files provide easy, consistent migration of programs using fixed-length records from other Synergy/DE environments; they make it possible to bring an application to OpenVMS from another environment without changing the file characteristics or the manner in which the file is processed.

Sequential files

A sequential file consists of a series of data records referenced in the order in which they exist in the file. The size of records can vary within the file. A sequential file is created when you specify the sequential (**S**) submode when opening the file for output or update mode. Data records are sequentially written to and read from a sequential file; they cannot be accessed by record number.

Stream files

A stream file consists of a series of data records whose contents are defined by their record terminators. On Windows and UNIX, there is no physical difference between stream files and sequential files except that stream files can be accessed randomly based on the size of the data area used in the I/O statements. On Windows and UNIX, stream files are created by default if you open a file in output mode without specifying a submode. On OpenVMS, you can change the default to stream (instead of sequential) by compiling with /STREAM or by specifying /STREAM in the OPEN statement’s OPTION string and not specifying a submode on the OPEN statement with either method.

File specifications

Any place you can specify a filename in Synergy Language—on command lines, in OPEN statements, in .INCLUDE statements, and so forth—you can specify a directory path or a search list logical for that file. A filename can contain spaces on Windows.



The “at” sign (@) is invalid in a file specification unless you are using it to denote a remote (xServer) filename.

Search list logicals

A search list logical is an identifier defined at the operating system level as a list of one or more directories. When a search list logical is specified, Synergy Language searches each of the directories listed for the specified file. If it’s specified when a file is created, Synergy Language places the file in the first directory specified in the logical.

An example of defining a search list logical under the UNIX Bourne shell is as follows:

```
DAT=/usr/fred,/usr/fred/dbl
```

An example of defining a search list on OpenVMS is as follows:

```
DEFINE DAT [usr.fred],[usr.fred.dbl]
```

If you use the DAT logical in a path specification, as shown in the example below, Synergy Language looks for the specified file first in /usr/fred and then, if it can’t find it, in /usr/fred/dbl:

```
open(1, i, "DAT:afire")
```

Here’s another example:

```
xcall renam("DAT:newfile", oldfile)
```

In this example, the renamed file (**newfile**) is placed in /usr/fred, because it’s the first directory specified in the search list.

See the “[Environment Variables](#)” chapter in *Environment Variables and System Options* for more information about setting search list logicals on your operating system.

Default extensions

If you don’t specify an extension in your file specification, Synergy Language assumes one. The default extension depends on the command or statement with which you are using your file specification. For example, on a compiler command line, the Synergy compiler assumes .dbl is the extension of your source file. Refer to the section of this reference manual that describes the command or statement you are using to find out what filename extension is assigned.

2

Defining Data

Data Types 2-2

Describes the data types available in Synergy Language and where they can be used. Also discusses the format for field and parameter definitions.

Using Data 2-39

Discusses where fields can be defined and the organization and layout of data.

Lifetime of Data 2-47

Discusses the scope and characteristics of different kinds of data and explains how memory is managed in a Synergy program.

Data Types

The table below summarizes the data types available for use in Synergy Language. The “Valid in” columns indicates whether types can be used with Synergy standard and Synergy .NET. See the descriptions beginning on [page 2-7](#) for more details on each data type. See the table on [page 2-18](#) for information on where (fields, parameters, return values) these types can be used.



The table below includes the .NET data types for which we have created a shorthand type (e.g., “uint” for System.UInt32), those for which there is also support in Synergy standard, and those for which we perform some special handling. However, you can use any data type in .NET Framework’s System namespace when you compile with the Synergy .NET compiler. (This namespace is imported automatically when you use the Synergy .NET compiler.) See also “[Understanding .NET data types](#)” on [page 2-5](#). For complete details on .NET data types, see the [.NET Framework documentation](#) in Microsoft Developer Network (MSDN).

Data type	Valid in		Description	Default value
	Standard	.NET		
a	X	X	Alpha	NA
a*	X	X	Alpha with initial value	The initial value used to size the a
asize	X	X	Fixed-length 8-bit character sequence	Spaces of specified size
boolean	X	X	System.Boolean in .NET; maps to i1 in standard	false
byte	X	X	Shorthand for System.Byte in .NET (an unsigned 8-bit integer); maps to i1 in standard (a signed 8-bit integer)	0
D_ADDR	X	X	Shorthand for System.IntPtr in .NET; maps to i4 (32-bit) or i8 (64-bit) in standard	0
d	X	X	Decimal	NA
d*	X	X	Decimal with initial value	The initial value used to size the d

Data type	Valid in		Description	Default value
	Standard	.NET		
<i>dsize</i>	X	X	Fixed-length decimal	Array of character “0” (zero) of specified size
<i>d.</i>	X	X	Implied-decimal	NA
<i>dsize.precision</i>	X	X	Fixed-length implied-decimal	Array of character “0” (zero) of specified size
<i>decimal</i>	X	X	System.Decimal in .NET; maps to d28.10 in standard	0.0 in .NET; array of character “0” (zero) of size 28 in standard
<i>double</i>	X ^a	X	Shorthand for System.Double in .NET; maps to d28.10 in standard	0.0
<i>float</i>	X ^a	X	Shorthand for System.Single in .NET; maps to d28.10 in standard	0.0
<i>i</i>	X	X	Integer	NA
<i>i*</i>	X	X	Integer with initial value	The initial value used to size the <i>i</i>
<i>i1</i>	X	X	Signed 8-bit integer	0
<i>i2</i>	X	X	Signed 16-bit integer	0
<i>i4</i>	X	X	Signed 32-bit integer	0
<i>i8</i>	X	X	Signed 64-bit integer	0
<i>int</i>	X	X	Shorthand for System.Int32 in .NET; maps to i4 in standard	0
<i>IntPtr</i>		X	System.IntPtr	0
<i>long</i>	X	X	Shorthand for System.Int64 in .NET; maps to i8 in standard	0
<i>n</i>	X	X	Whole number numeric	NA
<i>n.</i>	X	X	Implied and whole number numeric	NA
<i>p</i>	X		Packed	NA

Defining Data

Data Types

Data type	Valid in		Description	Default value
	Standard	.NET		
<i>p.size</i>	X		Packed	Array of character "0" (zero) of specified size
<i>p.</i>	X		Implied-packed	NA
<i>p.size.precision</i>	X		Implied-packed	Array of character "0" (zero) of specified size and precision
<i>sbyte</i>	X	X	Shorthand for System.Sbyte in .NET; maps to i1 in standard (a signed 8-bit integer)	0
<i>short</i>	X	X	Shorthand for System.Int16 in .NET; maps to i2 in standard	0
<i>string</i>	X	X	System.String in .NET; Synergy Language System.String in standard	^NULL ^b
<i>ushort</i>		X	Shorthand for System.UInt16	0
<i>uint</i>		X	Shorthand for System.UInt32	0
<i>ulong</i>		X	Shorthand for System.UInt64	0
<i>^VAL</i>	X	X	System.IntPrt in .NET; native int in standard	NA
<i>VOID</i>	X	X	Method does not return a value	NA
<i>@*</i>	X	X	Shorthand for System.Object in .NET; shorthand for Synergy Language System.Object in standard	^NULL ^b
<i>@class</i>	X	X	A specific class type	^NULL ^b
<i>@delegate</i>		X	A specific delegate	^NULL ^b
<i>@interface</i>		X	A specific interface	^NULL ^b
<i>@type</i>	X	X	A specific boxed type	Depends on the data type
<i>enumeration</i>	X	X	A specific enumeration	0

Data type	Valid in		Description	Default value
	Standard	.NET		
<i>structure</i>	X	X	A specific structure type	Individual fields are initialized based on their type
<code>[#, #]type</code>	X	X	Dynamic array of the specified type	Depends on the value type
<code>[#, #]@type</code> <code>[#, #]@class</code> <code>[#, #]@*</code>	X	X	Dynamic array, where <i>array</i> can have up to 9 strides (for example, <code>[#, #, #, #]</code>)	^b ^NULL

a. Not recommended for use in Synergy standard.

b. ^NULL represents an uninstantiated object handle and can be used wherever and object can be used.

Understanding .NET data types

With Synergy .NET, there are three basic types of data types: value types, descriptor types, and class types. All inherit from the .NET Framework class `System.Object`. This means, for example, that you can use `System.Object` methods, such as `ToString`, with all Synergy .NET data types.

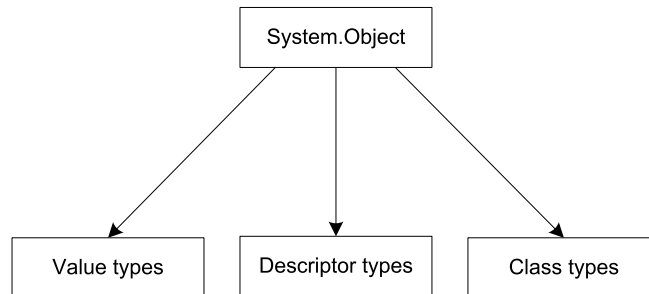


Figure 2-1. Data types in Synergy .NET.

Value types are structures in the .NET Framework `System` namespace (`System.Int32`, for example). They directly access data and are, therefore, the most efficient data types. For example, in Synergy .NET, `int` maps to the `System.Int` structure.

Descriptor types use a class that contains information on the data type and memory location of actual data. Consequently, descriptor types aren't as efficient as value types, but they do have the following advantages:

- ▶ Only parameters defined as descriptor types can be optional.
- ▶ Only descriptor types can be used as variable arguments (i.e., the extra arguments you get with `VARARGS`; this is not true of value types or class types).

Defining Data

Data Types

- Descriptor types have more flexibility than other types when assigning one type to another or when passing a type as a parameter that's defined as another type.

In this classification, all other data types are *class types*. This includes `System.Object` itself, as well as `@class`, `@interface`, `@delegate`, `@value_type`, and any other class data type, such as `System.String`.

Classification of Synergy .NET Data Types		
Value types	Descriptor types	Class types
boolean	a	@*, System, System.Object
byte	i*	@ class
D_ADDR	asize	@ delegate
double	d	@ interface
float	d.	@ value_type
int	d*	enumeration
i1, i2, i4, i8 ^a	dsize	[#, #]value
long	dsize.precision	[#, #]@ *
sbyte	i*	
short	decimal	[#, #]@ class
structure	i	[#, #]@ value
ushort	i1, i2, i4, i8 ^a	
uint	n	
ulong	n.	
^VAL		

- a. When passed, i1, i2, i4, and i8 are descriptor types. Otherwise, Synergy .NET optimizes them by making them value types.

Alpha

An alpha value (**a**, **a***, **asize**) is a consecutive sequence of characters that can be treated as a unit of information. Alpha data can include only printable ASCII characters, and alpha values are limited to 65,535 characters on 32-bit platforms running Synergy standard.

- ▶ **a** defines an alpha parameter, return type, or method property.
- ▶ **a*** defines an alpha data field where the initial value of the field (which is required for **a***) is used to determine the field's size.
- ▶ **asize** defines an alpha data field, parameter, return type, or method property of the specified size. By default, **asize** values are filled with spaces.

See [“Where Data Types Can Be Used” on page 2-18](#) for more information.

Alpha types are descriptor types and have the same functionality for both Synergy standard and Synergy .NET.

Boolean

The boolean data type represents a Boolean (true/false) value. The default boolean value is false.

With Synergy standard, boolean maps to **i1**. A boolean value is evaluated as %TRUE if true, and as %FALSE if false.

With Synergy .NET, boolean is a value type and maps to the .NET Framework System.Boolean structure.

Byte

With Synergy standard, the byte data type represents an eight-bit signed integer and maps to **i1**.

With Synergy .NET, byte represents an eight-bit unsigned integer and maps to the .NET Framework System.Byte structure. The default for a byte value is 0.

If you are using byte as a signed integer in Synergy standard, we recommend that you use sbyte instead.

D_ADDR

The D_ADDR type is used for routine addresses. It can be used for data fields, parameters, return types, and method properties. (See [“Where Data Types Can Be Used” on page 2-18](#) for more information.) The default for D_ADDR is 0.

With Synergy standard, D_ADDR maps to **i4** for 32-bit systems or **i8** for 64-bit systems. With Synergy .NET, D_ADDR is a value type and maps to the .NET Framework System.IntPtr structure.

Decimal and implied-decimal

The **d**, **d***, and **dsize** types represent decimal data, which are signed, whole numbers that consist of ASCII numeric characters. The **d**., **dsize.precision**, and **decimal** types represent implied decimal data, which are signed numbers that consist of a whole number part and a fractional precision.

- ▶ **d** and **d.** define a decimal or implied decimal parameter, return type, or method property. These types are descriptor types and have the same functionality for both Synergy standard and Synergy .NET.
- ▶ **d*** defines a decimal data field where the initial value of the field (which is required for **d***) is used to determine the field's size. This is a descriptor type and has the same functionality for both Synergy standard and Synergy .NET.
- ▶ **dsize** and **dsize.precision** define a decimal or implied decimal data field, parameter, return type, or method property of the specified size. These are descriptor types and have the same functionality for both Synergy standard and Synergy .NET. The default value is an array of character "0" (zero) of the specified size.
- ▶ **decimal** defines an implied decimal data field, parameter, return type, or method property. With Synergy standard, **decimal** maps to **d28.10**, and the default value is an array of character "0" (zero) of size **d28**. With Synergy .NET, **decimal** is a descriptor type, maps to the .NET Framework System.Decimal structure, and has a default of 0.0.

See [“Where Data Types Can Be Used” on page 2-18](#) for more information, and note the following:

- ▶ A negative sign (–) doesn't take a data storage place (the sign is stored in the right-most digit with a “p” through “y”, corresponding to 0 through 9). And for implied-decimal data, the decimal point does not take a data storage place.
- ▶ Decimal data can have from 1 to 28 significant digits. For implied-decimal data, the maximum size of the whole number part is 28 significant digits, and the maximum size of the fractional precision is also 28 digits (**d28.28**).



The **f** designation for the fixed-point data type in previous versions of Synergy Language is still valid with Synergy standard, where it is processed as implied-decimal. Fixed-point is not supported in Synergy .NET.

Double

With Synergy .NET, **double** is a value type and maps to the .NET Framework System.Double structure. The default value is 0.0.

Double is not recommended for use with Synergy standard (where it maps to **d28.10** and has a default of 0.0).

Float

With Synergy .NET, float is a value type and maps to the .NET Framework System.Single structure. The default is 0.0.

Float is not recommended for use with Synergy standard (where it maps to **d28.10** and has a default of 0.0).

Integer

An integer value is a byte-oriented, binary representation of a signed whole number. Depending on how and where it is specified, an integer can be a value type or a descriptor type.

- ▶ **i** defines an integer parameter, return type, or method property. This is a descriptor type and has the same functionality for both Synergy standard and Synergy .NET.
- ▶ **i*** defines an integer data field where the initial value of the field (which is required for **i***) is used to determine the field's size. This is a descriptor type and has the same functionality for both Synergy standard and Synergy .NET.
- ▶ **i1**, **i2**, **i4**, and **i8** define one-, two-, four-, or eight-byte integers. The default value for these types is 0, and they accept the following range of values:

Type	Minimum value	Maximum value
i1	-128	127
i2	-32768	32767
i4	-2147483648	2147483647
i8	-9223372036854775808	9223372036854775807

These types have the same functionality for both Synergy standard and Synergy .NET. In general, they are descriptor types, but Synergy .NET does optimize them when possible (for example, for local data) by making them value types and mapping them to the .NET Framework System.Byte, System.Short, System.Int32, and System.Int64 structures.

- ▶ **int** (or **integer**) maps to **i4** with Synergy standard. With Synergy .NET, it is a value type and maps to the .NET Framework System.Int32 structure. The default is 0.

Note that these types are stored in native integer form, so data files that contain integer data (including ISAM files) are not portable between big-endian and little-endian machines. (To convert integer data between native and portable formats, use the %CNV_IP and %CNV_PI intrinsic functions. See [%CNV_IP on page 9-53](#) and [%CNV_PI on page 9-54](#) for more information.)

Defining Data

Data Types

Long

With Synergy standard, long maps to **i8**. With Synergy .NET, long is a value type and maps to System.Int64. The default value is 0.

Numeric

The numeric types, **n** and **n.**, define numeric parameters for subroutines, functions, and classes (with some exceptions; see [“Where Data Types Can Be Used” on page 2-18](#)). Numeric parameters can be used to pass any of the numeric data types: decimal, packed, or integer (for **n**), and implied decimal or implied packed (for **n.**). The parameter is accessed as the type used in the calling routine. (You can use ^DATATYPE to find the data type passed.)

Numeric types are descriptor types and have the same functionality for both Synergy standard and Synergy .NET.

Packed and implied-packed

Data in packed or implied-packed form (**p**, **psize**, **p.**, or **psize.precision**) is stored as two digits per byte, plus an extra byte for the sign.

- ▶ **p** and **psize** define a packed field, parameter, or return type. For **psize**, the default value is an array of character “0” (zero) of the specified size.
- ▶ **p.** and **psize.precision** define an implied-packed field, parameter or return type.

Whenever a packed or implied-packed field is used in an expression, it is converted to a decimal or implied-decimal field, respectively. Because of these restrictions, the only way to pass an expression to a subroutine or function in packed form is to assign it to a packed field before using it. (See [“Assignment Statements” on page 3-26](#) for information about assigning an expression to a packed field.)

Note that packed and implied-packed types cannot be used with classes and are not supported with the Synergy .NET compiler. See [“Where Data Types Can Be Used” on page 2-18](#) for more information.

Sbyte

With Synergy standard, sbyte maps to an **i1**. With Synergy .NET, the sbyte data type represents an eight-bit signed integer. It is a value type and maps to the .NET Framework System.Sbyte structure. The default value is 0.

Short

With Synergy standard, short maps to an **i2**. With Synergy .NET, short is a value type and maps to System.Int16. The default value is 0.

String

With the Synergy standard, string (or System.String) maps to the Synergy Language System.String class (see “[System.String](#)” on page 9-404). With Synergy .NET, it maps to the .NET Framework System.String class. In either case, the default value is ^NULL.

Note the following for Synergy standard:

- ▶ String can be used in most places that alpha can be used. In addition, it can be ranged and concatenated, and it can represent data that exceeds the maximum alpha size (64K on 32-bit machines).
- ▶ On a 32-bit machine, if the string length is less than or equal to 64K, storing a string to an alpha (avar=string), where the size of the alpha is smaller than the size of the string, operates the same as two alphas (for example, a10=a15) by silently truncating to the size of the destination variable.
- ▶ On a 32-bit machine, once the string size exceeds 64K, the string is no longer directly interchangeable with alpha data, and a “Map outside bounds of field or handle” error (\$ERR_HSIZEERR) is generated if you attempt to store or convert such a string to alpha. Therefore, if you use string data in the place of an alpha argument or variable, you must use ranging to ensure the resultant size does not exceed 64K.

^VAL

^VAL is a qualifier to subroutine parameters and return values. It specifies that the parameter or return value will be passed by value as a native integer.

With Synergy standard, ^VAL values map to **i4** for 32-bit systems and **i8** for 64-bit systems. With Synergy .NET, ^VAL is a value type and maps to the .NET Framework System.IntPtr structure.

VOID

VOID specifies that a method does not return a value.

@*, Object, System.Object

@*, Object, or System.Object is the parent class of all objects, including all other data types. With Synergy standard this maps to the Synergy Language System.Object class (see “[System.Object](#)” on page 9-406). With Synergy .NET, it maps to the .NET Framework System.Object class.

The default value is ^NULL.

@class

@class assigns an object reference to a field, parameter, return type, or method property. That class becomes the variable's type and you can use the object to access the class's functionality.

If the class identifier is not unique within all imported namespaces, you must qualify the identifier with its namespace. If it is unique, specifying the namespace is optional. For example, if **class1** is unique, a declaration using just **class1** will work:

```
my_object    ,@class1
```

If **class1** is not unique, the declaration might look like this:

```
my_object    ,@UserNS.class1
```

You can use *@class* in a named or unnamed record with Synergy standard or Synergy .NET. The default value is ^NULL.

@type

@type assigns a boxed type to a field, parameter, return value, or method property. In Synergy standard, *type* may be an non-class data type. In Synergy .NET, *type* may be any value or descriptor type. (See the table on [page 2-6](#) for how types are classified in Synergy .NET.)

You can use *@type* in a named or unnamed record with Synergy standard or Synergy .NET. The default value depends on the data type.

enumeration

The enumeration data type represents a set of related values. It has a name and one or more enumeration values associated with it. The compiler allows these values, or other integer values, to be assigned to a variable of the enumeration type. Enumeration values are represented by integer values, and with Synergy standard, the enumeration type maps to **i4**. With Synergy .NET, it maps to System.Enum. The default value is 0.

It is possible to assign an integer value (including Boolean) to an enumeration variable by casting the integer value to the enumeration type. A cast is also required when an integer is expected but an enumeration is given, for example, in a conditional such as

```
if (intvar .eq. (int)color.red)
```

If the enumeration identifier is not unique within all imported namespaces, you must qualify the identifier with its namespace. If it is unique, specifying the namespace is optional.

The enumeration type supports bitwise comparisons and the == and != operations. See [ENUM on page 4-48](#) for details on declaring an enumeration.

structure

A defined structure can be used as a data type for a parameter, return type, field, property, or local variable. The resulting construct is called a structfield. For example,

```
var1    ,mystruct
```

A structfield replaces group syntax to allow definition of a variable (or array) that is strongly prototyped when passed as a routine argument. It supersedes the syntax of using ^M with an alpha field. It is especially useful when used in conjunction with the System.Collections.ArrayList class to provide dynamic arrays of structures to replace using %MEM_PROC and ^M.

The following example shows how to declare a field as a structure and use that structure:

```
structure MyStructure
    first    ,a10
    last     ,a10    ,"Initial"
endstructure

subroutine TestStructure
stack record
    tt      ,i4
    person  ,MyStructure ;Declare a field that has the size of the
                        ; structure
proc
    init person                ;Clear first and set last to "Initial"
    person.first = "Joe"       ;Assign a value to a field in the structure
    open(tt=%syn_freechn, i, "tt:")
    writes(tt, person)
    close(tt)
endsubroutine
```

If the structure identifier is not unique within all imported namespaces, you must qualify the identifier with its namespace. If it is unique, specifying the namespace is optional.

Note that you can use structfields with Synergy standard or Synergy .NET, where *structfield* is a value type. Individual fields are initialized based on their type. See [“Using a structure with a structfield” on page 2-44](#) for more information.

[#,#]type

[#,#]type defines a dynamic array of the specified data type. In Synergy standard, *type* may be an non-class data type. In Synergy .NET, *type* may be any value or descriptor type. (See the table on [page 2-6](#) for how types are classified in Synergy .NET.) The [#,#] is shorthand for System.Array.

With Synergy standard, this maps to the Synergy System.Array class (see [“System.Array” on page 9-410](#)) and [#,#] can have 1 to 9 strides, inclusive ([#], [#,#], and so on, up to [#,#,#,#,#,#,#,#]). For example,

```
my_str_array ,[#,#]struct1
```

Synergy standard does not support `[#]d28.10` or `[#]float` (in other words, an array of `d28.10` or an array of float, which becomes an array of `d28.10`).

With Synergy .NET, this maps to the .NET Framework `System.Array` class.

The default value is `^NULL`.

`[#, #]@type`

`[#, #]@type` defines a dynamic array in which the element is the specified boxed data type. In Synergy standard, *type* may be a non-class data type. In Synergy .NET, *type* may be any value or descriptor type. (See the table on [page 2-6](#) for how types are classified in Synergy .NET.) The `[#, #]` is shorthand for `System.Array`.

With Synergy standard, this maps to the Synergy `System.Array` class (see “[System.Array](#)” on [page 9-410](#)) and `[#, #]` can have 1 to 9 strides, inclusive (`[#]`, `[#, #]`, and so on, up to `[#, #, #, #, #, #, #, #, #]`).

With Synergy .NET, this maps to the .NET Framework `System.Array` class.

The default value is `^NULL`.

`[#, #]@class`

`[#, #]@class` defines a dynamic array of the specified class. (The `[#, #]` is shorthand for `System.Array`.)

With Synergy standard, this maps to the Synergy `System.Array` class (see “[System.Array](#)” on [page 9-410](#)) and `[#, #]` can have 1 to 9 strides, inclusive (`[#]`, `[#, #]`, and so on, up to `[#, #, #, #, #, #, #, #, #]`). For example,

```
my_dyn_array , [#, #, #]@system.string
```

With Synergy .NET, this maps to the .NET Framework `System.Array` class.

The default value is `^NULL`.

`[#, #]@*`

`[#, #]@` defines a dynamic array of any type of object. (The `[#, #]` is shorthand for `System.Array`.)

With Synergy standard, this maps to the Synergy `System.Array` class (see “[System.Array](#)” on [page 9-410](#)) and `[#, #]` can have 1 to 9 strides, inclusive (`[#]`, `[#, #]`, and so on, up to `[#, #, #, #, #, #, #, #, #]`).

With Synergy .NET, this maps to the .NET Framework `System.Array` class.

The default value is `^NULL`.

Rules for Passing One Type to Another

You can assign the following types interchangeably in an assignment statement. Implicit conversions are performed.

- ▶ *a*, *asize*
- ▶ *i*, *i1*, *i2*, *i4*, *i8*, *byte*, *short*, *int*, *long*, *IntPtr*, *sbyte*, *ushort*, *uint*, *ulong*
- ▶ *d*, *dsize*, *d.*, *dsize.precision*, *decimal*, *double*, *float*
- ▶ *n*, *n.*
- ▶ *p*, *psize*, *p.*, *psize.precision*
- ▶ *String*

The string type has the same implicit conversions as the alpha type. Booleans can be assigned to or from any of the signed integer types.

You can pass the following types interchangeably as an argument to an IN numeric parameter or a parameter without a specified direction:

- ▶ *i*, *i1*, *i2*, *i4*, *i8*
- ▶ *d*, *d.*

Int arguments can only be passed to **int**, **n**, or **n.**

The default passing convention is BYREF (or BYVAL on .NET). For BYREF, types must match exactly for the following types:

- ▶ *byte*, *short*, *int*, *long*, *IntPtr*, *sbyte*, *ushort*, *uint*, *ulong*
- ▶ *dsize*, *dsize.precision*, *decimal*, *double*, *float*
- ▶ *n*, *n.*
- ▶ *p*, *psize*, *p.*, *psize.precision*

You can pass the following types interchangeably as an argument to an IN parameter:

- ▶ *a*, *asize*, *string*

You can also pass an **a** type to an **n** type (or a **d** or **d.** type to an **a** type for subroutines and functions) if the parameter has the MISMATCH modifier.

Defining Data

Rules for Passing One Type to Another

You can assign the following types to the listed Synergy object type in an assignment statement or as the value of an IN parameter:

Destination type	Source type
@class	An instance of <i>class</i> , or a derived class, or the cast of a class that has an explicit operator to convert to the destination type (see “Conversion operators” on page 3-18 for more information)
@*	Any class instance, any interface (Synergy .NET only), or any boxed value type
@interface	The same interface, or a cast of another interface or class variable to this interface (Synergy .NET only)
@delegate	An instance of the same delegate, or a cast of a system object variable or method to this delegate (Synergy .NET only)
structure	Only the specified structure
enumeration	The same enumeration, one of the enumeration values, or a cast of another enumeration type or integer to this enumeration

You can cast a variable to a specific type to perform an explicit conversion between the types. (See [“Conversion operators” on page 3-18](#) for user-defined explicit conversions for classes.) For example,

```
subroutine test
    myshort, i2
proc
    myshort = 7
end

subroutine test2
record
    myshort, i2
    myint, i4
proc
    myshort = %int(myint,2)          ;convert to i2
    xcall test( %int(myint,2) )      ;convert to short
end
```

The difference between casting and converting is that while casting simply looks at the data as if it were a certain type without modifying its contents, converting changes the content to an appropriate representation of the semantic value of the original for the new type. For example, ^A (or %A) performs a cast to alpha type, while %STRING performs a conversion to alpha. Passing a value into an argument that is declared as alpha performs an implicit cast to alpha; assigning a value to an alpha variable invokes an implicit conversion to alpha.

If you name a class record or a routine record, you can use that name to reference all the fields as one record for use in ISAM routines as one contiguous block of memory whose size is based on the sum of the declared fields' sizes.

Where Data Types Can Be Used

Data types are used in fields, parameters, and return values. The following table shows where each type can be used. The term “class” refers to object-oriented code compiled with the standard Synergy compiler. “Nonclass” refers to non-object-oriented code. “.NET” refers to code compiled with the Synergy .NET compiler.

Where Data Types Can Be Used									
Data type	Class			Nonclass			.NET		
	field	parameter	return value	field	parameter	return value	field	parameter	return value
a		X	X		X	X		X	X
a*	X		X	X		X	X		X
asize	X	X	X ^a	X	X	X ^a	X	X	X ^a
boolean	X	X	X	X	X	X	X	X ^b	X
byte	X	X	X	X	X	X	X	X ^b	X
D_ADDR	X	X	X	X	X	X	X	X	X
d		X	X		X	X		X	X
d*	X		X	X		X	X		X
dsiz	X	X	X ^a	X	X	X ^a	X	X	X ^a
d.		X	X		X	X		X	X
dsiz.precision	X	X	X	X	X	X	X	X	X
decimal	X	X	X ^c	X	X	X	X	X ^b	X
double	X ^d	X ^d	X ^{cd}	X ^d	X ^d	X ^{cd}	X	X ^b	X
float	X ^d	X ^d	X ^{cd}	X ^d	X ^d	X ^{cd}	X	X ^b	X
i	X	X	X	X	X	X	X	X	X
i*	X		X	X		X	X		X
i1	X	X	X	X	X	X	X	X	X
i2	X	X	X	X	X	X	X	X	X

Where Data Types Can Be Used									
Data type	Class			Nonclass			.NET		
	field	parameter	return value	field	parameter	return value	field	parameter	return value
i4	X	X	X	X	X	X	X	X	X
i8	X	X	X	X	X	X ^e	X	X	X
int	X	X	X	X	X	X	X	X ^b	X
long	X	X	X	X	X	X ^e	X	X ^b	X
n		X ^f			X			X	
n.					X			X	
p				X	X	X			
psize				X	X	X			
p.				X	X	X			
psize.precision				X	X	X			
sbyte	X	X	X	X	X	X	X	X ^b	X
short	X	X	X	X	X	X	X	X ^b	X
string	X	X	X ^c	X	X	X	X	X ^b	X
ushort							X	X ^b	X
uint							X	X ^b	X
ulong							X	X ^b	X
^VAL			X			X			X
VOID			X ^g						X
@*	X	X ^b	X ^c	X	X ^b	X	X	X ^b	X
@class	X	X ^b	X ^c	X	X ^b	X	X	X ^b	X
@delegate							X	X ^b	X
@interface							X	X ^b	X

Where Data Types Can Be Used									
Data type	Class			Nonclass			.NET		
	field	parameter	return value	field	parameter	return value	field	parameter	return value
@ type	X	X ^b	X	X	X ^b	X	X	X ^b	X
structure	X	X ^b	X	X	X ^b	X	X	X ^b	X
enumeration	X	X ^b	X	X	X ^b	X	X	X ^b	X
[#,#]value	X	X ^b	X ^c	X	X ^b		X	X ^b	X
[#,#]@ value	X	X ^b	X ^c	X	X ^b		X	X ^b	X
[#,#]@ class	X	X ^b	X ^c	X	X ^b		X	X ^b	X
[#,#]@ *	X	X ^b	X ^c	X	X ^b		X	X ^b	X
group ^h	X	X		X	X		X	X	

- a. On return value, size ignored.
- b. Cannot be declared as an optional parameter.
- c. For methods and properties only.
- d. Not recommended for use with Synergy standard.
- e. Not supported on 32-bit.
- f. Excluding methods that do not have the UNIQUE modifier.
- g. For methods only.
- h. Technically not a data type but can be declared as a field or passed as a parameter.

Field definitions

A field defines a location in memory. A field can either be an unnamed filler that merely occupies space or a named variable that holds a data value to be processed by your program. You can use fillers when a record has many fields, but a routine uses only a small subset of the fields. Only the used fields must be named; many developers combine adjacent unreferenced fields into single filler fields to shorten their record definitions. (Note, however, that the side effect of creating merged filler fields is that your entire record structure is not fully documented in the program listing.)

For all types on Synergy standard and for descriptor types on Synergy .NET, named records, named commons, named literals, groups in unnamed records, structures, fields, static array elements, subscripts of memory handles, and string variables cannot be larger than 65,535 bytes on 32-bit systems. You can specify the same field name in more than one named record or class, but you must reference that field with a unique path specification. (See [“Variable path specifications” on page 1-24](#) for more information.)

For example:

```
record consignee
    name      ,a40
    .
    .
    .

record consigner
    name      ,a40
    .
    .
    .

proc
    consignee.name = "Jones"
    consigner.name = "Smith"
```

Fields have several characteristics: access type, name, size, type, position within the record, initial value, and number of optional array elements. Each field definition specifies the characteristics of a single field, and the entire collection of field definitions comprises the record's data layout.

A field definition has the following format:

[access] [field_mod] [name], [dimension | [dim, ...]]type[size] [@position][, init_value, ...]

Field definition components

access

(optional) For a class field only, one of the following access modifiers:

PUBLIC	Access is not restricted. This is the most accessible option.
PROTECTED	Access is limited to the containing class or types derived from the containing class.
PRIVATE	Access is limited to the containing type. This is the least accessible option. (default)

field_mod

(optional) For a class field only, one of the following field modifiers:

CONST	A static field whose value is not changed once it is initialized. CONST implies STATIC.
READONLY	The field's value can only be set during declaration or in the constructor of the class.
STATIC	The field's value can be accessed (read and write) even if the class that contains the field hasn't been instantiated.

Defining Data

Where Data Types Can Be Used

name

(optional) A name used to reference a field's data space. If the field is a member of an unnamed class record, a field name must be specified.

dimension

(optional) Indicates the number of times the field occurs. The default value is 1. The presence of *dimension* indicates a pseudo array. Although pseudo arrays are supported for backwards compatibility, they are considered deprecated in favor of real arrays. See [“Defining arrays” on page 2-24](#) for more information.

dim

(optional) Indicates the number of times the field occurs. The presence of *dim* indicates a real or dynamic array. You must type the brackets around the dimension elements; they do not mean that the arguments are optional. The brackets must contain at least one element.

- ▶ For real arrays, *dim* is one or more decimal expressions, separated by commas, that indicate the number of dimensions.
- ▶ For dynamic arrays, *dim* is one or more # characters, separated by commas, that indicate the number of strides. For example, a dynamic array of three strides would be specified as [#,#,#]. The actual sizes are specified using the NEW keyword on a System.Array at runtime.

See [“Defining arrays” on page 2-24](#) for more information on defining real and dynamic arrays.

type

One of the following data types. (For more information on these types see the table on [page 2-2](#); there are detailed descriptions of each type following the table.)

a

boolean

byte

d, d.

decimal

D_ADDR

double (Synergy .NET only)

float (Synergy .NET only)

i

int

intptr

long

p, p. (Synergy standard only)

sbyte

short

string

ushort (Synergy .NET only)

uint (Synergy .NET only)
ulong (Synergy .NET only)
@*
@class
@interface (Synergy .NET only)
@delegate (Synergy .NET only)
enumeration
structure
[#,#]value
[#,#]@value
[#,#]@class
[#,#]@*

size

(optional) For nonclass types, an expression that indicates the number of characters in each occurrence of the field.

If *type* is **d**. or **p**., the *size* must be specified in the form *size.places*, where *size* is the length of the entire field and *places* is the number of digits to the right of the decimal point. For example, **d5.3** is an implied-decimal field that is five digits long, with the last three digits to the right of the decimal point. *Size* cannot be greater than 28 for **d**. or 18 for **p**.. *Places* cannot be greater than 28 for **d**. or 10 for **p**., or greater than the value of *size*.

If *type* is **i**, *size* can only have a value of 1, 2, 4, or 8.

If *type* is **d**, *size* cannot be greater than 28.

If *type* is **p**, *size* cannot be greater than 18.

Size is optional and can be an asterisk (*) if you specify an initial value list. The size is the size of the largest initial value specified. See the Discussion section for more information.

position

(optional) For nonclass types, the position of the field either as an absolute character position or relative to the beginning of the current record. The *position* specification is described in more detail in [“Specifying a field position indicator” on page 2-27](#).

init_value

(optional) For nonclass types, an initial value to be assigned to successive occurrences of the field. The *init_value* specification is described in more detail in [“Specifying an initial value” on page 2-29](#).

Discussion

The `STATIC` modifier can only be specified on a field that is not defined in a common, literal, structure, or nonstatic record. If the owning record is static, then `STATIC` is the default *field_mod*.

If you don't specify *name*, you can still reference the data space by ranging the record variable of which the field is a part. If you don't specify a record name either, the field's contents won't be directly accessible. (See [“Ranged references” on page 1-32](#) and [“Relative ranging” on page 1-33](#) for more information.)

If you've specified a dimension for a nonclass field, the size must be a positive decimal literal. You can either omit the size or assign a size of `“*”` to indicate that the size of the field should be the size of the initial value assigned to that field. (If you specify a size of `“*”`, make sure that you also assign an initial value. If you don't, you'll get a syntax error.) This feature is most useful for fields that hold strings of undetermined size (such as title information), because you don't have to count characters—the compiler counts the characters for you.

Initial values are not allowed on `STACK` records.



An `i4` variable is more efficient than `i1` or `i2`. You can also use the compiler directive `.ALIGN LONG` before an `i4` variable to increase its efficiency.

Throughout this section, wherever we refer to a decimal expression, you can specify any expression that can be completely evaluated at compile time. (See [“Compile-Time Expressions and Optimizations” on page 3-41](#) for details.) When using decimal expressions, you may need to separate one syntax element from another using a space or parentheses so the compiler knows how to interpret each element correctly. For example, if you specify the data type and size as follows:

```
d2*^size(fld)
```

the compilation fails with a syntax error, because the compiler tries to interpret the number 2 as the field size. However, if you include a space:

```
d 2*^size(fld)
```

or parentheses:

```
d(2*^size(fld))
```

the compiler knows that `2*^size(fld)` is an expression representing the field size.

Defining arrays

Often a particular field must occur several times within a record structure, with each occurrence representing a separate data element. Any field that is defined to occur more than once is called an array. The data area allocated to an array is the product of the number of elements defined and the size of a single element. For example, a six-character field defined as having 10 elements is allocated a total of 60 characters of space in memory (10 x 6).

You can define an array as either pseudo (without brackets), real (with brackets and implicitly created at compile time), or dynamic (with brackets and explicitly created dynamically at runtime). Both pseudo and real arrays are also referred to as static arrays.

All array elements are initially set to their default value based on data type. See the table on [page 2-2](#) for a list of default values.

Static (pseudo and real) arrays

A pseudo array is a one-dimensional array of type **a**, **d**, **d.**, **i1**, **i2**, **i4**, **i8**, **p**, or **p.**. A real array also consists of the same types with the addition of structure, but it can be defined with multiple dimensions.

Here are some examples:

Pseudo array of alpha	<code>mparr, 10a5</code>
One-dimensional real array of alpha	<code>mrarr, [10]a5</code>
One-dimensional real array of structfields	<code>mrarr, [16]MyStruct</code>
Two-dimensional real array of decimal	<code>mrarr, [6,4]d8</code>

Pseudo arrays cannot be defined within a class. They are considered deprecated in favor of real arrays, and we do not recommend using them, for compatibility with .NET and because there is no runtime bounds checking even when compiling with **-qstrict**. Pseudo array bounds checking only occurs with **-qcheck**.

Dynamic arrays

A dynamic array is an array whose size is determined at runtime. There are two dynamic array types: Array and ArrayList.

An Array is a one- or multi-dimensional array of any descriptor, value, or reference data type. You can declare a dynamic array whose upper bounds are determined at runtime by specifying a “#” for each array dimension. The declared variable will be type System.Array.

An ArrayList is a one-dimensional array of reference types. You can declare an ArrayList as either a System.Collections.ArrayList or a Synergex.SynergyDE.Collections.ArrayList whose element type and size is determined at runtime. (See “[System-Supplied Classes](#)” on [page 9-403](#) for more information about these classes.)

Here are some examples:

One-dimensional Array of objects	<code>marr, [#]@MyClass</code>
One-dimensional Array of integers	<code>marr, [#]i4</code>
Two-dimensional Array of structfields	<code>marr, [#,#]MyStruct</code>
One-dimensional ArrayList of objects	<code>marrl, @System.Collections.Array</code>

Defining Data

Where Data Types Can Be Used

In addition, any descriptor type or value type may be specified as a boxed type in an Array. (See “[Boxing](#)” on page 8-19.) For example,

One-dimensional Array of boxed alphas marr, [#]@a

The runtime upper bound for a System.Array is determined when you create the array using the NEW keyword. When an array of classes is created using the NEW keyword, each of the entries is initialized to ^NULL, unless initial values are specifically stated. For example,

```
marr = new MyClass[5]           ;Create an array of 5 ^NULL object handles
marr = new MyClass[5,7]         ;Create an array of 5*7 ^NULL object handles
marr = new i4[100]              ;Create an array of 100 integers
marr = new @a[20]               ;Create an array of 20 boxed alpha handles
```

To specify initial values for a dimensioned NEW operation, enclose the initial values in curly braces. For example,

```
marr = new i4[#] {1, 2, 3}
```

Unlike static arrays, dynamic arrays can be used as return types.

You can specify multiple arrayed items in a dynamic array by declaring a bracket pair for each array. For example,

```
marr, [#][#]@MyClass
marr = new MyClass[3][#]
marr[1] = new MyClass[2]
marr[2] = new MyClass[2]
marr[3] = new MyClass[2]
```

This declares an array of arrays, whose sizes are determined at runtime. For example, if the method **meth1** was declared as

```
method meth1, void
    p1, [*]i4                ;Indicates it expects an array
```

and a local variable was declared as

```
myvar, [2]i4
```

then to pass **myvar** with its dimension information, you would use this syntax:

```
obj.meth1(myvar)
```

The ArrayList type is not truly an array but is actually a collections class that includes indexers for array-like access and methods for adding and removing elements. Unlike Array, the size of an ArrayList can also be dynamically changed at runtime. See [System.Collections.ArrayList](#) on page 9-414 or [Synergex.SynergyDE.Collections.ArrayList](#) on page 9-417 for more information.

Specifying a field position indicator

By default, Synergy Language allocates data for a field beginning at the end of the current record, extending the record by the size of the field. Sometimes, however, you'll want a field to begin at a specific point, or offset, in the record or group area. To begin a field at an offset, your field definition must include a position indicator (the *position* argument on [page 2-23](#)). A position indicator is always preceded by an "at" sign (@).

The data area referenced by the position indicator is the area that begins at the specified offset. If you specify an offset in an overlay record, the field must not extend beyond the end of the record being overlaid. If it does, you'll get an "Cannot extend record with overlay field" compilation error. If the field extends beyond the data allocation for a nonoverlaid record, and does not overlay another field, Synergy Language extends the record to include the newly defined field.

You can specify a position indicator for a field in a literal record only if the literal record is named or the field exists in a GROUP/ENDGROUP block.

You can specify a field's starting position in one of two forms: absolute or relative. In the absolute form, a single decimal value represents the actual character position within the record or group at which the field begins. For example, a starting position of @2 means that the field begins at the second character position within the record or group.

In the relative form, the starting position is specified relative to the beginning of another record or field. The relative form has the following syntax:

name [+ / -*offset*]

name

Either the name of the record or group currently being defined, the name of a previously defined record being overlaid, or the name of a previously defined field (in the current record or group, or in the record being overlaid).

offset

(optional) The number of characters from the beginning of *name* at which the field begins. *Offset* can be preceded by either a plus (+) or a minus (–), depending on where the field begins.

The following example shows the correct use of field position indicators:

```
record cusrec                                ;Master customer record
    name          ,a30
    address       ,a50
    company       ,a30
    phone         ,a12
    cdate         ,d6                        ;Next contact date
    day           ,d2 @cdate
    month         ,d2 @cdate+2
    year          ,d2 @cdate+4
```

Defining Data

Where Data Types Can Be Used

```
record report_header           ;Heading for report
    id      ,a12 @1,          "Inventory ID"
    onhand   ,a10 @20,         "Qty On-hand"
    commtd   ,a9  @35,         "Committed"
    order    ,a8  @50,         "On Order"
```

In the first record, the **day** field begins at the first character of **cdate**, the **month** field begins at the third character of **cdate**, and the **year** field begins at the fifth character of **cdate**. The layout for the second record creates blank fields between the defined fields.

Notice that if you add any fields to **report_header**, you must verify that the absolute field positions haven't changed.

The following two examples are *not* valid. In the first example, the field specified in the field position indicator (**key**) has not yet been defined. In the second example, the specified record (**rec1**) is not the record that is being overlaid.

```
record client                  ;Invalid field overlays
    company ,a2    @key
    clnt_id  ,a6    @key+2
    key      ,a8
```

and

```
record rec1
    f1      ,a10
record rec2
    f2      ,a10
record ,x
    f3      ,a3  @rec1+3
```

The example below breaks a field down into subfields:

```
record list
    name      ,a25
    phone     ,a14, "(XXX) XXX-XXXX"
    area      ,d3  @phone+1
    prefix    ,d3  @phone+6
    pnum      ,d4  @phone+10
```

The **phone** field name can reference the entire 14-character field. In addition, **area** references the second through fourth characters of **phone**, **prefix** references the seventh through ninth characters, and **pnum** references the last four characters.

The following example shows how easy it is to code report titles and even line layouts directly from conventional report layout forms. All you need to worry about is the beginning and ending positions of each report field; you don't need to count spaces or define unused blank sections for each report line.


```
record title
    ,a*    @1,    "Inventory ID"
    ,a*    @20,   "Qty On-hand"
    ,a*    @35,   "Committed"
    ,a*    @50,   "On Order"

record line
    id      ,a12   @1
    onhand  ,a10   @20
    commtid ,a9     @35
    order   ,a8     @50
```

Specifying an initial value

When program execution begins, the default value of an alpha field is blank, the default value of a decimal or implied-decimal field is 0, and the default value of an integer field or a packed or implied-packed field is 0. If you specify an initial value in your field definition, Synergy Language sets the field to that value before executing any statement in the procedure division.



Initial values are not set on stack records, structures, or local data fields. However, objects are guaranteed to be reset to ^NULL. If the INIT statement is used, the initial values specified for a structure or stack record are then used. Initial values on structures do apply to structfields and complex arrays of structfields.

Be careful when you specify initial values for local records and subroutines. The value is assigned on the first invocation of the subroutine, but on further invocations, the variable may contain the resulting value of the subroutine instead of the initial value. Your program is responsible for initializing the data in global data sections.

If you specify initial values for an array, Synergy Language assigns those values to each element successively, beginning with the first. You can provide initial values for one or more array elements as necessary; if you specify an initial value for one array element, you aren't obligated to specify values for all elements. Synergy Language assigns the default value to trailing array elements with no initial value.

Initial values for alpha fields must be alpha expressions, while initial values for numeric fields must be compile-time numeric expressions. Initial values are loaded into a field element according to the rules for storing data into that field's data type. See [“Assignment Statements” on page 3-26](#) for information about storing data into different data types.

If you specify an initial value that is longer than a single element of the field, you'll get an “Initial value too long” syntax error. For example, you can't specify a six-character alpha literal as an initial value for a field with a declared size of 5.

If an initial value for an integer variable exceeds the maximum signed 64-bit value, the compiler generates an error. An error is also generated if an automatically sized literal exceeds a 32-bit value. If the initial value does not exceed this maximum, the compiler automatically sizes it as an **i4**.

Defining Data

Where Data Types Can Be Used

You can only specify initial values for fields that allocate data space. Once the data space is allocated, the fields that later overlay the allocated area can't have initial values. Because the overlay indicator is ignored on external common fields, initial values are allowed on such fields, but they, too, are ignored (except to determine the size of automatically sized fields).

You can specify an initial value for a field with a field position indicator but only if the position in the record doesn't overlay any data that was previously declared.

Consider the following data division statements:

```
record
    f1      ,a30
    f2      ,a5    @f1+3
    f3      ,a15
    f4      ,a5    @f3+20
    f5      ,a2    @100
record ,x
    f01     ,a5
    f02     ,a3    @f2+2
```

Fields **f1** and **f3** can have initial values, because they are part of a nonoverlay record declaration and no field position indicators are associated with them, which means that they allocate data space. Although fields **f4** and **f5** have a field position indicator, they are also allowed to have initial values, because the field position for each goes beyond the data space that is already allocated and requires the allocation of additional data space.

Fields **f01** and **f02** cannot have initial values because they are part of an overlay record declaration; they merely specify alternate ways of viewing previously allocated data space. Field **f2** also references previously allocated data space and therefore cannot have an initial value.

Synergy Language ignores any initial values specified for variables in an EXTERNAL COMMON or EXTERNAL LITERAL definition; such variables only reference data space that was previously allocated by a GLOBAL COMMON or GLOBAL LITERAL statement.

All initial values are ignored except those in the initializing routine. To specify an initial value, use the INIT option on the GLOBAL statement. (See [“Specifying an initial value” on page 2-29](#) for more information about INIT.)

Examples

Here are some sample field definitions:

```
record
    ,a3,          "XYZ"
    inval      ,a*,          "This is the initial value"
    fld1       ,2d7,          123,987
    fld2       ,[2,2,2]d3.1,  11.1, 11.2, 12.1, 12.2,
&              21.1, 21.2, 22.1, 22.2
    date       ,d6,          32792
```

```
month      ,d2 @date
day        ,d2 @date+2
year       ,d2 @date+4
```

The following example further illustrates the three ways to access arrays. First, let’s assume the following data division statements:

```
record
  array1      ,4a6,          "ABCDEF",  "GHIJKL",
  &           "MNOPQR",    "STUVWX"
  array2      ,3d3,          123,   4,   789
.align LONG
  array3      ,[3,2]d2,      11,    12,
  &           21,    22,
  &           31,    32
```

The following data space is allocated from left to right and from top to bottom (shaded boxes are not part of the data space):

Variable	Data area (characters)					
array1(1)	A	B	C	D	E	F
array1(2)	G	H	I	J	K	L
array1(3)	M	N	O	P	Q	R
array1(4)	S	T	U	V	W	X
array2(1)	1	2	3			
array2(2)	0	0	4			
array2(3)	7	8	9			
alignment filler						
array3[1,1]	1	1				
array3[1,2]	1	2				
array3[2,1]	2	1				
array3[2,2]	2	2				
array3[3,1]	3	1				
array3[3,2]	3	2				

Here are some examples of variable references and the data they obtain:

Variable reference	Data obtained
array1(1)	ABCDEF
array1(3)	MNOPQR
array2	123
array2(7,7)	7
array1(array2(6:1),12)	DEFGHIJKL
array3[2,2]	22
array3[2,2] (3:4)	3132

Parameter definitions

Parameters can be declared for subroutines, functions, and methods. A parameter definition has the following format:

[param_mod] param_def [dimension]type[size]

Parameter definition components

param_mod

(optional) One or more of the following parameter modifiers. See the list of mutually exclusive modifiers on [page 2-35](#). The default for subroutines and functions is unspecified and for methods is INOUT.

BYREF	Argument passed by descriptor in Synergy standard or by reference in Synergy .NET (default)
BYVAL	Argument passed by value (default for descriptor types that are not marked OUT or INOUT for Synergy .NET)
IN	Read only
INOUT	Read and write
OUT	Write only
MISMATCH	Allow an inappropriate ^ARG function to access the parameter value, allow a type variables to be passed to n parameters (and vice versa for a subroutine or function), and allow d type variables to be passed to a parameters
OPT/IONAL/	Optional argument. (default for a subroutine or function)
REQ/UIRED/	Required argument (default for a method, object, or dynamic array parameter)

param_def

The identifier used to reference the parameter. *Param_def* has one of the following formats:

<i>name</i>	Parameter name
^DESCR (<i>name</i>)	Argument passed by descriptor (default)
^REF (<i>name</i>)	Argument passed by reference (<i>size</i> is required and <i>type</i> cannot be n). Not supported in Synergy .NET.
^VAL (<i>name</i>)	Argument passed by value (<i>type</i> must be integer)
<i>group_def</i>	Group definition (see GROUP-ENDGROUP on page 4-86 for syntax)

Parameter names adhere to the Synergy Language naming conventions for identifiers. (See [“Identifiers”](#) on [page 1-20](#).)

Defining Data

Where Data Types Can Be Used

dimension

(optional) One or more array dimensions for the parameter type, designated in one of the following ways:

[*] (or [*,*], etc.) Fixed array

[#] (or [#,#], etc.) Dynamic array

(*) Pseudo array (If the **-qnet** compiler option is specified, only arrays can be passed to arguments marked with (*). If **-qnet** is not specified, (*) is ignored.)

type

One of the following data types. (For more information on these types see the table on [page 2-2](#); there are detailed descriptions of each type following the table.)

a

boolean

byte

d, d.

decimal

D_ADDR

double (Synergy .NET only)

float (Synergy .NET only)

i

int

intptr

long

n, n.

p, p. (Synergy standard only)

sbyte

short

string

ushort (Synergy .NET only)

uint (Synergy .NET only)

ulong (Synergy .NET only)

@*

@class

@interface (Synergy .NET only)

@delegate (Synergy .NET only)

enumeration

structure

[#,#]value

[#,#]@value

[#,#]@class

[#,#]@*

size

The length of the parameter. *Size* is required only when ^REF is used.

Discussion

Synergy Language enables you to pass arguments to subroutines and functions in three ways: by descriptor (BYREF or ^DESCR), by reference (^REF), and by value (BYVAL or ^VAL). Arguments to Synergy Language routines are normally passed by descriptor, which is the default. ^REF and BYVAL/^VAL arguments are used primarily when passing arguments to non-Synergy Language routines, but they may also be used with Synergy Language routines. When using ^REF or BYVAL/^VAL to pass arguments to Synergy Language routines, the argument must be passed as ^REF or BYVAL/^VAL *and* declared as ^REF or BYVAL/^VAL in the receiving routine.

The primary difference between these three ways of passing arguments lies in how the value of the argument is treated if it is updated by the routine. When an argument is passed by descriptor (BYREF/^DESCR), if the variable associated with the parameter in the receiving routine is modified, upon returning from the routine, the value of the argument in the calling routine is also updated. A ^REF argument is treated the same way. However, when an argument is passed as BYVAL/^VAL, the value is changed only locally—not in the calling routine. In other words, passing a reference type BYVAL/^VAL allows members of the caller's type to be changed but the reference to the type cannot be changed. Passing a reference type BYREF/^DESCR allows both the reference and the members of the caller's type to be modified. (See Examples on [page 2-37](#).)

A call to a routine declaring a REQUIRED (or REQ) modifier must provide a value for that argument. A call to a routine declaring an OPTIONAL (or OPT) argument can omit a value for that argument. ^VAL arguments are automatically marked required. Some data types cannot be declared as optional parameters; see the table on [page 2-18](#) for details.

The MISMATCH modifier is allowed on an **n**, **n.**, **a**, or **d** parameter type and enables the program to pass an alpha type as an argument to a numeric type without a prototype mismatch error. For subroutines and functions, MISMATCH is also allowed on an **a** parameter type, which enables the program to pass a decimal or an implied-decimal argument to an alpha parameter.



In Synergy standard, local **n** parameters implicitly act like MISMATCH **n** parameters, while routines resolved via prototype require that MISMATCH be explicitly stated.

The following parameter modifiers are mutually exclusive:

- ▶ BYVAL/^VAL and BYREF/^DESCR and ^REF
- ▶ IN and OUT and INOUT
- ▶ BYVAL and OUT or INOUT
- ▶ OPTIONAL and REQUIRED

Defining Data

Where Data Types Can Be Used

If *param_def* is a group definition, the parameter can have subfields that are explicitly sized and typed. An example of a group parameter is shown below:

```
subroutine mysub
arg1                ,a10
group arg2          ,a
    arg2_field1     ,d3
    arg2_field2     ,d4
endgroup
arg3                ,a8
```

When defining object handles (*@class*), note the following restrictions:

- ▶ Object handles are not allowed within the scope of a group. For example, the following is invalid:

```
public method mymethod, void
    group p1
        h1 ,@myclass
    endgroup
```

- ▶ Object handles are not allowed in local records that are declared in REENTRANT routines or in common records or global data sections.
- ▶ Object handles are REQUIRED by default and cannot be defined as OPTIONAL.

For a dimensioned parameter (in other words, an array), the number of dimensions is indicated by asterisks (for a fixed array) or pound signs (for a dynamic array) separated by commas and enclosed in square brackets. The number of dimensions defined in the argument must match that defined in the calling routine. The number of elements in each dimension is not specified in the routine argument; rather, it is defined in the calling routine.

The following example shows how to declare two-dimensional fixed and dynamic arrays in a subroutine.

```
subroutine mysub
    parm1      ,[*,*]a      ; fixed alpha array
    parm2      ,[#,#]@myclass ; dynamic array of myclass objects
```

If you pass a dimensioned variable to a parameter that has not been defined as an array, the routine will be able to access only the first element of the array. Should this variable then be passed to a second routine where the parameter *is* defined as an array (with the correct number of dimensions), the second routine will be able to access the entire array.

Any numeric type (**i**, **d**, **p**, **n**, and so forth) can be passed to an **n** type.



When strong prototyping is enabled for routines marked IN or for which no directional modifier is specified, and the type is **n** or **i**, the compiler converts numeric literals to integers for improved performance.

If a parameter is a structfield, with strong prototyping, only a structfield of the same structure type can be passed.

Unless ^REF is specified, *size* is ignored on parameters, as it is defined by the caller. However, you may want (or, in some cases, need) to include *size* if you plan to attribute code for use with *xjServerPlus*. See “[Using Attributes to Define Synergy Methods](#)” in the “Defining Your Synergy Methods” chapter of the *Developing Distributed Synergy Applications* manual.

See also

- ▶ “[Data types](#)” on page 1-10 for more information about Synergy Language data types.
- ▶ “[Field definitions](#)” on page 2-20 for more information about field syntax in the calling routine.
- ▶ [GROUP-ENDGROUP](#) on page 4-86 for more information about defining group fields.

Examples

The following example illustrates the difference between ^REF and ^VAL arguments.

```
main
record
    ttchn      ,i4      ,1
    var1       ,i4
    var2       ,D_NATINT

proc
    var1 = 10                ;var1 and var2 are both 10 to start with
    var2 = 10

    xcall mysub(^ref(var1), ^val(var2)) ;pass var1 by reference
                                         ;pass var2 by value
    open(ttchn, o, "tt:")
    writes(ttchn, "^ref is " + %string(var1) + " ^val is " + %string(var2))
    close ttchn
endmain

subroutine mysub
    ^ref(arg1)    ,i4      ;declare arg1 as ^ref in routine
    ^val(arg2)    ,i      ;declare arg2 as ^val in routine
proc
    arg1=arg1 + 20          ;we add 20 to both arg1 and arg2
    arg2=arg2 + 20
    xreturn
endsubroutine
```

The output from this program is

```
^ref is 30 ^val is 10
```

Defining Data

Where Data Types Can Be Used

The value of the ^REF argument is updated in the calling routine, whereas the value of the ^VAL argument is updated only locally (i.e., within the subroutine).

Return values

To determine which Synergy data types can be used as return values, see the table on [page 2-18](#). For strong prototyping, it is best to specify the return type of all functions. Methods require a return value; VOID is a special return type that you can use to indicate that a method does not return a value.

In Synergy standard, a return value cannot be a pseudo or real array; for methods, return values may be dynamic arrays of objects. ^VAL indicates that the return type is a native int passed by value. On 32-bit platforms, return types on functions cannot be bigger than **i4**.

Using Data

Where fields can be defined

Fields can be defined in classes and in a data division or DATA statement for subroutines, functions, methods, or the main routine. For example, you can define fields as follows:

```
class
    ab                ,a10
subroutine
    record
        ac            ,d2
    proc
        begin
            data ad    ,i4
        end
    endsubroutine
endclass
```

Data organization

Data can be located in local records, common records, literals, global data sections, and in the procedure division.

Defining a record

Records are areas of computer memory in which dynamic data is stored during program execution. All Synergy Language data is stored in records. Each record consists of one or more fields, which may or may not be part of a group.

The record itself may or may not be part of a class. If it is part of a class, it is called a class record.

Each routine can have local record areas that it alone can access. In addition to local areas, a routine can define or access shareable data areas using common records or global data sections.

A record's fields determine the internal layout of the data area associated with that record. Data is stored sequentially in the order that the fields are declared in your data division. The total size of a record is the sum of the sizes of the data areas allocated to all of its fields.

A record can be named or unnamed. If you name a record, you can use the name to refer to the entire data space allocated to that record. For example, you probably use record names in Synergy Language I/O statements to read records from data files. If you refer to a record in your procedure division, Synergy Language treats the record as an alpha field. A named record cannot be larger than 65,535 bytes on a 32-bit machine.

With named records, the record definition often reflects the structure of records in the data file. The contents of the record area are processed, and the information is transmitted to or received from the data file.

Most programs, however, also require working storage for values that are not associated with data file I/O activity. Some examples of working storage variables are as follows:

- ▶ Loop-control variables
- ▶ Internal processing flags
- ▶ Variables to hold transitional results

Here's where unnamed records become useful: usually you'll want to declare this type of variable as part of one or more unnamed work records. The record definition allocates a section of data space to hold the working variables required by your program.

Sometimes you'll want some variables in a subroutine to maintain their contents between calls to the subroutine. Defining the variables in a static record does the trick. In a static record, the data areas are permanent and are therefore maintained between subroutine calls.

To allocate read-only data space, you can also define literal data space that is either local, global, or external (allocated in a previous routine).

Local records

Use the **RECORD** statement to define local records. (When you're defining a local record, the **RECORD** statement must *not* be part of a **GLOBAL-ENDGLOBAL** statement block.) The local record and all of its fields and groups are called local variables.

Only the routine that defines a local variable can directly access that variable; a local variable cannot be referenced directly by any other routine. However, local data *can* be referenced indirectly as an argument in a subroutine, provided the calling program specifies the name of a local variable in the **XCALL** statement that calls the subroutine.

See [RECORD-ENDRECORD](#) on page 4-156 for syntax.

A record's size is the total size of its fields.

Defining a common record

Use the **COMMON** statement to define records that are accessible to other subroutines or the main routine. **COMMON** enables routines to share common records and variables without having to pass the data as arguments. Using common data to pass information between a main routine and its subroutines is often faster than using subroutine arguments.

Common data remains resident in memory for as long as the program is running.

The **COMMON** statement has two forms: **GLOBAL COMMON** and **EXTERNAL COMMON**. The **GLOBAL COMMON** form creates new data space, while the **EXTERNAL COMMON** form references data that must be defined elsewhere. You can use the **GLOBAL COMMON** statement to

define a shared data record and its component fields in either the main routine or a subroutine. (In previous versions of Synergy Language, the `COMMON` statement could only define data in the main routine, because there were no `EXTERNAL` or `GLOBAL` qualifiers.)

Using `EXTERNAL COMMON` statements establishes references to data areas in the defining routine. Since the physical layout of a common record is defined by the `GLOBAL COMMON` declaration, the layout is not known while the referring routines are being compiled. Because Synergy Language cannot check the layouts during compilation of `EXTERNAL COMMON` statements, the compiler creates a symbolic reference to the named common variable. Linking (the process that combines the main routine and its subroutines into an executable program) determines the correct data address for each symbolic reference. Each external common variable references data starting at the same point as the corresponding global common variable.

Only the starting point of each external common variable is fixed while a program is being linked. The compiler can't check the sizes or types of the individual fields for the same reason it can't check the overall field layout. For example, subroutine A can declare a field named **cust** to be 15 characters long, while subroutine B can declare the same field to be only 8 characters long. In this situation, only the first 8 characters of subroutine A's field are referenced. If subroutine B declares **cust** to be 23 characters long, the entire **cust** field of subroutine A, as well as the 8 characters that follow it, are referenced.

Synergy Language stores data declared as common in a separate space from data that is local to a routine. Therefore, the data space of a record that follows a common is not contiguous with the data space of the common.

See [COMMON-ENDCOMMON on page 4-30](#) for syntax.

Defining a literal

The `LITERAL` statement defines read-only data (which is placed in a shareable read-only segment on OpenVMS). Literal data is only emitted by the compiler if it is a global literal or if any of the literal fields is accessed in the compilation unit. If unnamed literal sections are encountered and do not have overlays or use the `@position` field qualifier, only those literals that are used are emitted, and all integer literals are aligned on native boundaries.

Defining a global data section

Global data sections are shared data areas that are defined by the name of the area rather than the record or field name. Use the `RECORD` statement within a `GLOBAL-ENDGLOBAL` statement block to define shared data in a global data section. Each `GLOBAL` statement names a data area that is independent of and accessible to any routine; you can define a global data section in one place and then reference it anywhere. Synergy Language allocates global data space to global data sections in the order in which the records are declared in the data division.



Objects cannot be defined in a global data section.

Here's an example of a global data declaration:

```
global data section glb
  record alex
    fld1a   ,a6
    fld2a   ,d3.2
  record mary
    fld1m   ,d2
    fld2m   ,d7
endglobal
```

The example above defines a reference to a global data section named **glb**. The routine declares the first 18 characters of the global area and divides them into two consecutive global records, **alex** and **mary**, of 9 characters each.

The largest uniquely named global data section definition determines the size of that global area. For example, if global data section **glb** is also referenced in another routine as being 193 characters long, **glb** is established as 193 characters long when the program is executed. The example definition above references only the first 18 characters of the global area.



Be careful when changing the size of a global data section: If an ELB increases the defined size of a global data section that's owned externally, and no programs are relinked against the ELB, executing those programs could cause memory access violations. If you change the size of a global data section, all programs that use it must be relinked.

Because the Synergy compiler has no way of knowing whether other routines are referring to a given global area, you must make sure that all of the necessary routines assign the same name to that global area. When you reference the same data in another routine, the names of the global data sections must be the same.

Global variables serve an important function when you are developing libraries of interrelated subroutines or when you need a scratch data area. Like common variables, when you use global variables, your subroutines can share different data sets without affecting the routines that called them. Because the calling routines are not concerned with the content and layout of global areas, you don't have to modify your routines when you change these areas.

The main difference between common data and global data is in the way the data is accessed. While common variables are accessed relative to the name of each common variable, global data section variables are accessed relative to the name of the global data section. Think of global data sections as overlaid record groups, with each routine defining its own layout of a given global data section and the size of each named section being the size of the largest definition.

Synergy Language stores data of different types in separate spaces. For example, all static record data is gathered together, even if other types of data types are declared between static record data. Therefore, the data space of a record that follows a global data section is not contiguous with the data space of the global data section.



Do not depend on the relative placement of one global variable to another.

See [GLOBAL-ENDGLOBAL on page 4-80](#) and [RECORD-ENDRECORD on page 4-156](#) for syntax.

Defining data in the procedure division

The DATA statement enables you to define data in the procedure division. See [DATA on page 4-33](#) for more information.

The scope of the data is within the BEGIN-END block in which it is defined. Data variables are not initialized unless they are objects or have an initial value specified. (In other words, they operate the same as stack records.)

Data layout

Defining a group

A group is a subrecord within a record, structure, or class or within the layout of an argument passed as an alpha field. Fields or other group declarations can be members of a group. Groups are defined by the GROUP statement, and they enable you to maintain record arrays easily. See [GROUP-ENDGROUP on page 4-86](#) for more information.

A field or group inside a common record defines a global definition or reference. However, a field inside a group within a common record is always local. For example, let's assume you have the following declaration:

```
global common pet      ,a           ;Global reference
    dog                ,a5          ;Global reference
    cat                ,a3          ;Global reference
    group breed        ,a           ;Global reference
        long_hair      ,a4          ;Local reference
        short_hair     ,a4          ;Local reference
        group color    ,a           ;Local reference
            white      ,a3          ;Local reference
            calico      ,a3          ;Local reference
        endgroup
    endgroup
```

Pet, **dog**, **cat**, and **breed** are all global data, but **long_hair**, **short_hair**, **color**, **white**, and **calico** are local. This means that an EXTERNAL COMMON reference to **pet** in another routine is valid, but a reference to **white** is not. (A linker error will be generated.)

Global definitions and references are only made on fields and on groups at the outermost level. (In the example above, **breed** is the group at the outermost level.) All references to or definitions of fields and subgroups within a group are relative to the beginning of that group. See [“Variable path specifications” on page 1-24](#) for more information about referencing a field within a group.

Defining a structure

There are three kinds of structures: local routine structures, global structures (defined outside a routine or class), and class structures. All structures are defined by the **STRUCTURE** statement. Fields and groups can be members of a structure. See [STRUCTURE-ENDSTRUCTURE on page 4-179](#) for more information.

Using a structure with a structfield

A structfield is a named reference to a structure and an allocation of memory onto which the structure is mapped. You can use a structfield as a record wherever a Synergy record can be used (in other words, with I/O routines).

When a structure is used through the structfield syntax, all fields in the structfield are initialized according to the following precedence rules:

1. Use the initial field value if it has been specified.
2. Use the default value based on the field type.

To access a structfield, prefix the field name with the locally declared variable name (**var1** in the example below) that represents the structure. For example,

```
var1.field1 = 5  
x = var1.field1
```

See [“structure” on page 2-13](#) for an example of how to declare a field as a structure and use that structure.

Overlaying data

Sometimes you must redefine the format of a record area. For example, if you have a data file whose records are all the same size but have different layouts, as determined by a record-type variable in each record, you will probably want your program to use the same area of internal storage repeatedly during different stages of execution.

In Synergy Language, records and fields can overlay one another, thus giving your data several different layouts without defining any new data. When you define an alternate layout for a data area, you must include an “overlay indicator” immediately after the name of the alternate record layout in your record definition. The overlay indicator looks like this:

```
, X
```


If the alternate layout is not named, place the overlay indicator where you would otherwise specify the record name. The presence of the overlay indicator causes the field definitions that follow it to reference the data area in the most recently defined nonoverlay record.

The following rules apply to overlay records:

- ▶ Initial values are not allowed in overlay records.
- ▶ A static record can only overlay another static record.
- ▶ A stack record can only overlay another stack record.
- ▶ Because an external common record only establishes a memory reference, it cannot overlay other types of records. Any overlay indicators are ignored.
- ▶ If the record is overlaying an external common record, the overlaying record begins at the data area location of the overlaid record's name in the routine in which it is declared as global common. The overlaying record *can* be longer than the record being overlaid.
- ▶ If the record is overlaying another record defined by either the RECORD or the GLOBAL COMMON statement, the overlaying record *cannot* be longer than the record being overlaid.
- ▶ A local record can only overlay a common record if the common is named. All fields within the overlay record are referenced as an offset from the name of the external common.
- ▶ A global common record can only overlay another global common record.
- ▶ A literal record can only overlay another literal record if the record being overlaid is named.
- ▶ An external literal record cannot overlay anything.
- ▶ A global literal record can only overlay another global literal record.
- ▶ A local literal record cannot overlay a global or external literal record.
- ▶ An object variable cannot overlay a nonlocal path, such as a namespace or class.
- ▶ A class field that is not in a record cannot overlay another field.

When a record or group contains object fields, the record or group cannot be overlaid with **x**, and neither can the fields using the **@position** syntax to overlay the object fields. A named record, group, or common that contains object fields cannot be used in any statement (including the result of an assignment statement) that modifies the record or group. To clear such records or groups, use the INIT statement instead.

Here are some examples of overlaid and nonoverlaid records:

```
record info                                ;Named nonoverlaid record
    time      ,d6
    date      ,d6
record ,x                                    ;Unnamed overlay record
    hr        ,d2
    min       ,d2
    sec       ,d2
```

Defining Data

Using Data

```
record ,x                                ;Unnamed overlay record
    current    ,d12

global common cmn1                        ;Named global common
    c_id       ,a4
    c_code     ,d2
    c_name     ,a20
global common ,x                          ;Named global common overlay
    c_key      ,a6
```

Variable **hr** refers to the first two characters of variable **time**, variable **min** refers to the next two characters of variable **time**, and variable **sec** refers to the last two characters of variable **time**. Variable **current** refers to the entire data area allocated to **info**. Variable **c_key** refers to the first two fields defined in **cmn1**: **c_id** and **c_code**.

Lifetime of Data

Scope

Using our example from [page 2-39](#), the comments below specify the scope of each field.

```
namespace ns1
class class1
    ab          ,a10      ;Valid within the class
    subroutine sub1
    record
        ac      ,d2      ;Valid within the subroutine
    proc
        begin
            data ad ,i4      ;Valid within the BEGIN-END block
        end
        xreturn
    endsubroutine
endclass
endnamespace
```

Access to a structure and all of its members declared within a routine are limited to the scope of the routine in which it was declared.

Characteristics

Constant data

Constant data is not modified at all during program execution. It is defined with the **LITERAL** statement or by the class **READONLY** or **CONST** field modifiers.

Local data

Local data is specific to a routine. It can be defined within a **RECORD** statement that is not part of a **GLOBAL-ENDGLOBAL** block or in a subroutine or function that is declared outside of a class. The **LOCAL** and **STATIC** qualifiers on the **RECORD**, **FUNCTION**, and **SUBROUTINE** statements can also designate data as local. See the Discussion for [RECORD-ENDRECORD on page 4-156](#) for a more thorough discussion of **LOCAL** and **STATIC**.

The lifetime of local data is determined by the lifetime of the routine, and the data goes out of scope when the routine ends.

Instance data

Instance data is tied to an object variable. When you instantiate an object, it creates data and assigns a reference of that newly created object to an instance variable. Once there are no more references, either because the variable was explicitly cleared or because it went out of scope, the instance is destroyed and the data goes away.

An instance variable is specific to the block in which it is defined. The DATA statement is one way to define instance variables. The lifetime of an instance variable is determined by the block in which the object variable is declared. When the block ends, the variable goes out of scope.



All class instance data operates the same as if the **-qcheck** compiler option was specified. Class instance data cannot be subscripted, nor can arrays be accessed beyond their bounds for any class instance data. This is also true for local DATA statement variables.

Global data

Global data remains resident in memory for as long as the program is running. It is defined in a global data section, in a COMMON statement, or as static class data (in other words, as a field within a class that has the STATIC qualifier). Static class data is shared by all instances of the class.

Memory management

Memory can be managed either statically or dynamically. Static memory is persistent throughout the life of the program. A record's data is initialized when the program is started, and it remains constant until the entire program is exited.

Dynamic memory management is the allocation, usage, and deallocation of memory that is controlled at execution time, rather than compile time. The elements of the language that support dynamic memory are

- ▶ LOCAL and STACK qualifiers on the RECORD, SUBROUTINE, and FUNCTION statements.
- ▶ STRUCTURE statement.
- ▶ %MEM_PROC intrinsic function.
- ▶ ^M data reference operation.
- ▶ The dynamic arrays [#,#]@class and [#,#]structure.
- ▶ System.Collections.ArrayList class.

Activation-related data

Usually, a routine's data is defined using a `RECORD` statement. The data is processed by the Synergy compiler and allocated directly when the program that contains the routine is linked.

Using the `STACK` qualifier on the `RECORD` statement helps solve this problem. It declares that a record's data is created anew on each activation of the routine and "thrown away" on the associated deactivation.

Memory access using "handles" (^M)

Synergy Language supports memory access mechanisms using the "handle" concept. A memory handle is an integer identifier that the Synergy runtime associates with a given memory segment. It is to be used whenever that memory segment is referenced.

A memory handle is either volatile (dynamic) or static. Volatile memory handles remain valid until one of the following is true:

- ▶ The entire program is exited.
- ▶ The activation level of the routine in which the memory handle was generated is exited.
- ▶ The memory handle is explicitly released.

Static memory handles are "persistent" and remain valid until one of the following is true:

- ▶ The entire program is exited.
- ▶ The memory handle is explicitly released.

A memory handle is essentially an index into an access array that is maintained by the runtime. A positive value is a one-based index into the static handle array, while a negative value is a negative one-based index into the volatile handle array. Any given memory handle can be stored in whatever numeric Synergy Language field can represent that handle's value (although we strongly recommend an aligned `i4` field).

Note that we refer to operations in terms of *memory handles*, not *memory* (for example, "generate a memory handle," "release a memory handle," and so forth). We do so because, even though all memory handles have a memory segment associated with them, different operations can occur for a given memory handle function depending on the class of its associated memory. For example, releasing a memory handle associated with a Synergy Language window's user data set does not "free" its user data set's memory.

3

Expressions

What Is an Expression? 3-2

Defines what we mean by the term “expression” and explains how one is evaluated.

Operands 3-3

Describes the kinds of operands you can use in an expression.

Operators 3-4

Describes the alpha and numeric operators you can use in an expression and discusses the operators that can be used on a class. This section also includes a table that specifies the order in which operators are evaluated.

Assignment Statements 3-26

Describes what assignment statements are and discusses the rules for moving, formatting, and changing data between destinations and sources of the same or different data types.

Compile-Time Expressions and Optimizations 3-41

Explains how Synergy Language evaluates and optimizes expressions at compile time.

What Is an Expression?

An expression is usually a combination of data elements (called operands) and operators that produces a resulting value. The result of an expression is considered to be either alpha, numeric, or object.

The simplest form of an expression is a single alpha or numeric variable, literal, object handle, or function/method reference without any operators.



A method used in a simple expression must return a type other than VOID; otherwise, a “No operator void” error (\$ERR_NOOPER) will result.

You can build more complex expressions by combining operators with operands.

Truth values

The truth value of an expression is based on its result as follows:

- ▶ The truth value of an alpha expression is false if its length is zero (“”) or contains only ASCII space characters, and true in all other cases (including if it only contains ASCII zero characters).
- ▶ The truth value of a numeric expression is false if it is zero, and true if it is nonzero.
- ▶ The truth value of an object result is the truth value of the result returned from the `op_True` method declared by the class of the object. If the `op_True` method is not declared by the class, a “No operator `op_True()`” error (\$ERR_NOOPER).



A `System.String` object is always converted to alpha before its truth value is evaluated.

Operands

Operands provide the basic data that will be combined to give an expression its value. You can use the following types of operands in an expression:

- ▶ Literals (for example, “abc”, 123, \$ERR_FNF)
- ▶ Variables, in simple, subscripted, ranged, and path forms (for example, avar, dvar[1,2], avar(1:2), avar.fld)
- ▶ Function references (for example, %afunc)
- ▶ Data reference operations (for example, ^a(dvar), ^size(arg))
- ▶ Expressions (for example, “abc” + “def”, a.eq.b)
- ▶ Value assignments (for example, dvar = 3)

Operators

Operators control the evaluation order of an expression and determine how the operands are combined to yield a final result. Each operator has a precedence level within the entire group of available operators. When an expression is evaluated, the operators act in order of their precedence.

An operator is either unary or binary. (See the table on [page 3-20](#) for a list of operators and their unary/binary type.) Unary operators act on a single operand and must appear on the left side of the expression. Binary operators act on two operands and are placed between the operands.

Synergy Language has six types of operations that occur outside of a class:

- ▶ Assignment
- ▶ Alpha (only)
- ▶ Numeric (only)
- ▶ Relational
- ▶ Boolean
- ▶ Bitwise

See [“Object operators” on page 3-14](#) for information about operators that can be used on a class.

Assignment operations

The assignment operation is equivalent to a single-line assignment statement, which is described in detail in [“Assignment Statements” on page 3-26](#). The equal sign is the most common assignment operator. Additional assignment operators are specific to numeric and bitwise operations. (See [“Additional assignment operators” on pages 3-9 and 3-14](#).)

Assignment operator

We refer to the equal sign as an assignment operator, because it assigns one operand to another. When you use the equal sign as an operator, the operand on the left must be a variable specification, and the operand on the right is treated as an expression and stored in the variable before the variable is used in the rest of the expression.

For example:

```
xcall sub(a=1, b=2, c="ABC")
```

is equivalent to the following:

```
a=1  
b=2  
c="ABC"  
xcall sub(a, b, c)
```

In the following example, Y is set to 3 before it is added to X:

```
X + Y = 3
```

This characteristic enables you to create operations like the following:

```
if ((len=^size(arg)).eq.4)
```



The assignment operator causes parts of an expression to be evaluated from right to left instead of left to right. See [page 3-22](#) for more information.

Alpha operations

Conversion to and from System.String

Alpha operators allow both alpha and System.String operands in the same expression. However, prior to calling the appropriate operator, one will implicitly be converted to the other according to the following rules:

- ▶ Relational operators (.EQ., .NE., etc.) convert System.String operands to alpha.
- ▶ String relational operators (.EQS., .NES., etc.) convert alpha operands to System.String when one of the operands is a System.String.
- ▶ When using symbolic string relational operators (==, !=, etc.) with System.String operands, see [page 3-16](#).
- ▶ The concatenation operator (+) converts an alpha operand on the right to System.String if the operand on the left is a System.String. If the operand on the left is alpha and the operand on the right is System.String, the System.String is converted to alpha.

When an alpha operand is converted to System.String, it is not blank trimmed.

Concatenation and reduction

In alpha expressions, the addition (+) and subtraction (–) operators perform string operations on alpha operands. The addition operator performs string concatenation, in which the operand on the right is appended to the operand on the left. The subtraction operator performs string reduction, in which the first occurrence of the operand on the right is removed from the operand on the left. If the operand on the right does *not* occur in the operand on the left, the result is the content of the operand on the left.

For example, the following expression yields a result of “abcdef”:

```
"abc" + "def"
```

The expression below yields a result of “xydefabc”:

```
"xyabcdefabc" - "abc"
```

For concatenation using System.String, see the Concat method in System.String Class.

String relational operators

String relational operations compare two operands. Either operand can be an alpha or System.String, and they are compared slightly differently than relational operators. The string relational operators available in Synergy Language are as follows:

.EQS.	Equal to
.NES.	Not equal to
.GTS.	Greater than
.LTS.	Less than
.GES.	Greater than or equal to
.LES.	Less than or equal to

Alpha operands are compared according to the order of characters within the ASCII character set. The complete ASCII character set is listed in [Appendix B](#). If the alpha operands in an expression are the same size, the characters in corresponding character positions are compared. If the alpha operands differ in size, the shorter operand is logically extended with blanks on the right until it is the same size as the larger operand. For example, comparing “A” to “ABC” is logically equivalent to comparing “A ” to “ABC”.

The result of a string relational operation is either true or false. If the relation is true, the result is an integer value of 1. If the relation is false, the result is an integer value of 0.

- ▶ The result of the .EQS. operation is true if the operand on the left is equal to the operand on the right.
- ▶ The result of the .NES. operation is true if the operand on the left is *not* equal to the operand on the right.
- ▶ The result of the .GTS. operation is true if the operand on the left is greater than the operand on the right.
- ▶ The result of the .LTS. operation is true if the operand on the left is less than the operand on the right.
- ▶ The result of the .GES. operation is true if the operand on the left is greater than or equal to the operand on the right.
- ▶ The result of the .LES. operation is true if the operand on the left is less than or equal to the operand on the right.

Here are some examples:

Expression	Result
“ABCDEF” .eqs. “ABCDEF”	true
“ABCDEF” .eqs. “ABC”	false

The following expression yields a result of 5:

```
5 + 11 * ("AB" .eqs. "ABCD")
```

First, the statement that “AB” is equal to “ABCD” is evaluated to be false. Because false has a value of 0, the equation becomes $5 + 11 * 0$, which equals 5.

If either operand is System.String, the corresponding System.String operator is called. Strings are equal if the length of both strings are equal as well as the contents of the both strings (according to the order of characters in the ASCII character set). A System.String object that is equal to ^NULL is less than a System.String object that is equal to “”.

Numeric operations

A numeric data hierarchy exists that consists of the following data types, listed from highest to lowest:

- ▶ Implied-packed
- ▶ Packed
- ▶ Implied-decimal
- ▶ Decimal
- ▶ Integer

If an operator has operands with different numeric types, the operand that is lower in the hierarchy is promoted to the data type of the higher one. For example, if an integer value and an implied-decimal value are operands in an expression, the integer is promoted to implied-decimal before the operation occurs, and the result is implied-decimal.



For performance reasons, Synergy uses the platform's native integer size for expression intermediates. On 32-bit platforms, if an **i8** is explicitly used in an integer expression, an **i8** intermediate is used. Otherwise, an integer expression not using **i8**s that results in a value that would exceed the native **i4** will wrap (not be promoted to an **i8**) and no error will be generated. If necessary, you can avoid this by using **i8**s instead, but at a slight cost to integer performance. On 64-bit platforms, all integer expressions use native **i8** intermediates.

Unary plus and minus operators

Unary plus and minus operators determine whether the numeric operand is positive (+) or negative (–). Synergy Language ignores the unary plus operator, because it assumes unsigned values are positive. The unary minus operator changes the sign of the operand to its right. Successive minuses are combined algebraically. For example, two minuses become a plus, three minuses become one minus, and so forth.

The data type of the operation's result is the data type of the operand.

Arithmetic operators

Arithmetic operators perform addition (+), subtraction (–), multiplication (*), or division (/ and //).

If neither operand contains an implied decimal point, the addition, subtraction, multiplication, and division operators perform standard, signed, whole number arithmetic. If either operand contains an implied decimal point, the intermediate will contain a fractional portion.

If you add a 1 to a **d28** variable that contains all 9s, the number rolls over to zero.

If a division operation using “/” involves two whole numbers (decimal, integer, or packed), any fractional part of the result is truncated without rounding. For example, the following expression yields a result of 2, not 2.5 or 3:

```
5/2
```

Division by zero is illegal. If you try to divide by zero, you’ll get an “Attempt to divide by zero” error (\$ERR_DIVIDE).

When you multiply or divide implied-decimal or implied-packed operands, the intermediate result of the operation is calculated to a fractional precision of 29 digits. It is then truncated or rounded to 28 digits. For example, the expression below yields a result of 0.666666666666666666666666666666 if truncation is the default or 0.666666666666666666666666666667 if rounding is the default:

```
2.0/3.0
```

(Rounding is the default unless you set system option #11 or specify the TRUNCATE option in the SUBROUTINE statement, which indicates the start of the routine. See [#11](#) in the “System Options” chapter of *Environment Variables and System Options* and [TRUNCATE](#) on [page 4-182](#) in this manual for more information.)

Using the “//” division operator always results in an intermediate result that contains a fractional portion.

Synergy Language checks a numeric data field for invalid characters or length when that field is used in an arithmetic expression. If you use a decimal field that contains more than 28 digits, a packed field that contains more than 18 digits, an implied-decimal field that has a whole number part or fractional precision of more than 28 digits, or an implied-packed field that has a whole number part of more than 18 digits or a fractional precision of more than 10 digits, an “Arithmetic operand exceeds maximum size” (\$ERR_BIGNUM) error is generated. If any nonnumeric characters other than a blank, a decimal point, or a sign (+ or –) are present in the operands, a “Bad digit encountered” error (\$ERR_DIGIT) is generated.

Additional assignment operators

Additional assignment operators for numeric operands include +=, -=, *=, and /=. These operators are more efficient than their equivalent expressions (shown in the following table).

Operation	Equivalent expression
A += B	A = A + B
A -= B	A = A – B
A *= B	A = A * B
A /= B	A = A / B

Rounding operators

There are two kinds of rounding operators: # and ##.

The # rounding operator is a binary operator that rounds the left operand (the value being rounded) by the number of digits specified by the right operand (the round value). For example, if the right operand has a value of 3, the rounding operation discards the rightmost three digits of the left operand and then adds 1 to the result if the leftmost discarded digit is greater than or equal to 5.

The following rules apply to rounding with the # operator:

- ▶ The round value must be in the range 1 through 28. (Anything outside of this range generates an “Invalid round value” error [\$ERR_RNDVAL].)
- ▶ Both operands must be decimal or integer.
- ▶ The data type of the result is the same as the operand.
- ▶ The sign of the result is the sign of the value being rounded.
- ▶ If the number of significant digits in the value being rounded is less than the round value, the result is zero.

Here are some examples:

Expression	Result
345671 # -1	compiler error
345671 # 0	345671
345678 # 3	346
345678 # 4	35

Expression	Result
-345678 # 4	-35
345678 # 7	0
6789.456 # 1	compiler error

The ## rounding operator specifies true rounding (rounding without discarding any digits in the value to be rounded). It rounds the left operand (the value being rounded) by the number of digits specified by the right operand (the round value).

The following rules apply to rounding with the ## operator:

- ▶ The round value must be in the range -28 through 28.
- ▶ If the round value is positive, rounding begins “round value” places to the left of the decimal point. For example, if the round value is 3, rounding begins three places to the left of the decimal point.
- ▶ If the round value is negative, rounding begins “round value + 1” places to the right of the decimal point. For example, if the round value is -1, rounding begins two places to the right of the decimal point.
- ▶ Both operands must be decimal or integer.
- ▶ The data type of the result is decimal.
- ▶ The sign of the result is the sign of the value being rounded.
- ▶ If the number of significant digits in the value being rounded is less than the round value, the result is zero.

Here are some examples:

Expression	Result
123.456 ## -2	123.46
12345 ## 2	12300
345671 ## -1	345671.0
345671 ## 0	345671
345678 ## 3	346000
345678 ## 4	350000
-345678 ## 4	-350000

Relational operations

Relational operators

Relational operators (and their corresponding symbolic relational operators) compare two operands. The relational operators available in Synergy Language are as follows:

.EQ. (or ==)	Equal to
.NE. (or !=)	Not equal to
.GT. (or >)	Greater than
.LT. (or <)	Less than
.GE. (or >=)	Greater than or equal to
.LE. (or <=)	Less than or equal to

When you use a relational operator, both operands must be either alpha or System.String, or both must be numeric. When using System.String operands, see [“Conversion to and from System.String” on page 3-5](#).



See [page 3-16](#) for a list of string operators and their equivalents.

The result of a relational operation is either true or false. If the relation is true, the result is an integer value of 1. If the relation is false, the result is an integer value of 0.

- ▶ The result of the .EQ. (or ==) operation is true if the operand on the left is equal to the operand on the right.
- ▶ The result of the .NE. (or !=) operation is true if the operand on the left is *not* equal to the operand on the right.
- ▶ The result of the .GT. (or >) operation is true if the operand on the left is greater than the operand on the right.
- ▶ The result of the .LT. (or <) operation is true if the operand on the left is less than the operand on the right.
- ▶ The result of the .GE. (or >=) operation is true if the operand on the left is greater than or equal to the operand on the right.
- ▶ The result of the .LE. (or <=) operation is true if the operand on the left is less than or equal to the operand on the right.

Numeric operands result in standard, signed arithmetic comparisons.

Alpha operands are compared according to the order of characters within the ASCII character set for the length of the shortest operand. See [Appendix B](#) for the complete ASCII character set. If the alpha operands in an expression differ in size, only the number of characters in the shortest operand are compared.

Here are some examples:

Expression	Result
"ABCDEF" .eq. "ABC"	true
"ABCDEF" .eq. "ABD"	false

The following expression yields a result of 20:

`17 + 3 * ( "ABCD" .eq. "AB" )`

First, the statement that "ABCD" is equal to "AB" is evaluated to be true. Because true has a value of 1, the equation becomes `17 + 3 * 1`, which equals 20.

Unsigned integer operators

Synergy Language also supports the following unsigned integer operators:

- .EQU. Unsigned integer equal to
- .NEU. Unsigned integer not equal to
- .GTU. Unsigned integer greater than
- .LTU. Unsigned integer less than
- .GEU. Unsigned integer greater than or equal to
- .LEU. Unsigned integer less than or equal to

Boolean operations

Boolean operators

Boolean operators compare the truth value of operands. The Boolean operators are as follows:

- .OR. (or ||) Or
- .XOR. Exclusive or
- .AND. (or &&) And
- .NOT. (or !) Not

The result of a Boolean operation is either true or false. If the operation is true, the result is an integer value of 1. If the operation is false, the result is an integer value of 0.

- ▶ The result of the .OR. (or ||) operation is true if either operand has a truth value of true.
- ▶ The result of the .XOR. operation is true if only one of the operands has a truth value of true.
- ▶ The result of the .AND. (or &&) operation is true if both operands have a truth value of true.
- ▶ The result of the .NOT. (or !) operation is true if the operand on the right has a truth value of false.

Boolean operators are evaluated from left to right. If the result of an `.AND.` or `.OR.` Boolean operation can be determined by the evaluation of the operand on the left, Synergy Language won't process the operand on the right. For example, in the following Boolean expression:

```
ndx .and. array(ndx)
```

the right operand, `array(ndx)`, won't be evaluated if the left operand, `ndx`, has a false truth value, because if `ndx` is false, the entire expression is false.

Bitwise operations

Bitwise operators

Bitwise operators perform Boolean operations on the bits within integer operands. (A bitwise operator can also be used within an enumeration.) If an operand is decimal, it is converted to integer before the operation. The bitwise operators are as follows:

<code>.BOR.</code>	(or <code> </code>)	Bitwise or
<code>.BXOR.</code>		Bitwise exclusive or
<code>.BAND.</code>	(or <code>&</code>)	Bitwise and
<code>.BNAND.</code>		Bitwise not and
<code>.BNOT.</code>	(or <code>~</code>)	Bitwise not

If the size of the operands are not the same, the length of the smaller operand is logically extended to the length of the larger by adding null (0) high-order bytes. The size of the result is the size of the largest operand.

- ▶ Each result bit in a `.BOR.` (or `|`) operation is one if either of the corresponding bits of the operands is one, and zero if both bits are zero.
- ▶ Each result bit in a `.BXOR.` operation is one if only one of the corresponding bits of the operands is one, and zero if both bits are the same value.
- ▶ Each result bit in a `.BAND.` (or `&`) operation is one if both of the corresponding bits of the operands are one, and zero if either bit is zero.
- ▶ Each result bit in a `.BNAND.` operation is one if either of the corresponding bits of the operands is zero, and zero if both bits are one.
- ▶ Each result bit in a `.BNOT.` (or `~`) operation is one if the corresponding bit of the operand is zero, and zero if it is one.

For example, in the bitwise expression:

```
i2var .band. i1var
```

where the variables are declared as follows:

```
i2var      i2,      256
i1var      i1,      -1
```

ilvar is logically extended to an **i2** field by adding a null high-order byte. The result of this operation is an **i2** value of zero.

Additional assignment operators

Additional assignment operators for integer operands are **|=** and **&=**. Equivalent expressions are shown in the following table:

Operation	Equivalent expression
A = B	A = (A .BOR. B)
A &= B	A = (A .BAND. B)

These additional assignment operators can make bit manipulation much clearer. For example, the following statement sets all of the bits within A that are also set within B:

A |= B

The statement below sets A to all bits set in A and B:

A &= B

Object operators

The operators in this section can be implemented within a class so they can be used on instances of the class.

To find out if the runtime class of an instance variable is from a particular class, use the **.IS.** operator. If the class provided is in the class hierarchy of the instantiated class of the instance variable, the **.IS.** operator returns true. If not, it returns ^NULL. For example,

```
MyOther myobj = new MyClass
if (myobj .is. MyClass)
```

Unary class operators

Synergy Language supports the following unary operator methods for use on a class:

Unary operator method name	Unary operator	Example syntax
op_Increment	incr	incr obj
op_Decrement	decr	decr obj
op_UnaryNegation	-	-obj

Unary operator method name	Unary operator	Example syntax
op_UnaryPlus	+	+obj
op_LogicalNot	! (.not.)	!obj
op_True	truth condition	if (myclass)

A unary operator method is invoked with the appropriate syntax from the table above. For example, the following invokes the op_Increment() operator:

```
c1, @myclass
proc
    incr c1
end
```

A unary operator method must have one argument, and it must be the enclosing class type. The method must be declared as PUBLIC and STATIC, and it cannot return a VOID.

The unary operator method in the example below increments a **myclass** object:

```
public static method op_Increment, @myclass
    p1, @myclass
proc
    p1.m_value = p1.m_value + 1    ;m_value is an i4 field in myclass
    return p1
end
```

Binary class operators

Synergy Language supports the following binary operator methods for use on a class:

Binary operator method name	Binary operator	Example syntax
op_Addition	+	obj + obj2
op_Subtraction	-	obj - obj2
op_Multiply	*	obj * obj2
op_Division	/	obj / obj2
op_ExclusiveOr	.xor.	obj .xor. obj2
op_BitwiseAnd	& (.band.)	obj & obj2
op_BitwiseOr	(.bor.)	obj obj2
op_LogicalAnd	&& (.and.)	if (obj && obj2)

Expressions

Operators

Binary operator method name	Binary operator	Example syntax
op_LogicalOr	(.or.)	if (obj obj2)
op_OnesComplement	~	if (~obj)
op_Equality	== (.eq.)	if (obj == obj2)
op_GreaterThan	> (.gt.)	if (obj > obj2)
op_LessThan	< (.lt.)	if (obj < obj2)
op_Inequality	!= (.ne.)	if (obj != obj2)
op_GreaterThanOrEqual	>= (.ge.)	if (obj >= obj2)
op_LessThanOrEqual	<= (.le.)	if (obj <= obj2)

A binary operator method is invoked with the appropriate syntax from the table above.

A binary operator method must have two arguments, and at least one must be the enclosing class type. The method must be declared as **PUBLIC** and **STATIC**, and it cannot return a **VOID**. You can declare multiple binary operator methods of the same name within a class as long as their method signatures differ.

The binary operator method in the example below adds two **myclass** objects:

```
public static method op_Addition, @myclass
    p1, @myclass
    p2, @myclass
proc
    mreturn new myclass(p1.m_value+p2.m_value)
end
```

If either operand to a symbolic string relational operators is of type **System.String**, the operator will operate the same as its corresponding string relational operator:

Symbolic string relational operator	Equivalent Synergy string relational operator
==	.EQS.
!=	.NES.
>	.GTS.

Symbolic string relational operator	Equivalent Synergy string relational operator
<	.LTS.
>=	.GES.
<=	.LES.

If a string relational operator is used in an expression and one of the operands is `System.String`, the compiler assumes that both operands are `System.String` or can be converted to `System.String`, and the corresponding string operator is run. (See [System.String on page 9-404](#) for a list of string operators.) For `==` or `.EQS.` where an operand is `System.String`, the equality operator (`op_Equality`) is run, which first ensures the lengths are equal and then performs a character-by-character comparison.

If the nonstring relational operator is used, the compiler assume that both operands are of type `alpha` or can be implicitly converted to `alpha`.

The `op_Equality` operator for boxed types operates the same as the relational “equal to” operator for the type that’s boxed. When both operands are boxed, the boxed values are compared based on their types. In addition to the `op_Equality` operator, boxed types implement a virtual override for `Object.Equals`, which behaves like `op_Equality` for boxed types.



A boxed `alpha` type compared to a boxed numeric type always returns false.

When comparing two object references, the equality operator (`.EQ.` or `==`) returns true only if the objects reference the exact same instance of an object. Otherwise, it returns false. If you declare the `op_Equality` operator for the class, the result of the `op_Equality` method is returned.

For example, assume `ChildClass` extends `ParentClass`.

```
var1  ,@ChildClass
var2  ,@ChildClass
var3  ,@ParentClass
var4  ,@ChildClass
var1 = new ChildClass()
var2 = var1
if (var2 .eq. var1)           ;True
if (var1 .eq. var2)           ;True

var3 = (ParentClass)var1
if (var3 .eq. var1)           ;True
if (var3 .eq. var2)           ;True

var4 = new ChildClass()
```

```
if (var4 .eq. var1)           ;False
if (var4 .eq. var2)           ;False
if (var4 .eq. var3)           ;False

clear var1
clear var2
if (var1 .eq. var2)           ;True
```



In a Where expression (see [Synergex.SynergyDE.Select.Where](#) on page 9-427),

```
avar==avar
is equivalent to
avar .eqs. avar
```

The inequality operator (.NE. or !=) returns true if the objects do *not* reference the exact same instance of an object. Otherwise, it returns false. If you declare the op_Inequality operator for the class, the result of the op_Inequality method is returned.

The example below determines that **myobj** is uninstantiated (i.e., it has never been created with NEW):

```
if (myobj==^NULL)
```

Conversion operators

Synergy Language also supports the op_Explicit conversion operator method, as shown below:

Conversion operator method name	Description	Example syntax
op_Explicit	Explicit type conversion	obj1 = (class1)obj2

op_Explicit must have one argument that designates the source of a conversion and a return type to denote the destination for the conversion. Either the argument type or the return type must be the enclosing class type, but they cannot be within the same class hierarchy.

The example below converts from **i4** to **myclass**:

```
public static method op_Explicit, @myclass
    p1, i4
proc
    mreturn new myclass(p1);
end
```

The conversion operator method must be declared as PUBLIC and STATIC. You can declare multiple conversion operator methods within a class as long as their return types and argument types differ.

If you do an explicit conversion cast, the compiler performs the following steps in order.

- ▶ See if the class is in the class hierarchy. If so, cast it.
- ▶ If the class is not in the class hierarchy, and an `op_Explicit()` method whose argument is the source type and whose return type is the destination type exists in either the source or destination class, execute the `op_Explicit` method.

For example, if the above `op_Explicit()` were defined, it would be legal to invoke the explicit conversion operator using this syntax:

```
c1, @myclass
x, i4, 5
proc
    c1 = (myclass)x
end
```

`Op_Explicit` has the following restrictions:

- ▶ You cannot call `op_Explicit` explicitly. For example, the following will generate an error:
`myclass.op_Explicit(5)`
- ▶ You cannot use `op_Explicit` with both **d** and **d.**, because the compiler interprets them as being the same and generates a “Class *classname* already defines a method `op_Explicit` with the same parameter types” error.
- ▶ In Synergy standard, if you specify `op_Explicit` for **d.**, you can’t cast an object as (**d.**); you must use (**decimal**) instead.

Resolving object operators

When resolving object operators, the compiler looks for the operators only in the direct classes involved in the operation; it does not search up the hierarchy for an appropriate operator that could work. For example, using the code below, the compiler will attempt to run the `op_Explicit` operation from **class2**. If **class2** or **class3** doesn’t have that operator, the compiler will not go up the class hierarchy looking for appropriate operators, but will instead generate an “Invalid cast operation” error, because the operator cannot resolve.

```
namespace ns1
class class1
public static method op_Explicit, @class3
    parm1, @class1
proc
    .
    .
    .
end
endclass
```

```
class class2 extends class1
public static method op_Explicit, @class3
    parml, @class2
proc
    .
    .
    .
end
endclass
class class3
endclass

and

c2, @class2
c3, @class3
proc
    c2 = new class2()
    c3 = (class3)c2
```

Order of evaluation

The following table lists the numeric operators available in Synergy Language. It is organized by left-to-right precedence, with the operators that have the highest precedence at the top.

Precedence	Operator	Type	Description
11	<i>(class)</i> , ^A, ^D, ^F, ^I	Type conversion	Cast
10	<i>array[dim]</i>		Array access
	<i>function(arg)</i>		Function or method call
9	+	Unary	Positive
	–	Unary	Negative
8	# and ##	Binary	Rounding
7	*	Binary	Multiplication
	/	Binary	Division
6	//	Binary	Division implied
5	+	Binary	Addition
	–	Binary	Subtraction

Precedence	Operator	Type	Description
4	.EQ. or ==	Binary	Equal to
	.NE. or !=	Binary	Not equal to
	.GT. or >	Binary	Greater than
	.LT. or <	Binary	Less than
	.GE. or >=	Binary	Greater than or equal to
	.LE. or <=	Binary	Less than or equal to
	.EQS.	Binary	String alpha equal to
	.NES.	Binary	String alpha not equal to
	.GTS.	Binary	String alpha greater than
	.LTS.	Binary	String alpha less than
	.GES.	Binary	String alpha greater than or equal to
4	.LES.	Binary	String alpha less than or equal to
	.EQU.	Binary	Unsigned integer equal to
	.NEU.	Binary	Unsigned integer not equal to
	.GTU.	Binary	Unsigned integer greater than
	.LTU.	Binary	Unsigned integer less than
	.GEU.	Binary	Unsigned integer greater than or equal to
	.LEU.	Binary	Unsigned integer less than or equal to
3	.NOT. or !	Unary	Boolean NOT
	.BNOT. or ~	Unary	Bitwise NOT
2	.AND. or &&	Binary	Boolean AND
	.BAND. or &	Binary	Bitwise AND
	.BNAND.	Binary	Bitwise NOT AND

Precedence	Operator	Type	Description
1	.OR. or	Binary	Boolean OR
	.XOR.	Binary	Boolean exclusive OR
	.BOR. or	Binary	Bitwise OR
	.BXOR.	Binary	Bitwise exclusive OR
0	=	Binary	Assignment
	+=	Binary	Assignment
	-=	Binary	Assignment
	*=	Binary	Assignment
	/=	Binary	Assignment
	=	Binary	Assignment
	&=	Binary	Assignment

When two or more operators of the same precedence (except for the assignment operator) are adjacent, they are evaluated from left to right. For example, $A * B / C$ is evaluated by multiplying A by B and dividing the result by C .

The presence of an assignment operator ($=$) causes an expression to be evaluated from right to left, which often mixes a left-to-right precedence order with a right-to-left precedence order and makes the order of evaluation quite confusing. In a right-to-left evaluation, the right side always has precedence over the left side, and the assignment operation occurs before the left side is evaluated.

For example, the expression

$$X + Y = 3 + Z$$

is evaluated as follows:

$$X + (Y = 3 + Z)$$

The expression $3 + Z$ is evaluated first, then Y is assigned to the result of $3 + Z$, and finally $X + Y$ is evaluated.

Likewise, the expression

$$X + Y = Z * 3 / I = 15$$

is evaluated as follows:

$$X + (Y = Z * 3 / (I = 15))$$

First it sets I to 15, then it divides 3 by I, multiplies the result of 3 / I by Z, sets Y to (Z * (3 / I)), and adds Y to X, in that order.

A balanced set of parentheses can clarify or alter the expression’s order of evaluation and enable you to make your expressions as complex as necessary. Use parentheses if you need to override the precedence order of the operations in your expression. Parentheses can also help make your code easier to read, because they clearly indicate which operations are evaluated first.

Here’s an example of when you’d want to override the normal precedence order. Let’s assume you want to use the following equation with variables that have the same data type:

$$\frac{A + B}{C - D}$$

If you wrote the expression as A + B / C – D, precedence rules would dictate that the division operation be performed first. Thus, the value would be calculated as if the function looked like this:

$$A + \frac{B}{C} - D$$

The only way to get the correct order of evaluation is to override the precedence rules with parentheses and write the expression as (A + B) / (C + D).

The expressions in the most deeply nested sets of parentheses are evaluated first. For example, in the following expression the G – H operation is evaluated first because it’s in the most deeply nested set of parentheses. D + E is evaluated next, followed by (F * (G – H)):

$$A + B * (C - (D + E) / (F * (G - H)) + I)$$

As another example, let’s assume the following variable values:

$$A = 10 \quad B = 30 \quad C = 5 \quad D = 2$$

The following expressions will have the specified values:

Expression	Value
A + B / C * D	22
A + B / (C * D)	13
(A + B) / (C * D)	4

More examples of expressions

The examples in the table below assume that the data division contains the following information:

```
record
  d5      ,d5      ,12345
  d53     ,d5.3    ,12.345
  a6      ,a6
  money   ,d6      ,127654
  y       ,d3      , -326
  a       ,p1      , 4
  b       ,d2      ,10
  c       ,d2      ,20
  d       ,i1      ,5
  e       ,d5.3    ,12.300
```

Expression	Value
5/3	" 1"
5//3	"1.6666"
5//3##-4	"1.6666"
(5//3)#0	runtime error
d5#0	" 12345"
d5#1	" 1235"
d5##1	" 12350"
(5//3)##-4	"1.6667"
a+b- c	- 6
a*d	20
b/a	2
b//a	2.5
e/b	1.23
b+c/d*a	26
b+c/(d*a)	11
(b+c)/(d*a)	1

Expression	Value
$((b+c)/d)*a$	24
money#a	13
y#2	-3
y#a	0
y#1	-33
a .eq. 4	1
a .ne. 4	0
'abc' == 'def'	0
a .eq. 4 .and. b .eq. 10	1
a .and. b	1
a .and. 0	0

Assignment Statements

Assignment statements move data, format data, and change the content of specified variables. Several forms of assignment statements exist. Generally, an assignment statement has the following format:

destination = *source*

destination Either a simple, subscripted, or ranged variable whose content is to be modified.

source An expression whose result is assigned to *destination*.

Because Synergy Language evaluates any expressions contained in *source* before it alters the value of *destination*, the same variable can be present as both *destination* and part of *source*. For example, the following assignment statement multiplies the current value of **decimal_var** by 3 and then assigns the result as the new value for **decimal_var**:

```
decimal_var = decimal_var * 3
```

By default, Synergy Language rounds results when storing from an implied-decimal source expression. If you want Synergy Language to truncate results instead of rounding them in these situations, you can use the %TRUNCATE intrinsic function, specify the TRUNCATE option on MAIN, SUBROUTINE, and FUNCTION statements, or set system option #11. See [%TRUNCATE on page 9-356](#); [TRUNCATE on page 4-72, 4-105, and 4-182](#), and [#11](#) in the “System Options” chapter of *Environment Variables and System Options* for details about option #11.

This section discusses the rules for moving, formatting, and changing data between destinations and sources of the same or different data types.

Moving alpha data to an alpha destination

When the destination and source are both alpha, the result of the source is copied to the destination. If the source is shorter than the destination, its result is copied left-justified with trailing blanks into the destination. If the source is longer than the destination, only the leftmost characters are copied to the destination, and no warning or error occurs.

If you include a justification format in an alpha-to-alpha assignment, the source is treated as decimal. See [“Controlling justification” on page 3-32](#) for information about justification controls.

Below are examples of alpha-to-alpha assignments and their corresponding results:

```
record xyz
  result      ,a4
  aflld1      ,a6,  "abcdef"
  aflld2      ,a2,  "xy"
```



```

proc                                ;The results will be:
    result = afld2                  ;"xy  "
    result = afld1                  ;"abcd" (because result is truncated)
    result = afld1(4:5)             ;"defx" (because ranging spans afld1
                                    ; [starting with 4th character] and
                                    ; afld2 and result is truncated)
    result = "1234"                 ;"1234"
    stop
end

```

Moving alpha data to a numeric destination

When the destination is numeric and the source is alpha, the source is evaluated to produce a resulting numeric value. The expression is then moved according to the rules for moving a source of the resulting data type to the data type of the destination.

If a character that is not a blank, a decimal point, a sign (+ or –), or a numeric digit is encountered during the conversion process, a “Bad digit encountered” error (\$ERR_DIGIT) occurs. (See [#37](#) in the “System Options” chapter of *Environment Variables and System Options* for more information.) Sign characters can occur at any position within the alpha source and are applied in the order in which they occur. Blanks and plus signs (+) are ignored, and a single decimal point is allowed.

If a decimal point is present, the evaluated result is implied-decimal. The implied-decimal expression then moves according to the rules for moving an implied-decimal source to an implied-decimal destination. If no decimal point is present, the evaluated result is decimal. The decimal expression then moves according to the rules for moving an alpha source to a decimal destination.

Below are examples of alpha-to-decimal assignments and their corresponding results:

```

main a_to_n

record results
    decvar      ,d6
    impvar      ,d5.3
    int4var     ,i4
    int2var     ,i2
    int1var     ,i1

proc
    decvar = "-1-2-3"      ; -123
    decvar = "123456789"  ; 456789 (leftmost digits lost)
    decvar = " 3 5 8 "    ; 358
    decvar = "9.78"       ; 10 if compiled with DBLOPT #11 not set
                           ; 9 if compiled with DBLOPT #11 set
    decvar = "abcde"      ; "Bad digit encountered" error

```

```

    impvar = " 6448.3"      ;48.300 (leftmost digits lost)
    impvar = "54.32"       ;54.320
    impvar = "19.3927"     ;19.393 if compiled with DBLOPT #11 not set
                           ;19.392 if compiled with DBLOPT #11 set

    int1var = "456"        ;-56 (high-order byte of integer lost)
    int2var = "-231.796"   ;-232 if compiled with DBLOPT #11 not set
                           ;-231 if compiled with DBLOPT #11 set
    int4var = "123456789" ;123456789
endmain

```

Moving numeric data to an alpha destination

When the destination is alpha and the source is numeric, the assignment represents a formatting operation. The value of the decimal expression is loaded into the alpha variable's data area according to whether the format is implicitly or explicitly specified as part of the statement.



If the source is integer, packed, or implied-packed, it is first converted to decimal.

Implicit formatting rules

If the source is a numeric expression, the value is transferred to the destination according to the following rules:

- ▶ The significant digits of the value are loaded right-justified over leading blanks.
- ▶ If the value is negative, a minus sign is loaded immediately in front of the leftmost digit.
- ▶ If the source is implied-decimal, the decimal point character is inserted between the whole number part and the fractional precision.
- ▶ If the number of digits and any decimal point in the source value is larger than the size of the destination, only the rightmost part is transferred. If the value is negative, the minus sign is not loaded, and no warning or error occurs.
- ▶ If the number of digits and any decimal point in the value is the same as the size of the destination, all digits of the value are transferred. If the value is negative, the minus sign is not loaded, and no warning or error occurs.

Below are examples of the implicit formatting of decimal-to-alpha assignments and their corresponding results:

```

main n_to_a

record results
    alpha6      ,a6
    alpha9      ,a9

```

```

record source
    decfld1  ,d3,          -23
    decfld2  ,d6,          -123456
    impfld1  ,d4.2,        68.54
    impfld2  ,d12.4,       12345678.9876
    intfld1  ,i1,          99
    intfld2  ,i2,          1003
    intfld4  ,i4,          82355623

proc
    open(1, o, "tt:")

    alpha6 = decfld1          ; "  -23"
    alpha6 = decfld2          ; "123456" (minus is lost)
    alpha6 = impfld1          ; " 68.54"
    alpha6 = impfld2          ; "8.9876" (leftmost digits lost)

    alpha9 = decfld2          ; " -123456"
    alpha9 = impfld2          ; "5678.8976"

    alpha6 = intfld1          ; "    99"
    alpha6 = intfld2          ; "   1003"
    alpha6 = intfld4          ; "355623" (leftmost digits lost)

    alpha9 = intfld4          ; " 82355623"
endmain

```

Explicit formatting rules

If the source is an arithmetic expression followed by a comma and an alpha expression, the assignment contains an explicit format specification. The loading of the destination is determined by a format specification in the form:

destination = source, format

If the formatted result is shorter than the destination, it is loaded right-justified over blanks. If the formatted result is longer than the destination, only the rightmost portion is loaded, and no warning or error occurs.

The format specification is an alpha field or literal. It is a sequence of characters that depicts how the destination will look. The following characters are case sensitive and have special meanings:

- X** An **X** character represents a single digit. It causes a digit from the source to be loaded into the specified position of the destination. Digits are extracted from the source beginning at the right and continuing to the left, with leading zeros ignored. Thus, the rightmost **X** in the format receives the rightmost digit in the source. If more **X** characters are specified than there are digits in the source, the leftmost **X** positions are loaded with zeros. The **X** formatting character must be uppercase.

Any **Z** or asterisk (*) formatting character that appears to the right of an **X** is treated as an **X** character.

Z A **Z** character represents a digit position and is functionally similar to **X**, except when more **Z** characters are specified than there are significant digits in the source, they are loaded with blanks. If a period appears to the left but not to the right, or if an **X** appears to the left of the **Z**, the **Z** position is treated as an **X**. The **Z** formatting character must be uppercase.

***** An asterisk represents a digit position. It is similar to an **X** character, except that when there are no more significant digits to be transferred, the position is loaded with an asterisk rather than a zero. If an asterisk is to the left of a comma when there are no more digits to be transferred, the comma is replaced with an asterisk in the destination.

\$ The money sign represents a digit position. This format character is similar to a **Z** character in that it causes a digit from the source to be loaded into the specified position of the destination, and when there are no more significant digits to be transferred, the position is loaded with a money sign.

By default, the money sign is “\$”, but the **MONEY** or **LOCALIZE** subroutine can change it to any character you want. To prevent possible confusion in formatting, do not designate any of the standard formatting characters in this section to be money signs.

– If the minus sign is used as the first or last character in a format, it causes a minus sign to be loaded at the corresponding position in the destination when the source is negative. If the source is positive, a blank is loaded. If the minus sign is used within the format, a hyphen (–) is loaded at the corresponding position in the destination.

.

A period character causes a decimal point to be loaded at the corresponding position in the destination. Any **Z** character that appears to the right of the decimal point is treated as if it were an **X**.

,

A comma character causes a comma to be loaded at the corresponding destination position under the following conditions:

- ▶ More source digits remain to be transferred.
- ▶ An **X** character appears to the left of the comma.

Any other character that appears in a format string is loaded directly into the destination.



The **Z** and **X** formatting characters must be uppercase. Otherwise, they will be loaded into the destination.

Format characters should be considered “reserved” characters in format strings that contain both text and format characters: Do *not* use a format character as normal text. For example, the second period in the following format string causes unexpected results:

```
"XX.XX lbs."
```



In some parts of the world, the use of commas and periods is reversed. Use the LOCALIZE subroutine to substitute appropriate formatting characters. (See [LOCALIZE](#) on page 9-206 for more information.)



To avoid the overhead of explicit formatting controls, use casting to place the decimal point in the destination string. This strategy honors the decimal point values, whether or not they are set in the FLAGS or LOCALIZE subroutines, although it does not insert “thousands” separators.

```
record
    number          ,d6
    result           ,a10

proc
    number = 12345
    result = ^d(number, 2)      ;Result is "    123.45"
    stop
end
```

Below are examples of ways to use the format characters in decimal-to-alpha assignments:

```
record
    alpha          ,a10

proc
    alpha = 987,      "XXXXXX-"      ;Results will be:
    alpha = -987,     "XXXXXX-"      ;"    000987 "
    alpha = 987,      "-XXX"         ;"    000987-"
                                     ;"    987"

    alpha = 987,      "XXXXXX"       ;"    000987"
    alpha = 987,      "ZZZZZZ"       ;"    987"
    alpha = -987,     "-ZZZZZZ"      ;"    - 987"
    alpha = 987,      "*****"       ;"    ***987"

    alpha = 98765,    "Z,ZZZ,ZZZ"    ;"    98,765"
    alpha = 98765,    "*,**,*"       ;"    ***98,765"
    alpha = 9,        "*,**,*"       ;"    ***.09"

    alpha = 9876,     "$$$,$$$,$$"   ;"    $98.76"
    alpha = 9876,     "$$*,**,.XX"   ;" $***98.76"
    alpha = 9876,     "Val: Z.ZZ"     ;"Val: 8.76"
```

```
alpha = 95,      "This puts a X in"  ; "This puts a 5 in"  
stop  
end
```

Controlling justification

By default, Synergy Language loads the formatted information in numeric-to-alpha assignments right-justified over blanks. You can change this default and load formatted information left-justified, however, by including a justification control as the last item in the assignment statement as follows:

statement [*justification* [:*variable*]]

statement

A numeric-to-alpha assignment statement.

justification

(optional) One of the following justification options:

LEFT Left-justify formatted information.

RIGHT Right-justify formatted information. (default)

variable

(optional) Updated with the number of characters of the formatted information that are loaded into the destination, excluding any leading blanks. (**n**)

Below are examples of controlling the justification of implicitly and explicitly formatted numeric-to-alpha assignments. Each example uses the following data division:

```
record  
alpha      ,a10  
len        ,i2
```

In the following example, the first two statements don't affect the value of **len**. The third statement, however, assigns **len** the value of 5.

```
proc                                ;The results will be:  
alpha = 12345                      ; "      12345"  
alpha = 12345 [LEFT]                ; "12345      "  
alpha = 12345 [RIGHT:len]           ; "      12345"  
stop  
end
```

In the following example, the third statement assigns **len** the value of 7. Note that the trailing blank is included.

```
proc                                     ;The results will be:
  alpha = 12345, "ZZ,ZZZ.ZZ-"           ; " 123.45 "
  alpha = 12345, "ZZ,ZZZ.ZZ-" [LEFT]    ; "123.45  "
  alpha = 12345, "ZZ,ZZZ.ZZ-" [RIGHT:len] ; " 123.45 "
  stop
end
```

In the following example, the second statement assigns **len** a value of 5.

```
proc                                     ;The results will be:
  alpha = 12345 [LEFT]                  ; "12345  "
  alpha = 12345 [LEFT:len]              ; "12345  "
  stop
end
```

In the following example, the third statement assigns **len** a value of 6.

```
proc                                     ;The results will be:
  alpha = 12345, "ZZ,ZZZ.ZZ-"           ; " 123.45 "
  alpha = 12345, "ZZ,ZZZ.ZZ-" [LEFT]    ; "123.45  "
  alpha = 12345, "ZZ,ZZZ.ZZ-" [LEFT:len] ; "123.45  "
  stop
end
```

Moving decimal/packed data to a decimal/packed destination

When the destination is decimal or packed and the source is decimal or packed, the result of the source is copied to the destination. If the size of the source is less than the size of the destination, its result is loaded right-justified over leading zeros. If the size of the source is greater than the size of the destination, the leftmost digits are truncated, and no warning or error occurs.

Below are examples of decimal-to-decimal assignments and their corresponding results:

```
record
  result    ,d6

proc                                     ;The results will be:
  result = 123                          ;000123
  result = 1234567                      ;234567 (leftmost digit lost)
  stop
end
```

Moving decimal/packed data to an implied-decimal/ implied-packed destination

When the destination is implied-decimal or implied-packed and the source is decimal or packed, the result of the source is copied to the whole number part of the destination. If the value of the expression is less than the size of the destination, the value is loaded right-justified over leading zeros. If it is greater than the size of the destination, the leftmost digits are lost and no warning or error occurs. In both cases, zeros are loaded to the right of the decimal point. The sign is maintained.

Below are examples of decimal-to-implied-decimal assignments and their corresponding results:

```
record
    result      ,d6.2
    dfld1       ,d4,    1234
    dfld2       ,d6,   -567890

proc
    result = 2*dfld1           ;The results will be:
                                ;2468.00
    result = dfld2             ; -7890.00 (leftmost digits lost)
    result = dfld1/3           ;0411.00 (source was decimal, so value
                                ; was truncated before assignment
                                ; took place)
    result = ^d(dfld1,0)/3     ;411.33 source (source was casted as
                                ; implied-decimal, so decimal places
                                ; were preserved)

    stop
end
```

Moving decimal/packed data to an integer destination

When the destination is integer and the source is decimal or packed, the binary representation of the actual value of source is copied to the destination.

If the number of bytes to be moved from the source is greater than the total number of bytes in the destination, the high-order bits from the source that cause the overflow are ignored and the sign may change. If the number of bits to be moved from the source is less than the total number of bits in the destination, sign extension occurs.

Below are examples of decimal-to-integer assignments and their corresponding results:

```
record
    result1     ,i1
    result2     ,i4
    dfld1       ,d3,    999           ;Hex 3E7
    dfld2       ,d5,   28777         ;Hex 7069
    dfld3       ,d10, 4502228111     ;Hex 10C5A8C8F
```



```

proc                                ;The results will be:
    result1 = dfld1                ; -25,           Hex E7
    result1 = dfld2                ; 105,           Hex 69
    result1 = dfld3                ; -113,          Hex 8F

    result2 = dfld1                ; 999,           Hex 000003E7
    result2 = dfld2                ; 28777,         Hex 00007069
    result2 = dfld3                ; 207260815,      Hex 0C5A8C8F
    stop
end

```

Moving implied-decimal/implied-packed data to an implied-decimal/implied-packed destination

When the destination and source are both implied-decimal or implied-packed (or one of each), the result of the source is copied to the destination.

The decimal point of the source is aligned to that of the destination. The whole number part of the source is copied to the left of the destination's decimal point, right-justified over leading zeros. If there isn't enough room to load all of the whole number digits, they are truncated at the left, and no warning or error occurs.

The fractional part of the source is copied to the right of the destination's decimal point, left-justified over trailing zeros. If there isn't enough room to load all of the decimal place digits, the source fractional part is rounded, and no warning or error occurs. (See [page 4-105](#) in this manual and [#11](#) in the "System Options" chapter of *Environment Variables and System Options* for more information on changing the default from rounding to truncating.) The sign is maintained.

Below are examples of implied-decimal-to-implied-decimal assignments and their corresponding results (with rounding in effect):

```

record
    result      ,d6.2
    idfld1      ,d6.1,      12345.6
    idfld2      ,d4.3,      9.876

proc                                ;The results will be:
    result = idfld1                ; 2345.60 (leftmost digits lost)
    result = idfld2                ; 0009.88 (rightmost digits lost)
    result = idfld2 - dfld1        ; -2335.72 (left and rightmost
    stop                          ; digits lost)
end

```

Moving implied-decimal/implied-packed data to a decimal/packed destination

When the destination is decimal or packed and the source is implied-decimal or implied-packed, the whole number part of the source is copied to the destination. By default, the value of the implied expression is rounded to a whole number. (See [page 4-105](#) in this manual and [#11](#) in the “System Options” chapter of *Environment Variables and System Options* for more information on changing the default from rounding to truncating.) If the value of the expression is less than the size of the destination, the value is loaded right-justified over leading zeros. If the expression is greater than the size of the destination, the leftmost digits are lost, and no warning or error occurs. The sign is maintained.

Below are examples of implied-decimal-to-decimal assignments and their corresponding results (with rounding in effect):

```
record
    result      ,d4
    idfld1      ,d4.2,      98.12
    idfld2      ,d7.2,      12345.67

proc
    result = idfld1      ;The results will be:
    result = idfld2      ;0098
    stop                ;2346 (leftmost digits lost)
end
```

Moving implied-decimal/implied-packed data to an integer destination

When the source is implied-decimal or implied-packed and the destination is integer, the binary representation of the whole number part of the source is copied to the destination. By default, the value of the implied expression is rounded to a whole number. (See [page 4-105](#) in this manual and [#11](#) in the “System Options” chapter of *Environment Variables and System Options* for more information on changing the default from rounding to truncating.)

If the number of bits to be moved from the source is greater than the total number of bits in the destination, the high-order bits from source that cause the overflow are ignored and the sign may change. If the number of bits to be moved from the source is less than the total number of bits in the destination, sign extension occurs.

Below are examples of implied-decimal-to-integer assignments and their corresponding results:

```
record
    result1    ,i1
    result2    ,i4
    idfld1     ,d4.2,      8.72
    idfld2     ,d5.2,      535.99
    idfld3     ,d10.4,     852436.0292
proc
    ;The results will be:
    result1 = idfld1      ;9,          Hex 9
    result1 = idfld2      ;24,         Hex 18
    result1 = idfld3      ;-44,        Hex D4

    result2 = idfld1      ;9,          Hex 00000009
    result2 = idfld2      ;536,        Hex 00000218
    result2 = idfld3      ;852436,     Hex 000D01D4
    stop
end
```

Moving integer data to a decimal/packed destination

When the destination is decimal and the source is integer, the ASCII representation of the value of the source is copied to the destination. When the destination is packed and the source is integer, the compressed BCD representation of the value of the source is copied to the destination. The destination is loaded right-justified over leading zeros. If the destination is too small to load the source's digits, the leftmost digits are lost, and no warning or error occurs. The sign is maintained.

Below are examples of integer-to-decimal assignments and their corresponding results:

```
record
    result     ,d6
    ifld1      ,i1,      46
    ifld2      ,i2,      8220
    ifld4      ,i4,      1230984
proc
    ;The results will be:
    result = ifld1        ;000046
    result = ifld2        ;008220
    result = ifld4        ;230984 (leftmost digit lost)
    stop
end
```

Moving integer data to an implied-decimal/implied-packed destination

When the destination is implied-decimal and the source is integer, the ASCII representation of the value of the source is copied to the destination. When the destination is implied-packed and the source is integer, the compressed BCD representation of the value of the source is copied to the destination.

If the value of the source is less than the size of the destination, the value is loaded right-justified over leading zeros. If it is greater than the size of the destination, the leftmost digits are lost, and no warning or error occurs. In both cases, zeros are loaded to the right of the decimal point. The sign is maintained.

Below are examples of integer-to-implied-decimal assignments and their corresponding results:

```
record
    result      ,d6.2
    ifld1       ,i1,          65
    ifld2       ,i2,          5000
    ifld4       ,i4,          9232417

proc
    result = ifld1           ;The results will be:
    result = ifld2           ;0065.00
    result = ifld4           ;5000.00
    result = ifld4           ;2417.00 (leftmost digits lost)
    stop
end
```

Moving integer data to an integer destination

When the destination and source are both integer, the significant bits of the source are copied to the low-order bits of the destination.

If the number of bits to be moved from the source is greater than the total number of bits in the destination, the high-order bits from the source that cause the overflow are ignored and the sign may change. If the number of bits to be moved from the source is less than the total number of bits in the destination, sign extension occurs.

Below are examples of integer-to-integer assignments and their corresponding results:

```
record
    result1     ,i1
    result2     ,i2
    result4     ,i4
    ifld1       ,i1,  -99           ;Hex 9d
    ifld2       ,i2,  1003          ;Hex 03EB
    ifld4       ,i4,  8235623       ;Hex 007DAA67
```

```

proc                                ;The results will be:
    result1 = ifld1                 ; -99,           Hex  9D
    result1 = ifld2                 ; -21,           Hex  EB
    result1 = ifld4                 ; 103,           Hex  67

    result2 = ifld1                 ; -99,           Hex  FF9D
    result2 = ifld2                 ; 1003,          Hex  03EB
    result2 = ifld4                 ; -21913,        Hex  AA67

    result4 = ifld1                 ; -99,           Hex  FFFFFFF9D
    result4 = ifld2                 ; 1003,          Hex  000003EB
    result4 = ifld4                 ; 8235623,       Hex  007DAA67
    stop
end

```

Assigning an object handle to another object handle

You can assign one object handle to another object handle if the destination object type matches or is a parent of the object type of the source object instance. For the following example, assume **class4** inherits from **class3**:

```

c3      ,@class3
c4      ,@class4
c3 = new class3()                  ;Valid
c3 = new class4()                  ;Valid
c4 = new class4()                  ;Valid
c4 = new class3()                  ;Invalid

```

If you assign one non-null object handle (for example, **var1**) to another object handle (**var2**), the reference to **var1**'s object is automatically cleared, and **var1** now references **var2**'s object.

In the assignment below, a new string object is created that contains "abc".

```
strfld = "abc"
```

When assigning one handle to another, as in the example below, the compiler ensures that the right side can be assigned to the left side.

```
hand1 = hand2
```

If the right side can be assigned to the left side, the following occurs at runtime:

- ▶ The reference to **hand1**'s original object, if any, is released, which may in turn cause the destruction of the object pointed to by **hand1** (if it was the last reference).
- ▶ The reference to **hand2** from **hand1** is added.

The following example makes an equality test that will use the `op_Equality` method:

```
if (obj == 10)
```

In an IF statement like the latter example, if an object is on either side of the equality operator (or any Boolean-style operator), the compiler searches its class type for an `op_Equality` (or the corresponding `op_` operator). The compiler looks for a public static method with the specified name that has two parameters that match the types on both sides of the operator. If the method is found, it is changed into a method call of that operator. If the right side is a different type, that type is also searched to find the operator.

The following assignment isn't possible, because you cannot assign an object to an `int`. A compiler error would be generated for this statement:

```
hand1 = int
```

Compile-Time Expressions and Optimizations

By default, or if you specify the **-qoptimize=1** compiler option, the compiler evaluates expressions at compile time and performs other optimizations.

You can specify any expression that can be completely evaluated at compile time in a data declaration. This includes expressions whose operands are literals and intrinsic function calls that can be evaluated at compile time. The following functions are evaluated at compile time:

- ▶ ^ARGNUM
- ▶ ^VARIANT
- ▶ ^SIZE
- ▶ ^DEFINED
- ▶ %ABS
- ▶ %ATRIM
- ▶ %BIN
- ▶ %BIT_IS_CLR
- ▶ %BIT_IS_SET
- ▶ %CHAR
- ▶ %DATECOMPILED
- ▶ %DECML
- ▶ %FALSE
- ▶ %HEX
- ▶ %IMPLIED
- ▶ %INT
- ▶ %INTEGER
- ▶ %OCT
- ▶ %RND
- ▶ %ROUND
- ▶ %STRING
- ▶ %SYSID
- ▶ %TRIM
- ▶ %TRIMZ
- ▶ %TRUE
- ▶ %TRUN

Expressions

Compile-Time Expressions and Optimizations

- ▶ `%TRUNCATE`
- ▶ `%UNSIGNED`
- ▶ `%VMS`
- ▶ `%ZONED`

For example, the following is valid:

```
record
    fld    ,a(^size(prior_rec)+2)
```

As another example, on the following function:

```
%false
```

the compiler will evaluate `%FALSE` and emit a zero to the object file (instead of having the runtime evaluate `%FALSE`). On the function

```
%atrim("ab  ")
```

the compiler will emit the literal `ab` to the object file.

Optimizations

The compiler optimizes expressions as much as possible. At level 1, the compiler performs the following optimizations:

- ▶ Constant folding occurs (except for objects in expressions). For example, `A = B + 10 + 2` is converted to `A = B + 12`.
- ▶ Expressions such as `A = A + B`, `A = A - B`, and so forth are optimized to use the `+=`, `-=`, `*=`, and `/=` assignment operators. See [“Additional assignment operators” on page 3-9](#) for more information.
- ▶ Descriptors for unused variables are not emitted to the target `.dbo` file.
- ▶ Literal pooling occurs when multiple literals point to a single string. For example, if there are two literals `“AB”` and `“ABC”`, the compiler only stores `“ABC”` in the literal space. Alpha literals are sorted largest first to improve reuse.
- ▶ `A+=1` and `a = a+1` are optimized to `incr` for noninteger numeric variables.
- ▶ `A-=1` and `a = a - 1` are optimized to `decr` for noninteger numeric variables.
- ▶ `Var(cnt, cnt +offset)` is optimized to a relative range operation (`cnt:offset`).
- ▶ `Var(cnt, cnt + var -1)` is optimized to a relative range operation.
- ▶ A numeric variable `.ne.0` is optimized to the equivalent of `IF var`.
- ▶ A numeric var `.eq.0` is optimized to `.NOT.`
- ▶ An integer-to-integer store uses an `ISTO` opcode.

- ▶ An alpha-to-alpha store uses an ASTO opcode.
- ▶ Any function that can be resolved at compile time is replaced by the compile-time constant, for example, `%char(0)`.
- ▶ Expressions are reordered (except for expressions involving objects). Expressions containing mixtures of data types are ordered to evaluate related data types as long as the reordering does not affect the result. For instance, all integers are evaluated together, then all decimal variables. Therefore, the evaluation order of an expression may not be exactly the same left-to-right order that you've specified. For example, as of the publishing of this manual, the expression

$$I1 + D2 - 22 + -2.4 + F2 + D1 + 25 - F1 - I2$$

(where **I1** and **I2** are integer variables, **D1** and **D2** are decimal variables, and **F1** and **F2** are implied-decimal variables) is evaluated as follows:

$$I1 - I2 + \%integer(3) + D1 + D2 - F1 + F2 - 2.4$$

- ▶ The changing of single dimension real array access to pseudo array access (that is, changing the DIMEN opcode to the INDEX opcode) does not occur when **-qstrict** is specified due to enforcing the bounds checking of real arrays on all accesses.
- ▶ The code generator uses integer literals when numeric literals are passed as arguments to a method, function, or subroutine and both a prototype exists and the argument is either **n** or **i** (or **i1**, **i2**, **i4**, or **i8**).
- ▶ Real array access using an index of literal 1 is optimized to a nondimensioned access.

4

Synergy Language Statements

Introduction to Statements 4-6

Discusses statement format, including continuation lines, blank lines, and comments, and briefly describes the data and procedure divisions.

Compound Statements 4-9

Defines compound statements.

Statement Syntax 4-10

Contains specification pages for the following Synergy Language statements:

ACCEPT – Receive a character from a specified channel	4-11
ADDHANDLER – Register a method as an event handler for an event.....	4-13
BEGIN-END – Begin and end a compound statement	4-15
CALL – Call an internal subroutine	4-16
CASE – Select a statement from a set of statements	4-18
CLASS-ENDCLASS – Define a class	4-22
CLEAR – Clear a variable	4-26
CLOSE – Close a channel	4-28
COMMON-ENDCOMMON – Define a shared data record.....	4-30
DATA – Add a local stack variable declaration.....	4-33
DECR – Decrement a numeric variable	4-35
DELETE – Delete a record from an ISAM file.....	4-37
DETACH – Disconnect program from terminal	4-39
DISPLAY – Send output to a device or file	4-40
DO FOREVER – Repeatedly execute a statement	4-44
DO-UNTIL – Repeatedly execute a statement.....	4-45
END – End a routine.....	4-46
ENDPARAMS – End parameter list	4-47
ENUM – Declare an enumeration	4-48

EXIT – Terminate a BEGIN-END block.....	4-51
EXITLOOP – Terminate a loop.....	4-53
EXITTRY – Exit a TRY statement.....	4-54
EXTERNAL FUNCTION-ENDEXTERNAL – Declare an external function	4-55
FIND – Find a record.....	4-57
FLUSH – Flush all buffers.....	4-61
FOR-DO – Repeatedly execute a statement	4-62
FOR-FROM-THRU – Repeatedly execute a statement.....	4-64
FOR-UNTIL-DO – Repeatedly execute a statement (alternate).....	4-66
FOREACH – Iterate sequentially through elements in a collection	4-68
FORMS – Process ASCII control codes.....	4-70
FRETURN – Return a value to the calling routine.....	4-71
FUNCTION-ENDFUNCTION – Define a function.....	4-72
GET – Receive data from a channel	4-75
GETS – Get sequential binary data.....	4-77
GLOBAL-ENDGLOBAL – Define a global data section	4-80
GOTO – Transfer program control	4-84
GROUP-ENDGROUP – Define a group	4-86
IF – Conditionally execute a statement.....	4-90
IF-THEN-ELSE – Conditionally execute one of two statements (alternate).....	4-92
IMPORT – Import prototypes.....	4-94
INCR – Increment a numeric variable	4-96
INIT – Initialize a named data structure to its default values	4-98
LITERAL-ENDLITERAL – Define a literal.....	4-99
LOCASE – Convert uppercase characters to lowercase.....	4-101
LPQUE – Queue a file for printing by the system.....	4-102
MAIN-ENDMAIN – Explicitly define a main routine.....	4-105
MERGE – Combine two or more files.....	4-107
METHOD-ENDMETHOD – Define a method	4-112
MRETURN – Return control from a method	4-116
NAMESPACE-ENDNAMESPACE – Define a namespace	4-117
NEXTLOOP – Terminate current loop iteration	4-118
NOP – Perform no operation	4-119

OFFERROR – Disable runtime errors traps	4-120
ONERROR – Enable trapping of runtime errors.....	4-121
OPEN – Open a channel.....	4-124
PROC-END – Begin and end the procedure division	4-135
PROPERTY-ENDPROPERTY – Declaring a property.....	4-136
PURGE – Purge a channel.....	4-140
PUT – Write fixed-length data	4-141
PUTS – Write sequential fixed-length data	4-143
READ – Read a specified record.....	4-145
READS – Read the next sequential record.....	4-150
RECORD-ENDRECORD – Define a data record.....	4-156
RECV – Receive a message	4-160
REMOVEHANDLER – Unregister a registered event handler	4-162
REPEAT – Repeatedly execute a statement	4-163
RETURN – Return control from a subroutine.....	4-164
SEND – Send a message.....	4-165
SET – Assign a value to a set of variables.....	4-167
SLEEP – Suspend program execution.....	4-168
SORT – Sort a file	4-169
STOP – Terminate program execution.....	4-175
STORE – Store a record to an ISAM file	4-177
STRUCTURE-ENDSTRUCTURE – Define how data is laid out.....	4-179
SUBROUTINE-ENDSUBROUTINE – Define an external subroutine.....	4-182
THROW – Throw an exception.....	4-185
TRY-CATCH-FINALLY-ENDTRY – Handle exceptions.....	4-186
UNLOCK – Release a record lock on a channel	4-189
UPCASE – Convert lowercase characters to uppercase.....	4-191
USING – Conditionally execute a statement.....	4-192
USING-RANGE – Conditionally execute a statement with a range.....	4-194
WHILE – Repeatedly execute a statement	4-196
WRITE – Write a record to a file	4-198
WRITES – Write the next sequential record to a channel.....	4-202

XCALL – Call an external subroutine or function	4-204
XRETURN – Return control to a calling routine	4-205

Input and Output Statement Qualifiers 4-207

Contains specification pages for the following statement qualifiers:

ALLOC – Pre-allocate a number of blocks	4-208
BKTSIZ – Set bucket size.....	4-209
BLKSIZ – Set block size	4-210
BUFNUM – Set number of process local I/O buffers	4-211
BUFSIZ – Set buffer size.....	4-212
CONTIG – Allocate contiguity of a file	4-213
DEQ – Set number of blocks to add	4-214
DIRECTION – Set direction of a READS.....	4-215
FDL – Specify an FDL or XDL string or file specification.....	4-216
GETRFA – Return a record’s RFA.....	4-218
GUIWND – Open a new application window	4-221
KEYNUM – Specify a key of reference	4-222
LOCK – Control record locking	4-223
MASK – Specify activation characters	4-225
MATCH – Define how a record is located	4-227
NOFILL – Prevent the data area from being padded with blanks	4-229
NUMREC – Preallocate a number of records.....	4-230
OPTIONS – Specify runtime options	4-231
POSITION – Define the file position	4-233
RECSIZ – Specify a record size for a file.....	4-234
RECTYPE – Define the type of records to write to a file.....	4-236
RFA – Locate record with the specified RFA.....	4-237
SHARE – Control file access.....	4-239
TEMPFILE – Define an intermediate file for output mode.....	4-241
WAIT – Specify wait time for an I/O statement to finish processing.....	4-242

Record Locking 4-243

Describes the two types of record locking provided by Synergy Language: automatic locking and manual locking.

Record File Addresses (RFAs) 4-246

Explains the usage and storage of RFAs and static RFAs.

Introduction to Statements

Statement format

A Synergy Language statement is a single logical line of information and can be as long as 2048 characters of compilable text (excluding tabs and comments). Preceding and trailing blanks and tabs are ignored. We recommend that you adopt reasonable indentation conventions to make your programs as readable as possible; legibility simplifies all other aspects of program development and maintenance.

Continuation lines

Because it is impractical to work with lines that are longer than about 120 characters (80 on most CRT terminals), you'll sometimes need to split a logical line into several shorter physical lines. The second and all subsequent parts of the statement line are called continuation lines, and their first nonblank character must be an ampersand (&).

The Synergy compiler treats a physical line and all continuation lines that follow it as one logical line. Therefore, you cannot insert any noncontinuation line—regardless of whether it is a blank line, a comment, a compiler directive, or anything else—between related continuation lines or between the first physical line and its continuation lines. A physical line cannot be longer than 255 characters (excluding line terminators, such as carriage returns and line feeds).

Blanks and tabs can precede the ampersand to improve readability. For example:

```
lb102,
    val = "klh"      ;Transmitter
    &   "112385"     ;Package
    &   "txye1"      ;Destination
    &   "2b3"        ;Priority, class, level
```

can also be written as

```
lb102,
    val = "klh"      ;Transmitter
    & "112385"       ;Package
    & "txye1"        ;Destination
    & "2b3"          ;Priority, class, level
```

Both statements are functionally identical to

```
lb102,
    val = "klh112385txye12b3"
```

All characters that follow an ampersand are considered to be part of the statement. To avoid programming problems, be careful when splitting an alpha literal across a continuation boundary, and do not include comments inside a quoted string. If the opening delimiter for the literal is on one

line and the closing delimiter is on the next, the contents of the literal resume immediately after the ampersand. For example, assuming there are no blank spaces or tabs at the end of each physical line, the following statement:

```
1b102,
    val = "klh      ;Transmitter
          & 112385  ;Package
          & txye1   ;Destination
          & 2b3"    ;Priority, class, level
```

is equivalent to

```
1b102,
    val = klh      ;Transmitter 112385  ;Package txye1   ;Destination 2b3
```

which is quite different from the result of the first two examples and probably not what you are looking for. See [“Alpha literals” on page 1-22](#) for further discussion.

Comments

We highly recommend that you use numerous comments in your code to clarify what your program is doing, making it easier to maintain.

A comment begins with the first semicolon (;) that is not part of an alpha literal, and it continues to the end of the line. Comments are ignored by the compiler.

For example, in the following line of code, “;Initialize name” is a comment and is ignored by the compiler:

```
name = "John Doe"    ;Initialize name
```

Blank lines

Logical lines that consist entirely of blanks, tabs, or a comment are printed in the program listing but are otherwise ignored by the Synergy compiler. For example, the third line below is ignored:

```
1b102,
    code = "klh"      ;Transmitter's code

    snum = "112385"   ;Package serial number
    dest = "txye1"    ;Destination ID
```

Data division

The data division is the section of a Synergy Language routine in which you define most of the data that you will use in the procedure division of your program. The variables you define can be local to the routine or shared with other routines. Your field definitions are grouped into one or more of the following data structures: record, group, common, global data section, literal, class, or structure.

Here are some examples of data division statements:

```
global data section glb
    record rec1
        field1          ,a6,    "record"
        field2          ,d3.2
    endrecord
endglobal

global common rec2
    field3              ,a*,    "This is the initial value"
    field4              ,2d7
    field5              ,d4
    field5a             ,d2 @field5
    field5b             ,d2 @field5+2
endcommon

record rec3
    field6              ,a10
                      ,d4
    group grp1          ,a
        field7          ,d3,    542
        field8          ,d4
    endgroup
endrecord

public class class1
    public structure mystruct
        field9          ,i4
        field10         ,a20
    endstructure
endclass
```

Procedure division

The procedure division is the section of a Synergy Language routine that follows the data division. It contains the executable statements of your program and defines the processing algorithms. It begins with the PROC statement and ends either at the physical end-of-file or at the END statement.

The operators and operands used in expressions within procedure division statements are documented in [chapter 3, “Expressions.”](#)

Compound Statements

A compound statement is a collection of simple statements preceded by a BEGIN and followed by an END. You can include a compound statement within a compound statement.

A compound statement is considered to be a single unit and can appear anywhere a single statement is allowed. Because they have multiple substatements, compound statements can improve the readability of your programs.

For example, consider the following sequence of statements:

```

    if (option .ne. SCAN) goto noscan
    find(chn, record, ^FIRST)
    start = 1
    if (.not. endflag) goto maxend
    display(TTCHN, "Maximum records to scan ")
    reads(TTCHN, input(1:^size(end)))
    end = input
    goto forloop

maxend,
    end = MAXRECS

forloop,
    for idx from start thru end
        xcall proc_rec(idx)

noscan,
    .
    .
    .

```

Compare the preceding code with the following code, which uses compound statements:

```

    if (option .eq. SCAN)
        begin
            find(chn, record, ^FIRST)
            start = 1
            if (endflag) then
                begin
                    display(TTCHN, "Maximum records to scan")
                    reads(TTCHN, input(1:^size(end)))
                    end = input
                end
            else
                end = MAXRECS
            for idx from start thru end
                xcall proc_rec(idx)
            end
        end
    end

```

Statement Syntax

The following pages provide the syntax and usage for the Synergy Language statements.

ACCEPT – Receive a character from a specified channel

ACCEPT (*channel*, *variable*[, *label*][, WAIT:*wait_spec*]) [*error_list*]

Arguments

channel

The open channel from which to obtain the character. (**n**)

variable

The variable that will contain the received character. (**a** or **n**)

label

(optional) The label to which to transfer control if *variable* is an alpha variable and an end-of-file character is received.

WAIT

(optional) Specifies how long to wait for terminal input. See [WAIT](#) on page 4-242 for more information.

error_list

(optional) An I/O error list. If any one of the specified errors occurs as part of the ACCEPT, control is transferred to the associated label.

Discussion

The ACCEPT statement receives the next sequential character from the specified channel and loads it, or its numeric value, into *variable*.

If *variable* is alpha, the input character is loaded into the first byte of *variable*, leaving the remaining positions unchanged. If an end-of-file character is encountered and *label* is specified, control immediately transfers to *label*. If an end-of-file character is encountered and *label* is not specified, an “End of file” error (\$ERR_EOF) is generated.

If *variable* is numeric, the input character’s corresponding value is loaded. [Appendix B](#) lists all the ASCII characters and their corresponding integer codes. If an end-of-file character is encountered, the optional *label* specification does not apply; instead, the integer value is loaded (for example, 004 for CTRL+D on UNIX).

The ACCEPT statement is often used to get a character from a terminal device. To accommodate all operating systems on which Synergy Language runs, a carriage return character and a line feed character are both generated when you press ENTER. Your program must account for both the carriage return and the additional line feed.



Because ACCEPT is a character-level input operation, no special handling of record terminators occurs (as normally occurs in record-oriented operations such as READS); therefore, your program must handle such characters. Thus, it is more efficient to use GETS with the MASK qualifier and wait for serial I/O (see [GETS on page 4-77](#) for more information). For more information about serial ports on UNIX, see [INITPORT on page 9-160](#).

See also

[“I/O error lists” on page 6-3.](#)

Examples

The examples below use the following data division.

```
.define TTCHN          ,1

record
    afld          ,a5
    dfld1         ,d1
    dfld2         ,d5
```

The following example receives the next character from the terminal. If the character is an end-of-file marker, the program terminates with an “End of file” error (\$ERR_EOF).

```
accept(TTCHN, afld)
```

The following example also receives the next character from the terminal. If the character is an end-of-file marker, control transfers to the statement labeled **done**.

```
accept(TTCHN, afld, done)
```

The following example is functionally identical to the previous example, but it uses the I/O error list construct.

```
accept(TTCHN, afld) [eof=done]
```

The following example receives the next character from the terminal and load the corresponding value of that character into **dfld2**. The **end** label is ignored because **dfld2** is a numeric variable. If the received character is an end-of-file marker, no special error handling occurs.

```
accept(TTCHN, dfld2, end)
```

In the following example, **dfld1** is only one character long; therefore, only the low-order digit of the ASCII character integer code is loaded into **dfld1**.

```
accept(TTCHN, dfld1)
```

ADDHANDLER – Register a method as an event handler for an event

WIN

ADDHANDLER (*event*, *method_name*)

Arguments

event

The name of an event, or an object reference to a field that is of the type `Synnet.DotNetEvent` or inherited from `Synnet.DotNetEvent`.

method_name

The name of the method (including the instance) that will handle the event.

Discussion

The ADDHANDLER statement can only be used with code generated by the **gennet** utility. (See “[The Gennet Utility](#)” in the “General Utilities” chapter of *Synergy Language Tools* for more information about **gennet**.)

Method_name is in the form *instance.method*. For example, if **c1** is an instance of **class1**, *method_name* could be any of the following:

- ▶ **c1.mymeth**
- ▶ **this.mymeth** (for this instance in a class)
- ▶ **parent.mymeth** (for the parent’s instance in a class)

When registering delegates with ADDHANDLER, you are allowed to use generated classes as arguments, which allows a method like the following to be called as a delegate

```
method meth1          ;Definition of method registered with the delegate
    p1                ,@System.Object
    p2                ,@System.EventArgs
proc
.
.
.
endmethod
```

You can register more than one event handler for an accessible event using ADDHANDLER. You can also register an event handler for an event more than one time. If an event handler is registered multiple times, the event handler will be called once for each registration when the event is fired.

See also

- ▶ [REMOVEHANDLER](#) on page 4-162.
-

BEGIN-END – Begin and end a compound statement

```
BEGIN  
.  
.  
.  
END
```

Discussion

No statement is allowed on the same line as BEGIN.

See [“Compound Statements” on page 4-9](#) for more information about BEGIN-END blocks.

CALL – Call an internal subroutine

CALL *label*

or

CALL (*label*[, ...]), *value*

Arguments

label

Either a specific label to which execution control is transferred, or a list of labels (separated by commas) to which control is transferred based on the specified selection in *value*.

value

An expression whose result is used to select a label from a list of labels. Execution control is transferred to the selected label. A result of 1 selects the first label in the list, a result of 2 selects the second label, and so forth. If the result is less than or equal to 0, or greater than the number of labels, no transfer occurs. (**n**)

Discussion

The CALL statement transfers control to an internal subroutine. The internal subroutine is entered through the statement whose label matches the label that is specified or selected in the CALL statement. After returning from the internal subroutine, execution continues with the statement following the most recently executed CALL statement.

Internal calls use the same data area as the calling routine.

See also

[“Writing Subroutines and Functions” on page 7-2](#) for more information about internal subroutines.

Examples

```
.align LONG
record
    select      ,i2
    type       ,i2
    agedate    ,d8
    date       ,4d8
    cutoff     ,d8
```

```

proc
    xcall choice(select)           ;Get "select" value
    call(zapper, zinger), select ;Depending on select value, call either
                                ; zapper or zinger
    .                             ;Point A
    .
    .
zinger,
    agedate = date(type)
    .
    .
    .
    call inner
    .                             ;Point B
    .
    .
    return                       ;Return to point A

inner,
    if(date .gt. cutoff)
        over = 1

zapper,
    .
    .
    .
    return                       ;Return to point B if entered at inner
                                ;Return to Point A if entered at zapper

```

CASE – Select a statement from a set of statements

```
CASE expression OF
  BEGINCASE
    [label_1:] statement_1
    .
    .
    .
    [label_n:] statement_n
  ENDCASE
  [ELSE exception]
```

Arguments

expression

An expression whose value determines which statement to execute. (**a** or **n**)

label_1

(optional) A label that identifies the first statement in the set of statements from which to select.

statement_1

The first statement in the set that can be selected according to the result of *expression*.

label_n

(optional) A label that identifies the *n*th statement in the set of statements from which to select.

statement_n

The *n*th statement in the set that can be selected according to the result of *expression*.

exception

(optional) A statement to be executed if the value of *variable* does not select a statement in the set. If a statement is selected from the set, the optional ELSE is ignored.

Discussion

CASE selects a statement from a set of statements for execution. It consists of a selection variable, a set of statements from which one is to be executed, and an optional ELSE statement (which is executed if no statement is selected from the set).

Three types of CASE statements exist:

- ▶ Unlabeled CASE
- ▶ Labeled numeric CASE
- ▶ Labeled alpha CASE

In an unlabeled CASE statement, no statement has a label (*label_1* through *label_n* are omitted). *Expression* is non-implied numeric and is interpreted as an ordinal number 1–*n*, where *n* is the number of statements in the set. For example, a value of 6 selects the sixth statement in the set.

In a labeled CASE statement, all statements are identified with labels. The labels must all be literals and must be the same type as *expression* (alpha or numeric). Implied labels are allowed with implied numeric expressions only. The label (*label_1*, *label_2*, and so forth) can be an alpha literal, a numeric literal, or two numeric literals separated by a hyphen (-). In the latter case, the value of the first literal must be less than or equal to the value of the second literal; a selection value that falls within this range selects the associated statement.

When using a CASE selection variable that is not implied (**i**, **d**, **p**, or **n**), only integer or decimal values are allowed for labels. If the selection variable is declared as **n** and an implied variable is passed to the routine and used in a CASE statement, the implied variable will be rounded (following normal rounding rules) before being compared to the CASE labels.

When using a CASE selection variable that is implied (**d**., **p**., or **n**.), labels can be either integer, decimal, or implied values. Implied values will only match a label with the same exact value or a label range that includes the value.



If you use alpha labels, be aware that the comparisons are .EQ. and .NE., which will be compared for the length of the shortest field. If you would like to match against the exact label, you should use the USING statement with the label (.EQS."ABC").



CASE labels cannot be referenced by program control statements such as CALL and GOTO.

After a statement is selected and executed, control is transferred to the statement following the CASE. If a selected statement is a compound statement, execution proceeds to the end of the compound statement, and then control is transferred to the statement following the CASE.

If the result of *expression* exceeds the decimal range 1–*n* (where *n* is the number of statements in the set) or doesn't match a case label, control is transferred to the ELSE. If no ELSE exists, no statement is executed and the entire CASE statement is skipped.



Although functionally equivalent to a cascaded IF-THEN-ELSE statement, when using a numeric *expression*, CASE is more efficient and easier to maintain. The USING statement is as fast as an unlabeled CASE statement but provides more maintainable and readable code. CASE is most efficient when using an **i** or **d** *expression* for selection.

See also

- ▶ [USING on page 4-192.](#)
- ▶ [USING-RANGE on page 4-194.](#)

Examples

The following unlabeled CASE statement consists of four selectable statements.

```
function math
  a_select ,n
  a_v1     ,n
  a_v2     ,n
record
  result   ,i2
proc
  case a_select of
    begincase
      result = a_v1 + a_v2           ;Executed if a_select = 1
      result = a_v1 - a_v2           ;Executed if a_select = 2
      result = a_v1 * a_v2           ;Executed if a_select = 3
      result = a_v1 / a_v2           ;Executed if a_select = 4
    endcase
  else
    clear result
  freturn result
endfunction
```

If **a_select** has a value other than 1, 2, 3, or 4, the statement following the ELSE is executed.

The next example consists of three selectable statements. The first is a simple assignment statement that is executed if **a_select** has a value 2. The second is a compound statement that is selected if **a_select** has a value of 5, 6, or 7. The third is also a compound statement that is selected by an **a_select** value of -1. If **a_select** has a value other than 2, 5, 6, 7, or -1, no statement is executed.

```
case a_select of
  begincase
2:
  v3 = v1 - v2
5-7:
  begin
    v3 = v1 + v2
    v4 = v3 * factr4
  end
-1:
  begin
    write(TTCHN, varrec, recloc)
    call varinit
    varopt = 17
  end
endcase
```

In the following example, the CASE statement interprets **option** keywords entered by the user.

```
function math
    a_v1      ,n
    a_v2      ,n
record
    option    ,a4
    result    ,i2

proc
    repeat
        begin
            display(TTCHN, "Enter option:  ")
            reads(TTCHN, option) [eof=done]
            case option of
                begincase
                    "HELP":
                        call help
                    "ADD":
                        result = a_v1 + a_v2
                    "SUB":
                        result = a_v1 - a_v2
                    "MUL":
                        result = a_v1 * a_v2
                    "DIV":
                        result = a_v1 / a_v2
                endcase
            else
                writes(TTCHN, "...Invalid option")
            end
        done,
        freturn result

    help,
        xcall math_help
    endfunction
```

In this example, if the user enters an option that is not in the set of options, control transfers to the ELSE statement.

CLASS-ENDCLASS – Define a class

```
[access] [class_mod] CLASS name [EXTENDS parent_class] [IMPLEMENTS interface, ...]
    member_def
    .
    .
    .
ENDCLASS
```

Arguments

access

(optional) One of the following access modifiers:

PUBLIC	Access is not restricted. This is the most accessible option. (default)
PROTECTED	Access is limited to the containing class or types derived from the containing class.
PRIVATE	Access is limited to the containing type. This is the least accessible option.

class_mod

(optional) One or more of the following class modifiers:

ABSTRACT	The class cannot be instantiated.
PARTIAL	The class definition can be separated into two or more definitions that will be compiled together as a single definition.
SEALED	The class cannot be extended.

name

The name of the class to define.

EXTENDS

(optional) The class will extend the specified parent class and will thus inherit members from that parent class. See the Discussion section for more information.

parent_class

The name of a parent class from which the class inherits.

IMPLEMENTS

(optional) One or more interface names will follow.

interface

The name of an interface to implement. Multiple interfaces must be separated by commas.

member_def

The declaration of a method, function, subroutine, field, property, indexer, structure, operator, delegate, enumeration, event, or nested class within the class. You can specify as many member definitions as you want, although you must specify at least one.

Discussion

The CLASS-ENDCLASS statement declares a class in the current namespace or within another class.

SEALED and ABSTRACT are mutually exclusive. Typically, an ABSTRACT class contains one or more abstract methods and/or properties. Abstract methods can only exist in abstract classes.

Using the PARTIAL modifier, it is possible to split the definition of a class over two or more source files. Each source file contains part of the class definition, and all of these parts are combined when your application is compiled, as long as no syntax errors or naming collisions exist in the partial class declarations. You may want to split a class definition when working on a large project, for example, to enable multiple developers to work on different parts of the class simultaneously. Partial classes must be compiled together in one compilation unit (for example, a **.dbr** file or ELB); they can't be split across compilation units.

If the EXTENDS option is not specified, the compiler assumes the parent class is the System.Object class (which is the ultimate parent class of all objects). When a class extends a parent class, it inherits the following members from the immediate parent class, as well as any members that the parent inherits from its parent, in recursive fashion:

- ▶ All methods (not constructors)
- ▶ All fields and properties
- ▶ All nested classes

If *parent_class* is a fully qualified class name, the class definition is implicitly imported.



Neither a common record nor a global data section may be declared within a class.

An interface enables you to define required features, avoid changes that can break your program, and enforce consistent class design throughout different implementations. A class member can implement an interface member if the following criteria are met:

- ▶ The class's signature, including the return type, is the same as the interface member.
- ▶ The class has public accessibility.
- ▶ The class is non-static.

When you declare that a class implements an interface, that class, or one of its ancestors, must implement all members in the interface as well as any interfaces from which that interface inherits.

You can declare an explicit interface member by prefixing the interface name and a dot (.) to the class member name. This is particularly useful if a class implements more than one interface with the same member names that require different implementations. This rule applies to all interface members.

For example, you could declare a method that implements interface **Iface.mymethod** as follows:

```
public class class1 implements Iface
public method Iface.mymethod, void
proc
    mreturn
endmethod
endclass
```

Explicit interface members are only accessible via a variable of that interface type. For example, if you wanted to access **Iface.mymethod**, you would need to do the following:

```
var1, @Iface
var1 = (Iface)obj
var1.mymethod()
```

Nested classes

Nested classes have the same requirements as regular classes, with the following exceptions:

- ▶ A nested class cannot extend any of its outer classes.
- ▶ A nested class declaration can use an outer class for the following within the nested class declaration:
 - ▶ Return types, argument types, or local variable types in methods
 - ▶ Field and property types
- ▶ Nested classes are PRIVATE by default.

See [“Nested classes” on page 8-9](#) for more information.

See also

[“Accessibility” on page 8-16.](#)

Examples

In the example below, **class3** accesses **class2**’s private member **m_number**.

```
class class2
    class class3
        public method writeit, void
            c2, @class2
        proc
            open(1,o,"tt:")
```

```

        writes(1, "num="+%string(c2.m_number))
        close(1)
        mreturn
    end
endclass
public method class2
proc
    m_number = 567
    mreturn
end
private m_number, i4
endclass

```

The example below creates a custom attribute by extending System.Attribute and specifying the attributes usage by applying the AttributeUsage attribute to the new class.

```

[AttributeUsage(AttributeTargets.All)]
public class MyAttribute extends System.Attribute
public method MyAttribute
    parm1, string
proc
    name = parm1
end
private name, string
endclass

```

CLEAR – Clear a variable

```
CLEAR variable[, ...]
```

Arguments

variable

One or more variables to be cleared. (**a** or **n**)

Discussion

The CLEAR statement initializes one or more variables to blanks or their zero representations.

If *variable* is alpha, it is cleared to spaces. If *variable* is numeric, it is cleared to its zero representation. A CLEAR statement can include different data types.

CLEAR also affects object lifetime. Clearing an object is like setting it to ^NULL (and on Synergy standard reduces the reference count).



By default, the CLEAR statement clears a record to spaces. If you are using integer variables, group them together in one record, which should be aligned. Cast the record as an integer when you clear it:

```
.align LONG
record binary_data
    var      ,i4
    intarr   ,[10]i4

proc
    clear ^i(binary_data)
```

The integer variables contain zero.

If data types are mixed, use INIT instead of CLEAR.

See also

[.ALIGN on page 5-4](#) for information about aligning a record.

Examples

In the following example:

```
record
    avar      ,a1
    dvar      ,3d1
    idvar     , [15]d2.2
```

```
proc
    clear avar, dvar, idvar[ ]
end
```

the CLEAR statement is functionally equivalent to

```
avar = " " ;blank
dvar(1) = 0
for i from 1 thru 15
    idvar[i] = 0.0
```

CLOSE – Close a channel

```
CLOSE channel [, ...]
```

or

```
CLOSE channel THRU channel
```

Arguments

channel

A channel to close. (**n**)

Discussion

The CLOSE statement completes any pending I/O and terminates the association of a channel with its file or device.

When a channel is closed, all locks on the specified channel are released and the channel is available for future use. After a channel is closed, an I/O statement cannot reference that channel until an OPEN statement reactivates it.

If a channel is opened in output or update mode, any data contained in the system cache is output before the channel is released during a CLOSE. This ensures that any information written by the program is physically transferred to the device. The CLOSE statement saves the output file. If *channel* is not open, CLOSE returns without an error (it does nothing).



The PURGE statement deletes the file opened on *channel* if the file is opened in output mode.

VMS

When you CLOSE a mailbox, EOF is written to the mailbox for compatibility with DIBOL. If you PURGE the channel, EOF is not written.

If you do not CLOSE a terminal channel in a program and output is still pending when the program stops, the output is discarded. This feature prevents the program from hanging forever when a Lat, telnet, or modem control device has gone offline.



When using serial devices with modems attached, to avoid having your program hang on a CLOSE if the state of the port is unknown or hung, use PURGE in conjunction with the INITPORT subroutine and the GETS statement.

See also

[PURGE on page 4-140.](#)

Examples

```
open(TMPCHN, o, "test")           ;Create test file
writes(TMPCHN, "Record 1")
writes(TMPCHN, "Record 2")
close TMPCHN                       ;Save test file
open(TMPCHN, i, "test")           ;Open test file for input
reads(TMPCHN, testrec)            ;Read first record
close TMPCHN                       ;Close channel
open(TMPCHN, o "temp")            ;Open temporary work file
write(TMPCHN, testrec, 13)         ;Write the 13th record
read(TMPCHN, testrec, 13)         ;Read record back in
purge TMPCHN                      ;Delete temporary work file and
                                ; close channel
```

COMMON-ENDCOMMON – Define a shared data record

```
[form] COMMON [name] [,X] [,NOSUFFIX]
  member_def
.
.
.
[/ENDCOMMON]
```

Arguments

form

(optional) Defines the record as one of the following forms:

GLOBAL	Causes the statement to allocate data space in a shared data area.
EXTERNAL	Causes the statement to reference data space that was allocated by a GLOBAL COMMON statement in another routine.

name

(optional) The name of the record to define. If *name* isn't present, the common record is unnamed.

,X

(optional) Indicates that you're defining an overlay for the most recently defined nonoverlay common. If you specify ",X" on an external common record, the overlay indicator is ignored.

NOSUFFIX

(optional) Indicates that the common record name will not contain a trailing dollar sign (\$) when declared or referenced. See the Discussion section for more information.

member_def

Either the definition of a field within the record or a group declaration. Each field definition has the syntax described in ["Field definitions" on page 2-20](#). Each group declaration has the syntax described in [GROUP-ENDGROUP on page 4-86](#). You can specify as many member definitions as you want.

Discussion

The COMMON statement defines a shared data record and its component fields. A COMMON record can be either GLOBAL, which allocates the data space, or EXTERNAL, which references it. Fields in an EXTERNAL COMMON record reference the data in the GLOBAL COMMON record by defining where the field starts by matching the name. The size of the data referenced in an EXTERNAL COMMON field is defined locally by the size specified and does not have to match the GLOBAL COMMON field's size.

GLOBAL and EXTERNAL are mutually exclusive. If you don't specify either one, GLOBAL is the default in a main routine and EXTERNAL is the default in a subroutine or function. You can only define a shared data area once, in *one* global common record within a program. However, you can reference that data area in any number of external common records within a program. Neither external common records nor global common records are valid in a global data section.

For each common record name and field name within a common record, the compiler appends a dollar sign (\$) to the end of the name. This sign ensures that common names are unique and do not conflict with global names. Specifying the NOSUFFIX modifier inhibits this dollar sign from being appended. You might use NOSUFFIX if you're referencing or defining data for other languages in which the dollar sign is not a valid or reasonable character. If you use NOSUFFIX, be careful that your common name is unique and doesn't conflict with other global names, including external subroutine and function names.

At least one field definition must be present after the COMMON statement. Initial values are ignored in EXTERNAL COMMON statements, except to determine the size of an automatically sized field designated with a size of *.

You cannot specify common variables with the same name unless they have unique paths. For example, a "Symbol already uniquely defined" error is generated if you specify the following:

```
common
    smith      ,a10
common
    smith      ,a10
```

The size of the shared data area defined by the COMMON statement is the sum of the sizes of the data areas allocated to all of its members. The size is defined by the global declaration.



Object handles are not allowed in common records. Also, a common record cannot be declared within a class.

Examples

```
subroutine aaa
    arg      ,a
global common cmn
    fld1     ,a3, "abc"
    fld2     ,a4, "defg"
    fld3     ,a5, "hijkl"

proc
    arg = fld1           ;arg = "abc"
    arg = fld2           ;arg = "defg"
    arg = fld3           ;arg = "hijkl"
    arg = cmn            ;arg = "abcdefghijkl"
    xreturn
endsubroutine
```

```
;Subroutine bbb shows that although the order of the fields has changed,  
;the data accessed is the same since the fields are the same size.  
;  
;The cmn record still accesses the same data as defined in subroutine aaa  
;even though the order of the fields locally has changed, since the name  
;is matched to the GLOBAL COMMON name and the size is still the same.
```

```
subroutine bbb
```

```
    arg      ,a  
external common cmn  
    fld3     ,a5  
    fld1     ,a3  
    fld2     ,a4
```

```
proc
```

```
    arg = fld1           ;arg = "abc"  
    arg = fld2           ;arg = "defg"  
    arg = fld3           ;arg = "hijkl"  
    arg = cmn            ;arg = "abcdefghijkl"  
    xreturn
```

```
endsubroutine
```

```
;Subroutine ccc shows that although the order of the fields is the same,  
;the data accessed is different, since the fields are different sizes from  
;what was specified when the fields were defined.
```

```
;
```

```
;The beginning of the data accessed is defined by the name of the field.
```

```
;The size of the data accessed is defined locally by the size of the  
;local field.
```

```
;
```

```
;The cmn record still accesses the same data as defined in subroutine aaa  
;even though the size of the fields locally has changed, since the name is  
;matched to the GLOBAL COMMON name and the size is still the same.
```

```
subroutine ccc
```

```
    arg      ,a  
external common cmn  
    fld1     ,a6  
    fld2     ,a2  
    fld3     ,a4
```

```
proc
```

```
    arg = fld1           ;arg = "abcdef"  
    arg = fld2           ;arg = "de"  
    arg = fld3           ;arg = "hijk"  
    arg = cmn            ;arg = "abcdefghijkl"  
    xreturn
```

```
endsubroutine
```

DATA – Add a local stack variable declaration

DATA *name*, [*dim*, ...] *type*[*size*][, *init_value*]

Arguments

name

The name of the local stack variable. (**a**)

dim

(optional) One or more array dimensions (real or dynamic), separated by commas (for example, [2] or [#,#]). You must type the brackets around the dimension elements; they do not mean that the arguments are optional. The brackets must contain at least one element.

type

Any valid routine field data type. See [page 2-22](#) for a complete list of valid data types.

size

(optional) An expression indicating the number of characters in each occurrence of the field.

init_value

(optional) An initial value to be assigned to successive occurrences of the field. The *init_value* specification is described in more detail in “[Specifying an initial value](#)” on [page 2-29](#).

Discussion

The DATA statement defines a local stack variable in the procedure division. The scope of the local variable is limited to the block in which it is declared. It can have the same name as a local variable in another block. The DATA statement must precede any other type of statement in the block.

A call to a label in a higher scope is not allowed in a BEGIN-END block that contains a DATA statement using an object variable.

Examples

In the example below, an **i4** local variable named **myfield** is declared.

```
public method mymethod, void
    x, i4
proc
    if (x .eq. 5)
        begin
            data myfield, i4
            .
            .
            .
```

DECR – Decrement a numeric variable

DECR *variable*

Arguments

variable

The variable to be decremented. (**n**)

Discussion

The DECR statement decreases the value of a specified numeric value by 1.

The value of *variable* can be positive or negative.

If the decremented value of *variable* overflows its size, *variable* is set to its zero representation. For example, the following code fragment decrements **dvar** from -9 to 0, because **dvar** is a **d1** field instead of a **d2**:

```
record
    dvar          ,d1,    -9

proc
    decr dvar
```



Using the DECR statement is more efficient than subtracting 1 from a variable.



With integer variables, if the variable has the maximum negative value for its size, the following statements have different results:

```
decr ivar

ivar = ivar - 1
```

In the example below, **i1 = i1 - 1** is equivalent to **i1 = -129**, which results in 127. See [“Moving integer data to an integer destination” on page 3-38](#) for more information. This behavior is consistent with the behavior of decimal variables, which may be used as counters.

```
record
    i1      ,i1,    -128                ;Max negative value for an i1
    field

proc
    decr i1                            ;Result is 0
    i1 = -128
    i1 = i1 - 1                        ;Result is 127
```

See also

[INCR on page 4-96.](#)

Examples

The following example:

```
decr dvar(2,5)
```

is functionally equivalent to

```
dvar(2,5) = dvar(2,5) - 1
```

DELETE – Delete a record from an ISAM file

```
DELETE (channel) [error_list]
```

Arguments

channel

The channel on which the ISAM file is open in update mode (U:I). (n)

error_list

(optional) An I/O error list. If any one of the specified errors occurs during the DELETE, control is transferred to the associated label.

Discussion

The DELETE statement deletes the current record on the specified channel. This statement is used exclusively with ISAM files on Windows and UNIX.

VMS

You can use DELETE to delete records from a relative file. You cannot use U:* if you want to delete records. You must open the file with an explicit submode (for example, U:I) to allow the DELETE to work.

You can use the DELETE statement only after a FIND, READ, or READS statement. The current record being deleted must be locked if the file can be accessed by other users; otherwise, you'll get a "No current record" error (\$ERR_NOCURR).

See also

["I/O error lists" on page 6-3.](#)

Examples

```
.define TTCHN      ,1
.define ISMCHN     ,2
record rec
    data1          ,a30
    key            ,a16
    data2          ,a25
record
    io             ,a20

proc
    open(TTCHN, o, "tt:")
    display(TTCHN, "ISAM file to examine: ")
    reads(TTCHN, io, done)
```

```
open(ISMCHN, u:i, io)
repeat
  begin
    display(TTCHN, "Key: ")
    reads(TTCHN, io, done)
    read(ISMCHN, rec, io)
    writes(TTCHN, "Data: " + rec)
    display(TTCHN, "Delete this record? ")
    reads(TTCHN, io, done)
    if (io .eq. 'Y')
      delete(ISMCHN)
    end
  end
done,
close ISMCHN
stop
end
```


DETACH – Disconnect program from terminal

UNIX

DETACH

Discussion

The DETACH statement places the current program in the background and returns control of the terminal to the shell, thus enabling two programs to run concurrently from the same terminal. When a program executes a DETACH statement, it creates a new process that is detached from the terminal and continues to run as a background process. Any subsequent I/O request to the terminal from the detached program competes with I/O requests from programs running in the foreground.

The new process can only inherit certain characteristics from the original process. Because locks on UNIX are process-based, locks cannot be inherited by the new process. Any locks that a program has before a DETACH statement is executed are lost when the program detaches. If you need something locked after the DETACH statement, don't lock it until after the DETACH.

DISPLAY – Send output to a device or file

`DISPLAY (channel, item[, ...]) [ [error_list] ]`

Arguments

channel

The channel of the output device or file. (**n**)

item

One or more of the following items separated by commas:

- ▶ Alpha expression
- ▶ Numeric expression
- ▶ Terminal-independent screen function:

\$SCR_POS	Reset the absolute position of the cursor.
\$SCR_CLR	Clear a specific portion of the screen.
\$SCR_MOV	Move the cursor relative to the current position.
\$SCR_ATT	Modify one or more defined screen attributes.

See “[Terminal-independent screen function commands](#)” on [page 4-41](#) for a description of each terminal screen function.

error_list

(optional) An I/O error list. If any one of the specified errors occurs as part of the DISPLAY, control is transferred to the associated label.

Discussion

The DISPLAY statement sends character-oriented output to valid output devices or files through a specified channel.

The elements of *item* are processed in the order that they are specified. If *item* is an alpha expression, the resulting text is directly transmitted to the output device.

If *item* is a numeric expression, the value’s corresponding ASCII character is transmitted. [Appendix B](#) lists all the ASCII characters and their corresponding codes. If the result of *item* is greater than 256, it is divided by 256, and the ASCII character associated with the remainder is transmitted. For example, the result 321 is divided by 256, with a remainder of 65, and the corresponding ASCII character “A” is sent to the output device or file.

You can additionally use the DISPLAY statement to control terminal screens. Depending on the terminal on which you are operating, special functions, such as clearing CRT screens and positioning cursors, are performed by transmitting multiple-character function sequences.

Synergy Language provides terminal-independent screen function commands to eliminate hard-coding terminal-dependent screen functions in DISPLAY statements.

VMS

Performing an OPEN to the terminal, where terminal output is redirected to a disk file, results in separate records for each DISPLAY statement. On other operating systems, Synergy Language concatenates all such data in the file.

Terminal-independent screen function commands

The screen functions described in this section are based on the ANSI terminal standard. Some terminals may not use these functions or may use them differently from how they are documented here. See the “[Terminal Characteristics on UNIX](#)” chapter in your *Professional Series Portability Guide* to find out what functions your terminal supports.

\$SCR_POS(*row*, *column*)

row

An expression representing the row to which to position the cursor. (**n**)

column

An expression representing the column to which to position the cursor. (**n**)

\$SCR_POS resets the absolute position of the cursor. If either *row* or *column* is not in the range 1–255, the cursor is not moved, and no warning or error occurs.

\$SCR_CLR(*function*)

function

One of the following options:

SCREEN	Clear the entire screen.
EOL	Clear from the current position to the end of the line.
EOS	Clear from the current position to the end of the screen.
LINE	Clear the current line.
BOL	Clear from the beginning of the line to the current position.
BOS	Clear from the beginning of the screen to the current position.

\$SCR_CLR clears a specific portion of the screen.

\$SCR_MOV(*row_change*, *col_change*)

row_change

The number of rows to move and the direction of movement. A negative value indicates movement up the screen; and a positive value indicates movement down the screen. (**n**)

col_change

The number of columns to move and the direction of movement. A negative value indicates movement to the left in the current row; and a positive value indicates movement to the right in the current row. (**n**)

\$SCR_MOV moves the cursor relative to the current position. If one of the values is zero, the cursor does not move in that direction. If both values are zero, the cursor does not move.

\$SCR_ATT(*function*[, ...])

function

One or more of the following options separated by commas:

CLEAR	Turn off (clear) all attributes.
BOLD	Turn on bolding.
UNDER	Turn on underlining.
BLINK	Turn on blinking.
REVERSE	Turn on reverse video.
GON	Turn on graphics.
GOFF	Turn off graphics.

UNIX, VMS

SAVE	Save current attributes.
RESTORE	Restore attributes from last SAVE.

\$SCR_ATT modifies one or more defined screen attributes.

See the “[Terminal Characteristics on UNIX](#)” chapter in your *Professional Series Portability Guide* to check if your terminal supports the SAVE and RESTORE options.

Multiple attribute functions are processed in the order they are presented in the \$SCR_ATT command.



Once an attribute is set, it remains in effect until it is cleared. You can't clear individual screen attributes; you must clear all the attributes and then reset the ones you want.

Examples

Assuming TTCHN is the channel number associated with the terminal, the following example

```
display(TTCHN, "Enter option:  ")
```

sends this message to the terminal:

Enter option:

No other characters are transmitted.

In the following example, if **ernum** is a decimal expression and **errmsg** is an alpha array of error messages, this statement sends the error message associated with **ernum** to the specified channel. The error message is preceded by a bell character (7).

```
display(TTCHN, 7, "Error:  ", errmsg(%ernum))
```

The example below first clears the entire screen, positions the cursor to row 15 and column 30, and displays the following message:

This is a test.

It then clears all screen attributes, sets the bolding and blinking attributes, positions the cursor to row 1 and column 1, and displays the message

Sorting ...

```
display(TTCHN, $SCR_CLR(screen), $SCR_POS(15, 30), "This is a test.")
display(TTCHN, $SCR_ATT(clear, bold, blink), $SCR_POS(1, 1),
&      "Sorting ...")
```

See also

- ▶ The “[Terminal Characteristics on UNIX](#)” chapter in your *Professional Series Portability Guide* for how to find out what screen functions are available on your terminal.
- ▶ The programming specifications provided by the manufacturer of your terminal to determine what special functions are available for your terminal and what the corresponding request sequences are.
- ▶ “[I/O error lists](#)” on page 6-3.
- ▶ [Page 19-28](#) for a list of decimal values for ASCII line-drawing characters.

DO FOREVER – Repeatedly execute a statement

DO FOREVER *statement*

Arguments

statement

A single or compound statement to be repeatedly executed.

Discussion

The DO FOREVER statement executes a statement repeatedly until control is transferred outside the DO FOREVER loop by any of the following methods:

- ▶ An EXITLOOP statement
- ▶ A GOTO statement
- ▶ Error trapping

This statement is functionally equivalent to the REPEAT statement.

Statement can be on the next logical line.

See also

[REPEAT on page 4-163.](#)

Examples

The following example reads data from the terminal until an end-of-file character is typed.

```
.define TTCHN      ,1
record
    buffer          ,a60

proc
    xcall flags(7004020, 1)
    open(TTCHN, o, "tt:")
    display(TTCHN, $SCR_CLR(screen))
    do forever
        begin
            display(TTCHN, $SCR_POS(10, 1), $SCR_CLR(line), "Enter data: ")
            reads(TTCHN, buffer, eof)
        end
    eof,
        forms(TTCHN, 1)
    close TTCHN
    stop
end
```

DO-UNTIL – Repeatedly execute a statement

DO statement UNTIL condition

Arguments

statement

A single or compound statement to be repeatedly executed.

condition

An expression that controls whether *statement* is executed again.

Discussion

The DO-UNTIL statement executes a statement repeatedly until a specified condition is true. After each execution of the specified statement, the condition is evaluated. If *condition* evaluates to false, *statement* is executed again. If *condition* evaluates to true, *statement* is not executed again, and control is transferred to the statement following the DO-UNTIL statement.

DO, *statement*, and UNTIL can be on separate logical lines.

Examples

The following example stores 10 times the index value of each element into the first ten elements of **a** (for example, $a(6) = 60$).

```
record
  i          ,d2
  a          ,10d4

proc
  clear i
  do
    begin
      incr i
      a(i) = i * 10
    end
  until (i .ge. 10)
end
```

END – End a routine

END

Discussion

The END (or .END) statement indicates the termination of main routines, subroutines, functions, and methods. (Main routines, subroutines, functions, and methods can alternatively be terminated with ENDMAIN, ENDSUBROUTINE, ENDFUNCTION, or ENDMETHOD, respectively.)

Although the END statement is recognized with or without a beginning period, we recommend that you don't include the period.

When you end a routine with the END statement, replacement identifiers (.DEFINES) remain defined until the entire file is compiled. To change the behavior of the END statement so that it clears all replacement identifiers immediately after the routine is processed, use the **end** option with the **-qrelaxed** compiler option.

Examples

```
proc
.
.
.
end

subroutine subl
proc
.
.
.
end

function func1
proc
.
.
.
end
```


ENDPARAMS – End parameter list

ENDPARAMS

Discussion

ENDPARAMS is an optional keyword that can be added to the end of any parameter list to indicate the list is terminating. If ENDPARAMS is not used, the first declaration that is not a parameter declaration terminates the parameter list.

Example

The example below explicitly terminates the parameter list after **parm2**.

```
public method meth1, void
    parm1      ,int
    parm2      ,int
    endparams
proc
end
```

ENUM – Declare an enumeration

```
[access] ENUM name
    member_def [, value]
    .
    .
    .
ENDENUM
```

Arguments

access

(optional) One of the following access modifiers:

PUBLIC	Access is not restricted. This is the most accessible option.
PROTECTED	Access is limited to the containing class or types derived from the containing class.
PRIVATE	Access is limited to the containing type. This is the least accessible option. (default)

name

The name of the enumeration to define.

member_def

The name of the member. You can specify as many member definitions as you want, although you must specify at least one.

value

(optional) The value of the member.

Discussion

The ENUM statement enables you to declare one or more enumerations globally or within a class or namespace. You can declare one or more enumeration values within an enumeration by declaring a unique identifier for each value. If the identifier is not unique within the enumeration, a “Symbol already uniquely defined” error (TOKUDF) is generated.

If the starting enumeration value does not have an explicit integer value, the compiler assumes the integer value to be 0. For any subsequent field without an explicit integer value, the compiler assumes its integer value to be one more than the previous enumeration value’s integer value.

For example:

```
enum myenum
    f0          ;Assumed to be 0
    f1          ;Assumed to be 1
```

```

f2          , 7
f3          ;Assumed to be 8
f4          ;Assumed to be 9
f5          , 3
f6          ;Assumed to be 4
endenum

```

Specifying duplicate integer values within an enumeration is valid. For example, the following is allowed:

```

enum SystemState
  off        , 0
  down       , 0

```

If an enumeration inside a namespace or class hides a global enumeration, a level 4 “*Enum_name* enumeration hides global enumeration of same name” warning is generated, where *enum_name* is the name of the enumeration.

If *member_def* is one of the following reserved keywords, a comma is required after *member_def* whether *value* is specified or not: end, endclass, endenum, endexternal, endfunc, endfunction, endglobal, endliteral, endmain, endname, endnamespace, endparams, endrecord, endsub, endsubroutine, global, main, proc, start, record, common, literal. For example:

```

enum myenum
  value1     , 10
  main       ,
  start      , 10
  record     ,
endenum

```

See also

[“enumeration” on page 2-12.](#)

Examples

```

import System.Collections

enum SearchMode
    All, 1
    Selected, 2
endenum

main EnumTest
    record
        Results, @ArrayList
    endrecord
    proc
        SearchForSomething(SearchMode.All,,Results)
        stop
    endmain
1
2

subroutine SearchForSomething
    required in SearchType      ,SearchMode
    optional in Filter          ,a
    required out Results        ,@ArrayList
    endparams
    proc
        using SearchType select
            (SearchMode.All),
                . ;Code to return ALL in Results goes here
                .
                .
                nop
            (SearchMode.Selected),
                . ;Code to return SELECTED in Results goes here
                .
                .
                nop
        endusing
        xreturn
    endsubroutine

```

EXIT – Terminate a BEGIN-END block

EXIT [*label*]

Arguments

label

(optional) A label on a BEGIN statement for a BEGIN-END block in which the EXIT statement is contained.

Discussion

The EXIT statement is specified within a BEGIN-END block and transfers control to the END statement of the current BEGIN-END block.

The optional label format is useful in a BEGIN-END block nested within another BEGIN-END block. The label enables you to exit the outer BEGIN-END block by referring to the label of that BEGIN statement.

Examples

The following example receives a message from the Synergy message manager. If the destructive receive flag is true, the message is deleted from the message pool. See [NEXTLOOP on page 4-118](#) for an example that demonstrates the difference between the NEXTLOOP, EXITLOOP, and EXIT statements.

```
subroutine chk_msg
    a_dstflg ,d                      ;Destructive receive flag
record
    msg      ,a80
    len      ,d3,                  -1

proc
    if (a_dstflg) then
        begin
            recv(msg, none1)        ;Receive and delete message
            writes(TTCHN, msg)
            exit
        none1,
            writes(TTCHN, "No messages")
        end
    else
        begin
            recv(msg, none2, "[SELF]", len)
            writes(TTCHN, msg)
            exit
        end
    end
end
```

```
        none2,  
        write(TTCHN, "No messages")  
    end  
    xreturn  
endsubroutine
```

EXITLOOP – Terminate a loop

EXITLOOP

Discussion

The EXITLOOP statement terminates a looping statement (such as DO FOREVER, FOR, REPEAT, and WHILE). After terminating the loop, control is transferred to the statement immediately following the looping statement.

Examples

The following subroutine prints a large report and checks after each page prints to see if the user typed a “Q” to quit printing. See [NEXTLOOP on page 4-118](#) for an example that demonstrates the difference between the NEXTLOOP, EXITLOOP, and EXIT statements.

```
subroutine prnt_rpt
    a_file      ,a
    a_data      ,a
external function
    get_pg      ,i
record
    page        ,d3
    input       ,a1

proc
    page = 1
    repeat
        begin
            if (.not. %get_pg(page, a_data))
                exitloop
            xcall prnt_pg(a_data)
            if (%ttsts)
                begin
                    accept(TTCHN, input)
                    if (input .eq. 'Q')
                        exitloop
                end
            incr page
        end
    xreturn
endsubroutine
```

EXITTRY – Exit a TRY statement

EXITTRY

Discussion

The EXITTRY statement exits the current TRY-CATCH level. For example, if you have an EXITTRY in a TRY-CATCH block within another TRY-CATCH block, the EXITTRY exits one level.

EXITTRY can only be present within a TRY block; it is not allowed in a FINALLY block.

Examples

```
subroutine s1
  param      ,i
  record
    x        ,i4
  proc
    try
      begin
        x = param
        if (param .eq. 5)
          exittry      ;Will transfer control to "x = 6" below
        x = 2
      end
    catch (e)
      x = 5
    endtry
    x = 6
  end
end
```


EXTERNAL FUNCTION-ENDEXTERNAL – Declare an external function

```
EXTERNAL FUNCTION
    function_1 ,access_type
    function_2 ,access_type
[ENDEXTERNAL]
```

Arguments

function

The name of the external function you want to use.

access_type

The access type of the value returned by the function. *Access_type* is one of the following values:

- a** Alpha
- d** Decimal
- d.** Implied-decimal
- i** Integer
- p** Packed
- p.** Implied-packed
- ^VAL** Return integer by value

Discussion

The EXTERNAL FUNCTION statement declares one or more functions that your program will use and defines their return data types.

All nonprototyped, nonlocal user-defined functions must be declared with the EXTERNAL FUNCTION statement before you can use them. The EXTERNAL FUNCTION statement is *not* used with intrinsic functions; however, occasionally it is required for a system-supplied external function. When this is the case, the documentation will make clear what needs to be declared.

If an external function has the same name as an intrinsic function, the external function declaration replaces the intrinsic function.

Examples

The example below calls the external function **sqroot**, which returns a decimal value.

```
record
    val1      ,d4
    val2      ,d4
    result    ,d6.4
external function
    sqroot    ,d

proc
    .
    .
    .
    result = %sqroot(val1, val2)
```

FIND – Find a record

```
FIND (channel, [record], [key_spec][, GETRFA:new_rfa][, KEYNUM:krf_spec]
&      [, LOCK:lock_spec][, MATCH:match_spec][, POSITION:pos_spec]
&      [, RFA:match_rfa][, WAIT:wait_spec])  [ [error_list] ]
```

Arguments

channel

The channel on which the file is open. The channel must already have been opened in input or update mode. **(n)**

record

The record area that, for non-ISAM files, determines the record size. *Record* is required for non-ISAM files and optional for ISAM files. **(a)**

key_spec

(optional) Either an expression whose result is the key to find, the ordinal record number for relative files, or one of the following options:

^FIRST Position to the first record in the file.

^LAST Position to the last record in the file.



On Windows and UNIX, ^LAST works only on ISAM and relative files. On OpenVMS, ^LAST works only on ISAM files.

GETRFA

(optional) Returns the record's RFA. See [GETRFA on page 4-218](#) for a complete description.

KEYNUM

(optional) Specifies a key of reference by which to perform FIND for ISAM files. See [KEYNUM on page 4-222](#) for a complete description.

LOCK

(optional) Specifies whether the record is to be locked. See [LOCK on page 4-223](#) for a complete description.

MATCH

(optional) Defines how a specified key is matched. See [MATCH on page 4-227](#) for a complete description.

POSITION

(optional) Specifies an absolute position in the file. See [POSITION on page 4-233](#) for a complete description.

RFA

(optional) Finds the record with the specified RFA. See [RFA on page 4-237](#) for a complete description.

WAIT

(optional) Specifies how long to wait for a record lock to be released. See [WAIT on page 4-242](#) for a complete description.

error_list

(optional) An I/O error list. If any one of the specified errors occurs as part of the FIND, control is transferred to the associated label.

Discussion

The FIND statement positions to a record that can be retrieved by the next READS statement. It is functionally similar to the READ statement, except that FIND does not retrieve the record from the file; it only sets a pointer to that record so that it can be retrieved by the next READS statement. The READS statement must be the next I/O operation on the same channel. (DIRECTION and REVERSE qualifiers on that READS statement are ignored.)

FIND does not lock the record unless you specify the LOCK qualifier or compile with the FIND lock compiler option (/find_lock on OpenVMS or -F on all other systems).

The record to position to is specified either by absolute position (^FIRST, ^LAST, or the POSITION qualifier), the MATCH qualifier (valid on ISAM files only), the RFA qualifier, or *key_spec* in the following order of precedence:

1. If absolute position is specified, MATCH, RFA, and *key_spec* are ignored, unless POSITION:Q_IGNPOS is specified.
2. If the MATCH qualifier is specified, the RFA qualifier is ignored, unless MATCH:Q_RFA is specified. *Key_spec* is also ignored if MATCH:Q_RFA or MATCH:Q_SEQ is specified.
3. If absolute position is not specified but RFA is, *key_spec* is ignored.
4. If absolute position, MATCH, and RFA qualifiers are not specified, *key_spec* defines the record to position to, with the key of reference determined either by KEYNUM or the logic described in step 2 above.

If you specify a key value in *key_spec* and the key value length is greater than the record's key field, only the first part of the key value locates the record. If the key's length is less than the size of the key field, the value is called a partial key. In this case, the record located is the first one having a key that begins with the specified value.

The key of reference is determined as follows:

- ▶ If the KEYNUM qualifier is specified, it defines the key of reference.
- ▶ If KEYNUM is not specified, FIND uses the implied key of reference. The implied key of reference is a field within *record* that matches the offset and contiguous length of the key defined by *key_spec*. If a match is not found, the first key that matches the offset and has a greater length than *key_spec* is the implied key of reference.
- ▶ If KEYNUM is not specified and the conditions for an implied key of reference do not exist, FIND uses the primary key as the key of reference if system option #45 is not set.



We recommend that you use the KEYNUM option rather than relying on the implied key of reference; KEYNUM is more efficient.

VMS

If you specify both the RFA and KEYNUM qualifiers on the FIND statement, KEYNUM must specify the primary key as the key of reference (Q_PRIMARY or 0).

ISAM

If the MATCH qualifier is not specified, FIND matches on values greater than or equal to the specified value. If no record's key field begins exactly as specified by the key value, the first record with a higher key value is located, and a "Key not same" error (\$ERR_KEYNOT) occurs. If no higher record exists, an "End of file" error (\$ERR_EOF) occurs.

If you want to FIND a segmented key that is specified by *key_spec*, you must first construct that key by concatenating each segment together. You can use the %KEYVAL intrinsic function to return the extracted key value from the specified record. See [%KEYVAL on page 9-201](#) for more information.

The syntax

```
find(channel, ..., ^LAST)
```

is only available for consistency with Synergy Language version 4 and has been superseded by the POSITION I/O qualifier.



See the note under "[Record Locking](#)" on [page 4-243](#) for information about inconsistencies that may occur after an I/O error is encountered.

See also

- ▶ [“Input and Output Statement Qualifiers” on page 4-207.](#)
- ▶ [“I/O error lists” on page 6-3.](#)

Examples

```
subroutine toc
  a_chn      ,d
  .include "REC:custmas.ddf"           ;Customer master file layout

proc
  find(a_chn, customer,, POSITION:Q_FIRST)
  repeat
    begin
      reads(a_chn, custmas, eof)       ;Read the records
      writes(TTCHN, name)              ;Display names
    end

  eof,
  xreturn
endsubroutine
```

FLUSH – Flush all buffers

FLUSH *channel*

Arguments

channel

The channel on which the file is open. (**n**)

Discussion

The FLUSH subroutine flushes all buffers associated with a channel.

Due to the inherent I/O buffering or caching on a system, a WRITE(S)/STORE/DELETE to a file may not immediately reach the physical disk file. System failure before the physical write, store, or delete can corrupt a file.

The FLUSH statement immediately updates a physical disk file. FLUSH improves reliability, but at the cost of performance. For ISAM files, both the index and data files are flushed.

WIN, UNIX

For superior performance, we recommend setting system option #36 instead of using the FLUSH statement.



Use FLUSH only at the end of a transaction or any critical I/O processing.

See also

[#36](#) in the “System Options” chapter of *Environment Variables and System Options*.

FOR-DO – Repeatedly execute a statement

FOR variable = value[, ...] DO statement

Arguments

variable

A variable that is modified before each execution of the loop. (**a** or **n**)

value

An expression to be assigned to *variable* before each execution of *statement*. All values must be the same data type as *variable*. (**a** or **n**)

statement

A single or compound statement to be repeatedly executed.

Discussion

The FOR-DO statement executes a statement repeatedly until a specified list of values is evaluated.

The first value in the list is assigned to *variable* before the first execution of the statement, the second value is assigned before the second execution, and so forth. When each value in the list has been assigned, the FOR statement is terminated and control is transferred to the statement following the FOR statement.

Statement can be on the next physical line. It is executed once for each value specified. *Statement* must execute code; for example, it cannot contain only a label.



The FOR and the DO keywords must always be on the same line.

Examples

The following example assigns the values 8, 28, 13, 33, and 38 to **result** elements 1, 5, 2, 6, and 7, respectively.

```
record
    v1      , i2
    v2      , i2
    i       , i2

proc
    v1 = 3
    v2 = 5
    for i = 1, 5, 2, 6, 7 do
        result(i) = v2 * i + v1
    end
```


The following example assigns four separate alpha values to a complex manipulation.

```
record
  string      ,a100
  alf         ,50a10
  len         ,i2
  maxlen      ,i2
  result      ,a200

proc
  for string = alf(12), alf(17), alf(22), alf(35) do
    begin
      xcall cnvrt(string, len)
      if (len .gt. maxlen)
        string(maxlen + 1) = '%'
      xcall parse(string, result)
      xcall act(result)
    end
  end
end
```

FOR-FROM-THRU – Repeatedly execute a statement

FOR *var* FROM *initial* THRU *final* [BY *incr*] *statement*

Arguments

var

A variable that controls the repeated execution of the statement. **(n)**

initial

An expression whose value is initially assigned to *var*. **(n)**

final

An expression for the terminating value, which is compared against the current value of *var*. **(n)**

incr

(optional) An expression whose result is added to *var* after each repetition of the statement. The default value is 1. **(n)**

statement

A single or compound statement to be repeatedly executed.

Discussion

The FOR-FROM-THRU statement executes a statement repeatedly as long as the control value is within a specified range.

If *incr* is positive, *statement* is executed while *var* is less than or equal to *final*. If *incr* is negative, *statement* is executed while *var* is greater than or equal to *final*.

Each time *statement* is executed, *incr* is added to *var*. If *initial* does not match the loop conditions, *statement* won't be executed.

Incr and *final* are evaluated only once upon entry of the FOR-FROM-THRU statement, and then they are treated as constants during the rest of FOR statement processing.

Var, *initial*, *incr*, and *final* must all be the same data type.

Statement must execute code; for example, it cannot contain only a label.

The difference between FOR-FROM-THRU and FOR-UNTIL-DO is that you cannot modify the *final* and *incr* values with FOR-FROM-THRU. FOR-UNTIL-DO, on the other hand, can modify these values.

Examples

The following example assigns the values 3, 6, 9, 12, 15, 18, 21, and so forth, through 99, to the first 33 elements of result.

```
record
    i          ,d4,  1
    d1         ,d4,  3
    result     ,100d4

proc
    for d1 from 3 thru 100 by d1
        begin
            result(i) = d1
            incr i
        end
    end
```

FOR-UNTIL-DO – Repeatedly execute a statement (alternate)

FOR var = initial[STEP incr] UNTIL final DO statement

Arguments

var

A variable that controls the repeated execution of the statement. **(n)**

initial

An expression whose value is initially assigned to *var*. **(n)**

incr

(optional) An expression whose result is added to *var* after each repetition of the statement, with the exception of the first repetition. The default value is 1. **(n)**

final

An expression for the terminating value, which is compared against the current value of *var* at the beginning of each repetition of the statement. **(n)**

statement

A single or compound statement to be repeatedly executed.

Discussion

The FOR-UNTIL-DO statement executes a statement repeatedly as long as the control value is within a specified range.

If the entering *incr* is positive, *statement* is executed while *var* is less than or equal to *final*. If the entering *incr* is negative, *statement* is executed while *var* is greater than or equal to *final*.

Incr and *final* are evaluated prior to each execution of *statement*.

Statement must execute code; for example, it cannot contain only a label.

The difference between FOR-UNTIL-DO and FOR-FROM-THRU is that FOR-UNTIL-DO can modify the *final* and *incr* values, while FOR-FROM-THRU cannot.



The FOR and the DO keywords must always be on the same line.

Examples

The following example assigns the values 65, 72, 79, and 86 to the elements of result. The first value is the initial value 65. Each of the remaining values is equal to the immediately preceding value plus 7. Only four values are assigned because **v4** is assigned a value of 93 ($86 + 7$), which is greater than the final value of 91 before the fifth repetition.

```
record
    v1          ,d3, 13
    v2          ,d3, 5
    v3          ,d3, 7
    v4          ,d3
    i           ,d3, 1
    result      ,d4

proc
    for v4 = v1 * v2 step v3 until v1 * v3 do
        begin
            result(i) = v4
            incr i
        end
    end
end
```

The following example assigns the values 20.2, 13.8, 7.4, and 1.0 to the first four elements of result.

```
record
    idval       ,d3.1
    i           ,d2, 1
    idresult    ,10d3.1

proc
    for idval = 20.2 step -6.4 until 0.0 do
        begin
            idresult(i) = idval
            incr i
        end
    end
end
```

FOREACH – Iterate sequentially through elements in a collection

```
FOREACH loop_var in collection
    <iteration processing>
    .
    .
    .
```

Arguments

loop_var

A loop variable whose value is set to each element in sequence as FOREACH iterates through them.

collection

One of the following:

- ▶ A dynamic system array of rank 1 (for example, [#]@myclass)
- ▶ A real Synergy array of rank 1 (for example, [10]a5)
- ▶ The System.Collections.ArrayList class or a descendant
- ▶ The Synergex.SynergyDE.Collections.ArrayList class or a descendant
- ▶ A class generated by the **gennet** utility that inherits from the DotNetObject class and can be converted to IEnumerable (a .NET Framework interface that implements collection-like behavior)

Dicussion

When the FOREACH statement is executed, it iterates sequentially through all elements in the collection, setting *loop_var* to each element in turn. If the collection is a dynamic array, the loop variable's type must match or be an ancestor of the element type of that array; otherwise, a "Type mismatch" error (TYPMISMCH) will occur.

The loop variable is a copy of the processed element in *collection*, so we can think of it as an assignment statement. This means changes to members of *loop_var* are not reflected back in the original collection. As a result of the implied assignment, FOREACH is less efficient than a FOR loop.

If the collection is a Synergex.SynergyDE.Collection.ArrayList, a System.Collection.ArrayList, or a descendent of one of these classes and the items of the collection cannot be cast to the type of the loop variable, the runtime will report an "Incompatible classes" error (\$ERR_INCPTCLS).

You cannot add an element to the collection, remove an element from the collection, or set a collection item to another handle while the FOREACH statement is being executed. If any of these changes occur, the runtime will report an "Invalid operation: Collection was modified" error (\$ERR_INVOPER).

A call to a label in a higher scope is not allowed in the iteration processing section of a FOREACH statement whose loop variable is an object.

Examples

The example below iterates through the **alist** collection for every customer.

```
import System.Collections

namespace ns1
  class customer
    public name, string
    public method customer
      parml, string
    proc
      name = parml
    endmethod
  endclass
endnamespace

main
record
  alist      ,@ArrayList
  c1         ,@customer
proc
  alist = new ArrayList()
  alist.Add(new customer("joe"))
  alist.Add(new customer("fred"))
  alist.Add(new customer("fran"))
  open(2, o, "tt:")
  foreach c1 in alist
  begin
    writes(2, c1.name)
  end
  close(2)
end
```

FORMS – Process ASCII control codes

FORMS (*channel*, *control_code*)

Arguments

channel

The channel of the device where the control values should be sent. (**n**)

control_code

An expression whose result is one of the following ASCII control values to be processed: (**n**)

0 Transmit a form feed.

1–9999 Transmit a specified number of new line controls.

-1 Transmit a vertical tab.

-2 Transmit a CTRL+N.

-3 Transmit a new line.

Discussion

The FORMS statement processes special ASCII control codes. It is most commonly used to control vertical alignment on line printers. Control values of less than -3 cause no action, while numbers greater than 9999 generate an “Arithmetic operand exceeds maximum size” error (\$ERR_BIGNUM).

Examples

```
forms(TTCHN, 0)           ;Transmits one form feed
forms(TTCHN, -1)          ;Transmits one vertical tab
forms(TTCHN, 15)          ;Transmits 15 new lines
```


FRETURN – Return a value to the calling routine

`FRETURN expression`

Arguments

expression

An expression whose type is the same as the return type of the function.

Discussion

The FRETURN statement enables you to specify the return value of a user-defined function and return control to the calling routine. The size of the value returned to the calling routine is the size of the result of the expression.

When FRETURN is used in a ^VAL function, the value specified in the statement should be an integer. If it is not, it is converted to an integer.

If the compiler encounters an FRETURN statement outside of a function, it generates a “Not within function or subroutine” error (\$ERR_NOTINSUB).

Examples

The following example checks to see if the current paint character needs to be changed to the specified paint character. It returns %TRUE if it changed the paint character or %FALSE if it didn't.

```
function chk_paint
  a_paint  ,a
record
  d_val    ,d3

proc
  if (%paint .ne. %decml(a_paint))
  begin
    xcall paint(a_paint)
    freturn (%true)
  end
  freturn (%false)
endfunction
```

FUNCTION-ENDFUNCTION – Define a function

```
FUNCTION name[, return_type[size]][, options, ...]  
    param_def  
    .  
    .  
    .  
[ENDFUNCTION | END]
```

Arguments

name

The name of the function.

return_type

(optional) The return type of the function being defined. See [“Where Data Types Can Be Used” on page 2-18](#) for the data types that are valid return types. If not specified, the return type defaults to the type of the variable or literal of the first FRETURN statement.

size

(optional) The size of the function return value.

options

(optional) One or more of the following modifiers:

LOCAL LOCALDATA	Keep record contents constant for as long as the function is activated at one or more levels.
STACK STACKDATA	Make record contents unique for every activation of the function.
REENTRANT	Allow multiple active instances of the function.
VARARGS	More than the declared number of arguments can be passed to this routine.

WIN, UNIX

RESIDENT	Keep the function in memory, even when not in use.

ROUND	Use rounding rules for implied-decimal data types within the function.

TRUNCATE	Use truncation rules for implied-decimal data types within the function.

param_def

Function parameter definitions. See “[Parameter definitions](#)” on page 2-33.

Discussion

The FUNCTION statement identifies the beginning of a user-defined function.

Function names of longer than 30 characters are truncated when the routine is linked (although you can specify up to 255 characters as long as the first 30 characters are unique).

The LOCAL and STACK modifiers specify the default state of an unqualified RECORD statement. For example, given the following statement:

```
function fred ,STACK
```

all unqualified RECORD statements in function **fred** are treated as if they are STACK RECORD statements. You can specify either LOCAL or STACK, but not both. If you don’t specify LOCAL or STACK, the default is LOCAL, unless the REENTRANT option is specified. If REENTRANT is specified, unqualified RECORD statements default to STACK data, rather than LOCAL data, for each instance of the function. For example, given the following statement:

```
function fred ,REENTRANT
```

all of the unqualified RECORD statements in function **fred** are treated as if they are STACK RECORD statements.

When a combination of LOCAL, STACK, or REENTRANT is used, the last one on the line determines how unqualified RECORD statements are processed.



The REENTRANT modifier causes all local records to be STACK records unless otherwise qualified. Fields in STACK records do not allow default or explicit initial values.

By default, a function cannot be re-entered. If you specify the REENTRANT option, however, the function can call itself.



Object handles are not allowed in local records in REENTRANT routines.

VARARGS is optional for unprototyped subroutines and functions, but it is required when using **-qnet** or strong prototypes (or internal routine prototyping unless **-qrelaxed:param** is set). It permits you to pass more arguments than declared.

WIN, UNIX

By default, a function may be removed from memory when it is no longer in the calling chain. If you specify the RESIDENT modifier, the function stays in memory.

A program by default rounds all expression results. You can change the default to truncate by setting system option #11. However, specifying the ROUND or TRUNCATE option on a FUNCTION statement overrides the default rounding behavior for that function (including if system option #11 is set). You cannot specify both TRUNCATE and ROUND in the same statement. See #11 in the “System Options” chapter of *Environment Variables and System Options* for more information about rounding versus truncation rules.

See also

[Chapter 7, “Understanding Routines.”](#)

Examples

In this example, the implied-decimal values within **avg** are processed using truncation rules; therefore, the intermediate result of num1 + num2 (7.5) is truncated to 7. The final result of 7/2 is truncated to 3.

However, if we specified ROUND instead of TRUNCATE (or if neither modifier is specified), the implied-decimal values would be processed using rounding rules. Thus, the intermediate result (7.5) would be rounded to 8 before being divided by 2. The final result would be 4.

```
.define TTCHN      ,1
.define MINVAL     ,3
external function   ;Declare the function and
    avg            ,i ; specify an integer result
record
    var1           ,d6.2,      3.4
    var2           ,d6.2,      4.1

proc
    xcall flags( 7000000, 1)
    open(TTCHN,i,"tt:")
    if ((var1 = %avg(var1,var2)).GT.MINVAL)
        writes(TTCHN, "Average = " + %string(var1))
endmain

function avg, i, TRUNCATE
    num1  ,d.
    num2  ,d.
record
    rslt  ,i4

proc
    rslt = num1 + num2
    freturn (rslt/2)
endfunction
```

GET – Receive data from a channel

`GET (channel, data_area, record) [[error_list]]`

Arguments

channel

The channel from which to get the data. The channel must already have been opened in input, output, or update mode, with no submode specified. **(n)**

VMS

If *channel* is open to a disk file, the file must have a stream record format.

data_area

A variable that will contain the input characters. The size of the data area determines how many characters are received. If the file contains too few characters, you'll get an "End of file" error (\$ERR_EOF). **(a)**

record

An expression defining which record is read. If the value is less than 1, you'll get an "Illegal record number specified" error (\$ERR_RECNUM). **(n)**

error_list

(optional) An I/O error list. If any one of the specified errors occurs during the GET, control is transferred to the associated label.

Discussion

The GET statement receives or reads data from the specified channel and places it in the designated data area. It performs random-access, binary input. It does not interpret the data being transmitted, and assumes that each character is an 8-bit binary value.

The record number is interpreted according to the size of the data area. The starting position is calculated as follows:

$$((record - 1) * (size\ of\ data\_area)) + 1$$

During GET processing, record terminators are not automatically attached or stripped. Your program can maintain complete control of the output file's contents and can process every character in the input file with the GET statement.

See also

["I/O error lists" on page 6-3.](#)

Examples

The examples below use the following data division.

```
record
  area1      ,a38
  area2      ,a1967
  area3      ,a5
```

The following example receives the 123rd 38-byte record from the specified channel. If less than 123 records (of 38 bytes each) are left in the file, the program terminates with an “End of file” error (\$ERR_EOF).

```
get (CHN, area1, 123)
```

The next example reads a 1967-byte record, whose record number is given in **recno**. If the record is beyond the end-of-file, control is transferred to the statement labeled **end_data**.

```
get (CHN, area2, recno) [eof=end_data]
```

GETS – Get sequential binary data

GETS (*channel*, *data_area*[, *label*][, WAIT : *wait_spec*][, MASK : *act_char*]) [[*error_list*]]

Arguments

channel

The channel from which to get the data. The channel must already have been opened in input or output mode or updated if it is a file. (**n**)

VMS

Channel must be opened to a file with stream record format or to a serial or terminal device.

data_area

A variable that will contain the data. The size of the variable determines how many characters are received. For example, a data area of 623 receives the next 623 characters. If the data area is a file and the file contains too few characters, the remainder of the file is transferred into the data area, but an “End of file” error (\$ERR_EOF) is generated. (**a**)

label

(optional) A label to which control is transferred if an “End of file” error (\$ERR_EOF) occurs.

WAIT

(optional) The time to wait for serial or terminal input. See [WAIT](#) on page 4-242 for a complete description.

MASK

(optional) Activation characters at the statement level. See [MASK](#) on page 4-225 for a complete description. (Synergy standard only)

error_list

(optional) An I/O error list. If any one of the specified errors occurs during the GETS, control is transferred to the associated label.

Discussion

The GETS statement reads sequential binary data from the specified channel to the designated data area. It does not interpret the data being transmitted, except that it terminates on activation characters. GETS assumes that each character is an 8-bit binary value.

During GETS processing, record terminators are not automatically attached or stripped. Your program can maintain complete control of the output file’s contents and can process every character in the input file with the GETS statement.

The MASK qualifier enables you to specify your own activation characters to replace those specified by the ACCHR, ACESC, DACHR, or DAESC subroutines if you have opened a serial or terminal device other than the console (TT:). The termination character is not returned in the data buffer (nor is the number of characters), but this character can be determined by calling %RTERM.

If you specify the WAIT qualifier with a value of zero, GETS reads all currently pending characters. If the WAIT value is nonzero, the runtime waits either for the field to be filled, an activation character to be found, or the time to elapse. If the time expires, the runtime generates a “Terminal input operation timeout” error (\$ERR_TIMEOUT), which you can trap. Use %RDLEN to return the number of characters read.

If the channel is opened to the console terminal (TT:), the GETS statement works much like the READS statement, with one exception. When a nonterminating character exceeds the receiving data area, the READS statement ignores the character and echoes an ASCII bell character to warn the user. The GETS statement, on the other hand, immediately stops the input and suppresses echoing of the typed character.



Avoid doing single-character GETS/ACCEPT sequences with the TTSTS subroutine (which slows down the system). A computer may be slowed noticeably by trying to accept characters at 9600 baud. Instead, use GETS with the WAIT and MASK qualifiers for timed I/O to a serial port. In this mode, several devices at 9600 baud can be handled simultaneously. For more information about serial ports on UNIX, see [INITPORT on page 9-160](#).

See also

- ▶ [“Input and Output Statement Qualifiers” on page 4-207.](#)
- ▶ [“I/O error lists” on page 6-3.](#)

Examples

The first two examples below use the following data division.

```
record
  area1      ,a38
  area2      ,a1967
  area3      ,a5
```

The following example receives the next 38 characters from the specified channel. It returns only when all 38 characters are received or when one of the characters matches an activation character. If fewer than 38 characters are left on the channel, the program terminates with an “End of file” error (\$ERR_EOF).

```
gets(CHN, area1)
```


The following example transmits the next 1967 characters from the specified channel. If fewer than 1967 characters remain, control transfers to the statement labeled **end_data**.

```
gets(CHN, area2) [eof=end_data]
```

In the next example, the GETS statement terminates if and when the buffer is full or a plus (+) character is read.

```
record
.align LONG
    actchars      , [32]i1
    .
    .
    .
clear actchars[]
actchars[2] = 2048 ;Activate on '+'
gets(chan, in_field, eof_label, WAIT:Q_WAIT, MASK:actchars)
```

GLOBAL-ENDGLOBAL – Define a global data section

```
GLOBAL[ DATA][ SECTION] name[ ,INIT]
.
.
.
ENDGLOBAL
```

Arguments

DATA

(optional) A keyword that you can use to make your global declaration clearer. Its presence does not affect the function of the GLOBAL statement.

SECTION

(optional) A keyword that you can use to make your global declaration clearer. Its presence does not affect the function of the GLOBAL statement.

name

The name of the global data area being referenced.

INIT

(optional) A keyword that enables you to declare the initial value of the global data area. Each global data section must have exactly one INIT option in the set of references to that data section. The module that contains the initializer defines which executable program or ELB “owns” that global data section.

Discussion

The GLOBAL statement defines the beginning of a global data section, and the ENDGLOBAL statement defines the end of a global data section.

Because a global data section is not directly associated with any specific routine, you can define it in any routine and then reference it from any other routine in your program. After you link your program, the size of the global data section matches the largest reference to that area in any of your program routines.

To define global records, use the unqualified RECORD statement after the GLOBAL statement. A GLOBAL statement block must end with the ENDGLOBAL statement. Any RECORD statement that appears between the GLOBAL and ENDGLOBAL statements defines a global record and its component fields.

Unlike COMMON, the names of the records and fields in a GLOBAL statement block in one subroutine do not have to match the names of the records and fields in the same GLOBAL statement in another subroutine. A GLOBAL statement in one subroutine is an overlay of the

GLOBAL statement in another subroutine by the same GLOBAL name. The data accessed by a field in a GLOBAL statement block is defined by the position of the field relative to the beginning of the GLOBAL statement and the locally specified size of the field.

Your global record declarations will look something like this:

```
global data section glb
  record rec1
    f1      ,a10
    f2      ,d5
  record rec2
    fa      ,d3
    fb      ,d4.2
endglobal
```

Global data sections can be scratch areas: their contents may be undefined upon entry to a subroutine unless the INIT option is specified. When INIT is present, the RECORD statements within the GLOBAL-ENDGLOBAL block can declare initial values. If the global data section in more than one routine uses the INIT option, an error is generated when you link your program.

If you have several definitions of a global data section and you specify the INIT option on one of the smaller layouts, only the size of the locally defined data is initialized. The rest of the global data section remains undefined.

If the INIT option is not present, Synergy Language ignores any initial values specified in the global RECORD definitions, except to determine the size of an automatically sized field designated with a size of *. The data in that global data section remains undefined.

Within any given routine, only one GLOBAL statement can reference a particular global data section. A global data section cannot have the same name as a common record or field that gets a global definition.



Object handles are not allowed in global data sections. Also, a global data section cannot be declared within a class.

Examples

The example below defines a global data section named **gbl**.

```
subroutine aaa
  arg      ,a
global data section gbl, init
record xyz
  fld1     ,a3, "abc"
  fld2     ,a4, "defg"
  fld3     ,a5, "hijkl"
endglobal
```

```
proc
    arg = fld1                ;arg = "abc"
    arg = fld2                ;arg = "defg"
    arg = fld3                ;arg = "hijkl"
    arg = gbl                 ;arg = "abcdefghijkl"
    xreturn
endsubroutine
```

;Subroutine bbb shows that the order of the fields has changed; thus the
;data accessed by the fields named the same is different based on the
;offset from the beginning of the GLOBAL section.

;

;The xyz record still accesses the same data as defined in subroutine aaa
;even though the order of the fields locally has changed, since the name
;is at the same position relative to the beginning of the GLOBAL DATA
;SECTION name and the size is still the same.

```
subroutine bbb
    arg      ,a
global data section gbl
record xyz
    fld3     ,a5
    fld1     ,a3
    fld2     ,a4
endglobal
```

```
proc
    arg = fld1                ;arg = "fgh"
    arg = fld2                ;arg = "ijkl"
    arg = fld3                ;arg = "abcde"
    arg = xyz                 ;arg = "abcdefghijkl"
    xreturn
endsubroutine
```

;Subroutine ccc shows that even though the order of the fields is the
;same, the data accessed is different, since the fields are different
;sizes from what was specified when the fields were defined.

;

;The beginning of the data accessed is defined by the offset from the
;beginning of the GLOBAL section. The size of the data accessed is
;defined locally by the size of the local field.

;

;The xyz record still accesses the same data as it is defined in
;subroutine aaa even though the size of the fields locally has changed,
;since the name is at the same position relative to the beginning of the
;GLOBAL DATA SECTION name, and the size is still the same, as the sum of
;the fields is also the same.

```
subroutine ccc
    arg      ,a
global data section gbl
record xyz
    fld1     ,a6
    fld2     ,a2
    fld3     ,a4
endglobal

proc
    arg = fld1
    arg = fld2
    arg = fld3
    arg = xyz
    xreturn
endsubroutine

;arg = "abcdef"
;arg = "gh"
;arg = "ijkl"
;arg = "abcdefghijkl"
```

GOTO – Transfer program control

GOTO *label*

or

GOTO (*label*[, ...]), *selector*

Arguments

label

A single statement label or a list of one or more statement labels that exist in the program. The first label is element 1, the second is element 2, and so forth.

selector

An expression whose value selects the corresponding element from the list of labels. (**n**)

Discussion

The GOTO statement transfers execution control to a specified label. The unconditional form of this statement specifies a single label. The computed form of this statement specifies a list of labels and a selection expression whose value indicates the desired label.

In the computed form of the GOTO statement, if *selector* is between 1 and the number of elements in *label*, the corresponding element is selected and control immediately transfers to the to the named label. However, if the value is less than 1 or more than the number of elements in *label*, no element is selected and execution continues with the statement following the GOTO.

Examples

This subroutine takes a result value to decide which code section to process. If the routine **decide** returns a result value of 1, control is transferred to **lbl1**. A value of 2, 3, or 4 transfers control to labels **lbl2**, **lbl3**, or **lbl4**, respectively. Any other value generates the message "... invalid result."

```
subroutine choice
    a_data    ,a
record
    result    ,d2

proc
loop,
    xcall decide(a_data, result)
    goto (lbl1, lbl2, lbl3, lbl4), result
    writes(TTCHN, "... invalid result")
    goto loop

lbl1,
    writes(TTCHN, "Processing result #1")
    goto loop
```

```

lbl2,
    writes(TTCHN, "Processing result #2")
    goto loop

lbl3,
    writes(TTCHN, "Processing result #3")
    goto loop

lbl4,
    xreturn
endsubroutine

subroutine decide
    a_data    ,a
    result    ,d

proc
    display(TTCHN, "Result value? ")
    reads(TTCHN, %a(result))
    return
endsubroutine

```



We recommend that you not use GOTO when the logic can be written using structured statements instead. For example, we could rewrite the above code as follows:

```

subroutine choice
    a_data    ,a
    external function
        decide    ,d

    proc
        repeat
            begin
                using %decide select
                begincase
                    (1),
                        writes(TTCHN, "Processing result #1")
                    (2),
                        writes(TTCHN, "Processing result #2")
                    (3),
                        writes(TTCHN, "Processing result #3")
                    (4),
                        xreturn
                writes(TTCHN, "... invalid result")
                endusing
            end
        endsubroutine

```

GROUP-ENDGROUP – Define a group

```
[access] GROUP name, [ [dimension, ... ] ][type][size] [,X]
    member_def
.
.
.
ENDGROUP
```



The brackets around *dimension* are required if more than one *dimension* argument is specified.

Arguments

access

(optional) One of the following access modifiers for a group within a class:

PUBLIC	Access is not restricted. This is the most accessible option.
PROTECTED	Access is limited to the containing class or types derived from the containing class.
PRIVATE	Access is limited to the containing type. This is the least accessible option. (default)

name

The name of the group to define.

dimension

(optional) A dimension of the group, which implies that the group is an array. If you specify more than one dimension, they must be enclosed in square brackets and separated by commas.

type

(optional) The data type of *name* which must be one of the following:

a	Alpha (default)
d	Decimal or implied-decimal
i	Integer

size

(optional) The maximum total size of all elements in the group. If *type* is implied-decimal, *size* has the form *size.places*, where *size* is the size of the entire field and *places* is the number of digits to the right of the decimal point. *Size* cannot be greater than 28. *Places* cannot be greater than 28 or greater than the value of *size*.

,X

(optional) Indicates that you're defining an overlay for the last nonoverlay field or group.

member_def

Either the definition of a field within the group or another group declaration. Each field definition has the syntax described in “[Field definitions](#)” on [page 2-20](#). You can specify as many member definitions as you want, although you must specify at least one.

Discussion

The GROUP statement defines a data structure within a record or another group structure. The ENDGROUP statement indicates the end of a group declaration.

You can only specify access modifiers (PUBLIC, PROTECTED, or PRIVATE) on a group within a class.

A group name must be unique among all other field and group names within a parent group. If a group isn't part of another group, its name must be unique within the record. For example, the following group declaration is not valid, because group **xyz** and field **xyz** have the same name:

```
record abc
    xyz                ,a5
    group xyz          ,a
    ijk                ,a10
    lmn                ,a4
endgroup
```

The following declaration is also invalid, because two groups in the same record with the same immediate parent have the same name.

```
record fred
    group joe          ,[12]a
    ed                 ,d5
    leroy              ,a2
endgroup
    group joe          ,d
    bruno              ,d12
    al                 ,d4
endgroup
```

However, the example in the Examples section below *is* valid, even though it declares two groups called **address**, because the two **address** groups don't belong to the same parent group. The first **address** belongs to the **office** group, while the second **address** belongs to the **contact** group.

If you declare a named group within a class record, the group and its members inherit the accessibility of the record, if specified. Otherwise, they default to private.

You can specify more than one *dimension*. The number of dimensions in the group is the number of *dimension* arguments that you specify. The number of elements in the dimensioned group is the product of all *dimension* arguments. If initial values are specified in any fields that are members of a group, each declared element in the dimensioned group is initialized to the same initial values. Initial values are not allowed in overlay records.

If you specify a group *size*, the size of the fields and/or groups within that group cannot exceed the specified size. If they do exceed *size*, an error is generated. For example, the compiler generates an error for the following group because the size of **fld1** (2000) is greater than the size specified for **grp** (10):

```
record rec
  group grp          , [15]a10
    fld1             , a2000
  endgroup
```

All valid syntax for a group (multidimensioned fields, subgroups, dimensioned subgroups, and so forth) is allowed for subroutine and function arguments, including dimensioning of the group itself. For example, a subroutine header can be declared as follows:

```
subroutine mysub
  arg1              , a
  group arg2        , [*]a
    fld1            , d5
    fld2            , [8]a3
  endgroup
  arg3              , a
```

Arg2[1].fld1 references the first five characters of the currently passed second argument, **arg2[1].fld2[3]** references characters 12 through 14, and so forth.



Object handles are not allowed within an argument group. For example, you cannot declare an argument with a data type of **@class** within a group.

See also

- ▶ [“Variable path specifications” on page 1-24](#) for information about referencing group elements.
- ▶ [“Parameter definitions” on page 2-33](#) for the syntax for declaring subroutine and function arguments.

Examples

The example below defines the **customer** group, which has one dimension and 100 elements. Members of the **customer** group are the **name** field, the **office** group, and the **contact** group. The **office** group has two members, the **bldg** field and the first **address** group. The **contact** group has two members, the **name** field and the second **address** group. Each **address** group has two members, the **street** and **zip** fields. Note again that Synergy Language allows the two groups called **address** because they don't belong to the same parent group.

```
record info
  group customer      , [100]a
    name              , a30
    group office      , a
      bldg            , a20
      group address   , a
        street        , a40
        zip           , d10
      endgroup
    endgroup
  group contact       , a
    name              , a40
    group address     , a
      street          , a40
      zip             , d10
    endgroup
  endgroup
endgroup
```

IF – Conditionally execute a statement

```
IF condition statement_1
or
IF condition THEN statement_2
ELSE statement_3
```

Arguments

condition

An expression whose value controls whether the statement is to be executed.

statement_1

A single or compound statement to be executed if *condition* is true.

statement_2

A single or compound statement to be executed if *condition* is true.

statement_3

A single or compound statement to be executed if *condition* is false.

Discussion

The first form of the IF statement conditionally executes a specified statement.

The second form of the IF statement executes one of two statements based on the value of an expression.

In the first form of the IF statement, if *condition* is true, *statement_1* is executed; if *condition* is false, *statement_1* is skipped. In the second form of the IF statement, if *condition* is true, *statement_2* is executed and *statement_3* is skipped; if *condition* is false, *statement_3* is executed and *statement_2* is skipped. In all cases, execution continues with the statement following the IF.

Statement_1, *statement_2*, *statement_3* and the THEN and ELSE keywords can each begin on the next logical line.

By default, Synergy Language requires that an IF statement with a THEN present must also have an ELSE, and the ELSE belongs to the last THEN in the same lexical level.



You can perform IF tests of the following two forms:

```
if .NOT. dfield
```

```
if dfield
```

Examples

Below is an example of the first form of the IF statement.

```
clear dflag
display(TTCHN, "Do you want extended screens?")
reads(TTCHN, yesno)
if (yesno .eq. "Y")
    xcall ext_scrns
```

Below is an example of the IF-THEN-ELSE form of the IF statement. In this example, **statement1** is processed when **answer** is equal to Y and **choice** equals 1, and **statement3** is processed when **answer** is not equal to Y.

```
if (answer .eq. 'Y') then
    if (choice .eq. 1) then
        statement1
    else
        statement2
else
    statement3
```

The following is an example of a cascaded IF statement, which is a series of IF-THEN-ELSE statements that have another IF-THEN-ELSE statement following the ELSE keyword:

```
if (a_select .eq. 1) then
    result = a_v1 + a_v2
else if (a_select .eq. 2) then
    result = a_v1 - a_v2
else if (a_select .eq. 3) then
    result = a_v1 * a_v2
else if (a_select .eq. 4)
    result = a_v1 / a_v2
```

Although the cascaded IF statement is functionally equivalent to the CASE statement, CASE is more efficient and easier to maintain. See [CASE on page 4-18](#) for more information.



To avoid logic errors in nested IF statements, do one of the following:

- ▶ Always use the default Synergy Language format.
 - ▶ Enclose the embedded IF statement in a BEGIN-END compound statement.
-

IF-THEN-ELSE – Conditionally execute one of two statements (alternate)

```
IF condition[ THEN] statement_1  
[ELSE statement_2]
```

Arguments

condition

An expression whose value controls which statement is executed.

statement_1

A single or compound statement to be executed if *condition* is true.

statement_2

(optional) A single or compound statement to be executed if *condition* is false.

Discussion

The IF-THEN-ELSE statement (alternate form) executes one of two statements based on the value of an expression.

The result of *condition* is evaluated for its truth value. If it's true, *statement_1* is executed and *statement_2* is skipped; if it's false, *statement_1* is skipped and *statement_2* is executed (if present).

Statement_1, *statement_2*, and the THEN and ELSE keywords can each begin on the next logical line.

By default, Synergy Language requires that an IF statement with a THEN present must also have an ELSE, and the ELSE belongs to the last THEN in the same lexical level. However, if you specify either the alternate IF compiler option or system option #31, Synergy Language recognizes an alternate, non-ANS DIBOL form of the IF statement. In the alternate form of the IF statement, the THEN is optional and the ELSE matches the last IF statement, not the last THEN.

See also

- ▶ [“Compiler options”](#) in the “Building and Running Synergy Language Programs” chapter of *Synergy Language Tools* for more information about the alternate IF compiler option.
- ▶ [#31](#) in the “System Options” chapter of *Environment Variables and System Options*.

Examples

In the following example, if system option #31 or the alternate IF compiler option is set, **statement2** is processed when **answer** is equal to Y and **choice** is not equal to 1.

```
if (answer .eq. 'Y') then
  if (choice .eq. 1)
    statement1
  else
    statement2
```

IMPORT – Import prototypes

```
IMPORT identifier [DIRECTORY 'location' ]
```

Arguments

identifier

A namespace (or nested namespace) name.

location

The path of the directory from which prototypes will be imported.

Discussion

The IMPORT statement imports the prototypes in the specified namespace to enable you to use any members of that namespace. Routine and class prototypes are created by the **dblproto** utility. (See “[The Synergy Prototype Utility](#)” in the “General Utilities” chapter of *Synergy Language Tools* for more information about **dblproto**.)

The IMPORT statement cannot occur inside of a namespace declaration. We highly recommend that you put an IMPORT statement at the beginning of every source file in which you intend to use a prototype or imported symbol, before any other declarations, including structure declarations.



An implicit main—that is, a PROC statement or a data division construct such as RECORD without the MAIN keyword—cannot immediately follow an IMPORT statement (although it can occur elsewhere in the program or source file). If a main routine immediately follows an IMPORT statement, it must have the keyword MAIN.

If *identifier* is a nested namespace (for example, MyNamespaces.Subnamespace), only the nested namespace is imported (and not the parent namespace).

If you specify “DIRECTORY *location*”, that directory will be used as the import directory. The directory specification can include logicals. If you don’t specify an import directory, the compiler searches for the prototype in the following locations:

- ▶ The directories specified by the **-qimmdir** compiler option, in order of appearance on the command line
- ▶ The directory/directories specified by the SYNIMPDIR logical, in order of appearance within the logical
- ▶ The current directory



If you have multiple imports of the same namespace from different directories, the first IMPORT statement is used, and subsequent imports of that namespace are ignored.

When the compiler encounters an `IMPORT` statement, all of the prototypes in the directory belonging to the namespace are included in the compile. This includes classes, structures, subroutines, and functions.

When looking for prototypes in the import directory, the compiler searches first for prototypes of classes in that namespace and then for prototype files with the same base name. For example, if

```
import MyNS
```

is specified, the compiler first checks for class prototypes in the `MyNS` namespace (files named **MyNS-*.dbp**) and then looks for a prototype file of the name **MyNS.dbp**.

A nested namespace is not imported automatically when its parent namespace is imported. Each nested namespace requires its own `IMPORT` statement.

If you import a prototype into the same file that contains the declaration for a class or routine, the compiler will ensure that the prototype matches the declaration.



The compiler always imports prototypes from the default namespace specified by `SYNDEFNS` if it is defined, or from `synglobal` if `SYNDEFNS` is not defined, regardless of whether the `IMPORT` statement is used.

Examples

The statement below imports prototypes in the `Namespace1` namespace, which the compiler looks for in the directory pointed to by the `MYDIR` logical.

```
import Namespace1 directory 'MYDIR:'
```

INCR – Increment a numeric variable

INCR *variable*

Arguments

variable

A variable whose value is to be incremented by 1. (**n**)

Discussion

The INCR statement increases the value of a specified variable by 1.

If the incremented value of *variable* overflows its size, *variable* is set to its zero representation. For example, the following code fragment increments **dvar** from 9 to 0, because **dvar** is a **d1** field instead of a **d2**.

```
record
    dvar    ,d1    ,9
proc
    incr dvar
```



It is more efficient to use the INCR statement than to add 1 to a variable.



With integer values, if the variable has the maximum value for its size, the following statements has different results:

```
incr ivar
ivar = ivar + 1
```

In the example below, **i1 = i1 + 1** is equivalent to **i1 = 128** which results in **-128**. See [“Moving integer data to an integer destination” on page 3-38](#) for more information.

```
record
    i1      ,i1,  127          ;Max value for an i1 field

proc
    incr i1                    ;Result is 0
    i1 = 127
    i1 = i1 + 1                ;Result is -128
```

The following statement “rolls” the value to zero. Doing so maintains consistency with the behavior of decimal variables, which may be used as counters and may depend on this property.

```
incr i1
```

See also

[DECR on page 4-35.](#)

Examples

```
record
  ifld      ,i2      ,20
  dfld      ,d4      ,20
  idfld     ,d4.2    ,20.00
```

```
proc                                ;Incremented result is:
  incr ifld                          ;21
  incr dfld                          ;21
  incr idfld                         ;21.00
```

INIT – Initialize a named data structure to its default values

```
INIT data_structure [, ...]
```

Arguments

data_structure

A named record, group, structfield, or individual field. (**a**)

Discussion

The INIT statement initializes a named data structure to its declared default values, if provided, or to the defaults for the specified types, if declared default values are not provided.

When an array field is initialized, all elements in the array are initialized.

The INIT statement can only be used on nonarrayed structfield arguments. If INIT is applied to any other argument type, an error will be generated.

The only type of routine parameter that is allowed with an INIT statement is structfield.

Examples

The INIT statement below initializes **field1** to NULL, **field2** to 28, **field3** to “Initial value”, and **field4** to 0.

```
record myrecord
    field1    ,i4
    field2    ,d3    ,28
    field3    ,a20    ,"Initial value"
    field4    ,d8.2
```

```
proc
.
.
.
init myrecord
.
.
.
```

LITERAL-ENDLITERAL – Define a literal

```
[form] LITERAL [name]
    member_def
    .
    .
    .
[ENDLITERAL]
```

Arguments

form

(optional) Defines the literal as one of the following forms:

GLOBAL	Causes the statement to allocate data space in a shared data area.
EXTERNAL	Causes the statement to reference data space that was previously allocated by a GLOBAL LITERAL statement.
LOCAL	Causes the statement to allocate space in local literal space. (default)

name

(optional) The name of the literal to define.

member_def

Either the definition of a field or a group declaration. Each field definition has the syntax that's described in [“Field definitions” on page 2-20](#). Each group declaration has the syntax described in [GROUP-ENDGROUP on page 4-86](#). You can specify as many member definitions as you want.

Discussion

The LITERAL statement defines a local read-only data structure that cannot be modified during program execution. The ENDLITERAL statement indicates the end of a literal declaration.

A literal can be declared within a routine. A literal can either be GLOBAL, which allocates the data space, or EXTERNAL, which references it. Fields in an EXTERNAL LITERAL reference the data in the GLOBAL LITERAL by matching the field name. If you don't specify GLOBAL or EXTERNAL, the literals are placed into local literal space.

If you specify *name*, the literal is accessed as data type alpha. If you don't specify *name*, the literal is unnamed. Each literal name must be unique within the routine in which it is declared.

In an unnamed literal, data space associated with a given field is not created until it is first referenced. If a literal is named, all of the literal's contents are created unconditionally. If you define a group within an unnamed literal, the group's contents are created unconditionally regardless of whether they are referenced.

If the size of an integer global literal is not specified, Synergy Language defaults the size to **i4**. For example, in the following definition, **lit** is created as an **i4** literal:

```
global literal
    lit          ,i,          2
```

You cannot specify automatic size (*) for an external literal.

You can only specify a field position indicator for a literal field if the literal record is named or the field exists in a GROUP-ENDGROUP block. For example, the following is invalid:

```
literal                                ;Invalid literal declaration
    lit1          ,i4
    lit2          ,i4  @lit1
```

A literal record can overlay another literal record if the record being overlaid is named.

You cannot specify a packed data field within the scope of a literal field.

Examples

```
literal headers
    title ,2a16,      "Monthly Income", "Monthly Expenses"
    comp  ,a*,        "XYZ Corporation"
    pg    ,a*,        "Page  "
```

LOCASE – Convert uppercase characters to lowercase

LOCASE *variable*

Arguments

variable

A variable containing the characters to be converted to lowercase. (**a**)

Discussion

The LOCASE statement converts uppercase alphabetic characters to their corresponding lowercase characters. It does not change the value of any other character.



The LOCALIZE subroutine controls the effect of the LOCASE statement on the characters [, \,], and ^ by enabling ANSI-compliant translation. See [LOCALIZE on page 9-206](#) for more information.

On those systems that support a user setting of the current language locale (Windows, OpenVMS, and most UNIX), the runtime automatically determines which 8-bit characters are uppercase or lowercase according to the current language locale of the user.

See also

[UPCASE on page 4-191.](#)

Examples

```
record
    fld1      ,a14,  "JUST A [DEMO]."
```



```
proc                                ;The results will be:
    locase fld1(4,11)                ;"JUSt a [demO]."
```

```
    locase fld1                      ;"just a [demo]."
```

```
    stop
```

```
end
```

LPQUE – Queue a file for printing by the system

```
LPQUE (filename[, LPNUM:device][, COPIES:number][, FORM:form_name][, ALIGN]
&      [, DELETE[:flag]])
```

Arguments

filename

The name of the file to be queued for output. The filename must conform to the form and restrictions imposed by the spooler device. (a)

LPNUM

(optional) Indicates the spooled device to which the file is directed for output.

device

The print device. Possible values depend on the spooler being used.

COPIES

(optional) Indicates the number of copies to print.

number

The number of copies you want to print. Possible values depend on the spooler being used. (n)



The FORM and ALIGN qualifiers do not affect printing on Windows unless system option #22 is set. See the [Migrating Your Application to Windows](#) document, available on the Online Manuals CD, for more information about option #22 and LPQUE.

FORM

(optional) Indicates a special form name. If you don't specify FORM, the default is no special forms.

form_name

The name of the form you want. The format of the possible form names depends on the spooler being used.

ALIGN

WIN, UNIX

(optional) ALIGN indicates that a spooler-defined alignment pattern is to be created and accepted by the system operator before the file contents are printed. This alignment depends on the spooler being used.

VMS

ALIGN tells the system spooler to place the entry on hold.

DELETE

(optional) Indicates whether the file is to be deleted after its contents have been printed.

flag

(optional) A value indicating whether the file will be deleted after it is printed. If *flag* is a nonzero value, the file *won't* be deleted. If DELETE is specified but *flag* is not specified or is zero, the file *will* be deleted. By default, if DELETE is not specified, the file *won't* be deleted. (**n**)

Discussion

The LPQUE statement queues a file for printing by the system. This statement relies closely on the spooling facilities offered by each operating system. Not all systems provide support for each of the LPQUE options.

When specifying a filename, the default extension is **.ddf**. To print a file with no extension, append a period (.) to the end of the filename. For example:

```
lpque ("filename.")
```

WIN

The LPNUM value must be an alpha variable specifying the printer device in the form “*name*, [*device*]”, unless system option #22 is set.

When using LPQUE with *x/Server*, we recommend that the client generate a print file locally and then print it on a printer accessible to the client. We do not recommend that the client print a remote file using *x/Server*; however, if you choose to, the printer must be attached directly to the server machine. Network printers are not supported from *x/Server* on Windows.

If you don't specify the COPIES qualifier or if *number* is less than 1, one copy is printed.

See also

- ▶ The [Professional Series Portability Guide](#) for more information about system option #22 and other operating system-specific implementation details of the LPQUE statement.
- ▶ The [Migrating Your Application to Windows](#) document, available on the Online Manuals CD.

Examples

The following example sends the file **test.1st** to the print queue, prints three copies, and then deletes the file.

```
lpque("test.1st", COPIES:3, DELETE)
```

The following example sends the file, which is named in the variable `file`, to the printer to print aligned, and with the **labels** form. The file is deleted after printing.

```
lpque(file, FORM:"1-part", ALIGN, DELETE)
```

MAIN-ENDMAIN – Explicitly define a main routine

```
MAIN[ routine_name][, option]
.
.
.
[ENDMAIN]
```

Arguments

routine_name

(optional) The name of the main routine. **(a)**

option

(optional) One of the following modifiers:

ROUND	Use rounding rules for implied-decimal data types within the routine.
TRUNCATE	Use truncation rules for implied-decimal data types within the routine.

Discussion

The MAIN statement is recognized with or without a beginning period. You can specify only one MAIN statement within a program.

Routine_name will appear in compiler listing headers and runtime error messages. MAIN\$ always precedes the resulting routine name, regardless of whether or not MAIN\$ was specified as part of *routine_name*. The routine name is automatically uppercased.

The *routine_name* field is limited to 30 characters (or 31 on OpenVMS). If more characters are specified, an “Identifier too long” error (BIGIDEN) occurs. Those 30 characters may include MAIN\$ in the first 5 characters followed by up to 25 more characters. If the *routine_name* field does not contain MAIN\$ in the first 5 characters but is between 25 and 30 characters long, it is truncated without error to 25 characters.

If *routine_name* is omitted, the default routine name is MAIN\$ followed by the first 25 characters of the program’s source filename. (If the **-t** compiler option is being used, the default is MAIN\$ followed by the first 6 characters of the source filename.)

The default message ID for the SEND and RECV statements is *routine_name* stripped of MAIN\$. A message ID can have a maximum of 6 characters on Windows and UNIX or 39 characters on OpenVMS.

Here are some examples:

MAIN statement	Routine name	Default message ID	
		Windows/UNIX	OpenVMS
main avg	MAIN\$AVG	AVG	AVG
main myaddressbookroutine	MAIN\$MYADDRESSBOOK ROUTINE	MYADDR	MYADDRESS BOOKROUTINE
main main\$contact	MAIN\$CONTACT	CONTAC	CONTACT
main	MAIN\$source, where source is the first 25 characters of the program's source filename	The first 6 characters of the program's source filename	The first 39 characters of the program's source filename

A program by default rounds all expression results. You can change the default to truncate by setting system option #11. However, specifying the ROUND or TRUNCATE modifier on a MAIN statement overrides the default rounding behavior for that routine (including if system option #11 is set). You cannot specify both TRUNCATE and ROUND in the same directive. See #11 in the “System Options” chapter of *Environment Variables and System Options* for more information about rounding versus truncation rules.

Examples

In this example, **orders** is the name of the main routine. Since TRUNCATE is specified, **var** is assigned 3. If ROUND were specified (or if no modifier were specified), **var** would be assigned 4.

```
main orders, TRUNCATE
record
    var ,d3

proc
    .
    .
    .
    var = 3.5
    .
    .
    .
endmain
```

MERGE – Combine two or more files

```
MERGE ( INPUT= (filename_1, filename_2[, ...]) [, RECORD=record[ / rec_switch]]
&      [, KEY= (key_field[ / key_switch, ...]) ][, OUTPUT=out_file][, SKIP= ([skip_rec, ...]) ]
&      [, END=[end_var, ...]][, OPTIONS=option_string])
```

Arguments

INPUT

Precedes two or more input files to merge. You can specify at least two and up to seven files.

filename_1

The first (primary) input file to merge. **(a)**

filename_2

The second input file to merge. **(a)**

RECORD

(optional) Precedes the record that defines the file record format.

record

The name of the record that defines file record format. If *record* is not specified, the RECSIZE qualifier must be included in the OPTIONS qualifier string. **(a)**

rec_switch

(optional) One of the following switches that identifies the record as either fixed-length or variable-length. Each switch must be preceded by its own slash (/). **(a)**

f Fixed-length (default)

v Variable-length

KEY

(optional) Precedes one or more fields within a record to use as a key. If KEY is not specified, the KEY qualifier must be specified in the OPTIONS string. You can specify up to 15 keys, separated by commas. The order in which the keys are specified defines the order of comparison in the MERGE.

key_field

A named field within *record* to use as a key. **(a)**

key_switch

(optional) One or more of the following key switches that indicates the order of the sorted records. Each switch must be preceded by its own slash (/). (**a**)

- f** Forward (ascending) sorted order (default)
- r** Reverse (descending) sorted order
- a** Alpha key (no case distinction)
- d** Decimal key (numeric ordering)
- c** Character key (case-sensitive ordering)

OUTPUT

(optional) Precedes an output filename. If the OUTPUT qualifier is not specified, the MERGE is assumed to be an in-place reverse merge into the first (primary) input file listed (only two files can be used in this case).

out_file

The name of the output file. If *out_file* is specified, an *n*-way merge is performed, where *n* is the number of files specified. (**a**)

WIN, UNIX

SKIP

(optional) Indicates the number of records to skip at the beginning of the input file. Each number in the list, in the order specified, corresponds to an input file. If a skip count is not specified for each file, the previous value in the list is carried over.

skip_rec

(optional) A list of one or more records to skip at the beginning of each input file. If *skip_rec* is not specified, the MERGE begins with the file's first record. (**n**)

END

(optional) Specifies a list of alpha or numeric expressions to use as either maximum record counts or logical end-of-file characters for the input files.

end_var

(optional) One or more variables or literals. Each element in the list, in the order specified, corresponds to an input file. If an element is not specified for a file, the previous one is carried over. If *end_var* is not specified, the files are merged to the physical end-of-file. (**a** or **n**)

OPTIONS

(optional) Specifies one or more MERGE options to be parsed at runtime.

option_string

Has a maximum of 256 characters and the following format: **(a)**

```
[RECSIZE=size[/rec_swch]][, KEY=(key_spec) ][, OUTPUT=out_file]
&    [, SKIP=(skip_rec[, ...]) ][, END=(end_var[, ...]) ]
```

RECSIZE

(optional) Precedes the size of the records in the file. This qualifier must be specified if the RECORD compile-time qualifier is not specified.

size

A literal that states the size of the records in a file. For files containing variable-length records, use the maximum record length. **(n)**

rec_swch

(optional) One of the following switches. Each switch must be preceded by its own slash (/). **(a)**

- f** *Size* is the size of a fixed-length record. (default)
- v** *Size* is the maximum size of a variable-length record.
- e** *Size* is the exact size of a fixed-length record, including record terminators.

KEY

(optional) Specifies MERGE keys. You can specify up to 15 keys in *key_spec*. This qualifier must be used if the KEY compile-time qualifier is not specified.

key_spec

A key specification in the following form: **(a)**

pos:len[/swch, ...]

pos The position of the key field within the record **(n)**

len The length of the given key **(n)**

swch (optional) Any one of the switches listed above under *key_switch*. Each switch must be preceded by its own slash (/). **(a)**

OUTPUT

(optional) Precedes an output filename (*out_file*). This qualifier is functionally similar to the OUTPUT compile-time qualifier, except that only literal values are accepted.

WIN, UNIX

SKIP

(optional) Precedes a list of the number of records to skip (*skip_file*) at the beginning of each input file. This qualifier is functionally similar to the SKIP compile-time qualifier, except that only literal values are accepted.

END

(optional) Precedes a list of alpha or numeric literals (*end_var*) to use as either maximum record counts or logical end-of-file characters for the input files. This qualifier is functionally similar to the END compile-time qualifier, except that only literal values are accepted.

Discussion

The MERGE statement combines two or more ordered files to create one ordered output file.

The merged output file always supersedes any existing file. If the output file already exists, any existing contents are lost. All files used by the MERGE are opened and closed internally.



The file structures and record definitions for each input file must be identical, and the files must be sorted in the same order.

If the files are not sorted, no error or warning occurs; the output file MERGEs unsorted. A file containing variable-length records cannot be merged in place.

If the logical end-of-file is specified but not found, no error or warning occurs; the file is processed until the physical end-of-file is reached.

The OPTIONS qualifier overrides the corresponding MERGE qualifier. It enables you to build MERGE commands during program execution. This causes some minor extra runtime overhead, so we recommend that you use compile-time arguments wherever possible. The maximum size of the OPTIONS string is 256.

The */e* switch is allowed within the OPTIONS string, but not within the RECORD option. Also, if both record options are used in the MERGE statement, the runtime specification is used. The */e* switch is designed for files with nonstandard record terminators; however, it may restrict portability between some operating systems if record terminators vary in length. The */f* or */v* switch should be used with standard files.

You must specify at least one key, whether it be in the KEY compile-time qualifier or within the OPTIONS string.

By default, an in-place MERGE treats the logical end-of-file character or record as the end of usable data. Any records following the logical end-of-file are overwritten during the in-place MERGE.

If you want to have any records that follow the logical end-of-file moved to the end of the extended primary file, set system option #18.

By default, an in-place MERGE places duplicate records from the secondary file before their matching records from the primary file. Set system option #23 to place duplicate records from the secondary file after their matching record in the primary file.

See also

- ▶ [#18](#) in the “System Options” chapter of *Environment Variables and System Options*.
- ▶ [#23](#) in the “System Options” chapter of *Environment Variables and System Options*.

Examples

The following example assumes that two fixed-length files containing address lists sorted by zip code are to be merged.

```
merge(INPUT=(list1, list2),
      &   RECORD=customer,
      &   KEY=zip,
      &   OUTPUT=newlist,
      &   END= ("[" , var))
```

If **newlist** were not specified, **list1** would be the default output file (in-place MERGE). If **customer** were a variable-length record, however, the in-place MERGE would not be allowed. For the first input file, the “[” character is to be interpreted as a logical end-of-file character if it is found as the first character in a record within the file. **Var** is treated differently depending on whether it is a decimal or alpha variable. If **var** is decimal, it is interpreted as a maximum record count. If **var** is alpha, the first character is used as a logical end-of-file character.

The following example is a five-way MERGE using the runtime option string for the record and key information.

```
merge(INPUT=(f1, f2, f3, f4, f5),
      &   SKIP=( , 2, , 2)
      &   END= (" / " , , 12, "?" )
      &   OPTIONS="RECSIZE=40,KEY=(5:10/f/c, 23:2/a) ")
```

The MERGE doesn’t skip records in the first, third, or fourth input files, but it skips two records in the second and the fifth. For the first and second input files, the “/” character is to be interpreted as the logical end-of-file character. The third file has a maximum record count of 12, so only the first 12 records of the file are merged. The fourth and fifth files have the “?” character as a logical end-of-file.

METHOD-ENDMETHOD – Define a method

```
[access] [method_mod ...] METHOD name, return_type[, options]
  parameter_def
  [constructor_init ([arguments])]
.
.
.
PROC
.
.    ;Method implementation
.
ENDMETHOD | END
```

Arguments

access

(optional) One of the following access modifiers:

PUBLIC	Access is not restricted. This is the most accessible option.
PROTECTED	Access is limited to the containing class or types derived from the containing class.
PRIVATE	Access is limited to the containing type. This is the least accessible option. (default)

method_mod

(optional) One or more of the following modifiers, with the restrictions stated in the Discussion section:

ABSTRACT	Must have a method implemented in a derived class of the same signature. Subclasses must implement this method.
SEALED	Cannot be overridden any further by inheriting classes without the NEW keyword.
NEW	Hide an inherited member of the same name.
OVERRIDE	Overrides the parent class’s virtual method that has the same method signature.
STATIC	Accessed without a reference object. The method can be executed even if the class that contains the method hasn’t been instantiated.
UNIQUE	Cannot be overloaded, overridden, or redeclared.

VARARGS More than the declared number of arguments can be passed to this routine.

VIRTUAL Can be overridden in a subclass.

name

The name of the method to define.

return_type

The return type of the method being defined. See [“Where Data Types Can Be Used” on page 2-18](#) for the data types that are valid return types.

options

(optional) One of the following options:

ROUND Use rounding rules for implied-decimal data types within the function.

TRUNCATE Use truncation rules for implied-decimal data types within the function.

parameter_def

The definition of an parameter that is unique within this method. Each parameter definition has the syntax that’s described in [“Parameter definitions” on page 2-33](#).

constructor_init

(optional) One of the following initializers:

parent Run the parent class constructor next higher in the class hierarchy.

this Run the current class constructor of a different signature.

arguments

(optional) One or more argument values to the constructor initializer. This value must be a literal, a STATIC field value, a CONST field value, or an argument name from the constructor.

Discussion

By default, a method cannot be overridden unless it is marked as ABSTRACT, OVERRIDE, or VIRTUAL. (ABSTRACT and OVERRIDE methods are also assumed to be virtual and are mutually exclusive.) A method defined with one of these modifiers is called a virtual method. An ABSTRACT method has no implementation, which means it has a PROC and an END with no intervening statements. The method in the derived class that is overriding the method in the parent class must also be marked as OVERRIDE.

The SEALED modifier can only be specified in conjunction with the OVERRIDE modifier.

You can hide a class member in a parent class with an identical signature in an inherited class by specifying a member with the same name, along with the **NEW** member modifier, within the inheriting class. You can also hide dissimilar members. (For example, a field or a sealed member in the parent can be hidden by a method or field of the same name marked **NEW** in the child.)

The class members can be different kinds (for example, a method and a field).

The **VARARGS** modifier is required if you want to permit passing additional, nondeclared arguments. We recommend that you don't use **VARARGS** on methods.

In the table below, an X indicates that the two modifiers can be used in the same method definition

Modifier	ABSTRACT	SEALED	NEW	OVERRIDE	STATIC	UNIQUE	VARARGS	VIRTUAL
ABSTRACT			X			X	X	
SEALED				X			X	
NEW	X				X	X	X	X
OVERRIDE		X					X	
STATIC			X			X	X	
UNIQUE	X		X		X		X	
VARARGS	X	X	X	X	X	X		X
VIRTUAL			X				X	
PRIVATE		X	X	X	X	X	X	
PROTECTED	X	X	X	X	X	X	X	X
PUBLIC	X	X	X	X	X	X	X	X

All defined methods are reentrant, and method data is stack based.

A program by default rounds all expression results. You can change the default to truncate by setting system option #11. However, specifying the **ROUND** or **TRUNCATE** option on a **METHOD** statement overrides the default rounding behavior for that function (including if system option #11 is set). You cannot specify both **TRUNCATE** and **ROUND** in the same statement. See #11 in the “System Options” chapter of *Environment Variables and System Options* for more information about rounding versus truncation rules.

The method name (*name*) cannot be any of the following reserved keywords: **new**, **parent**, or **value** (for a property set method). Also be careful when using any other Synergy keywords, such as the names of I/O statements; if the method signature coincidentally matches that of the statement with the same name, the statement will be assumed.

Declaring constructor and destructor methods

You can optionally declare one or more constructor methods for a class by giving the constructor methods the same name as the class. Arguments are allowed on constructor methods to make the method signatures of the constructors unique.

You can also declare a single destructor for a class by giving the method the same name as the class with a tilde (~) appended to the beginning. See [“Constructors and destructors” on page 8-10](#) for more information.

Method implementation

You can return from a method that has a declared return type using the MRETURN statement. (See [MRETURN on page 4-116](#) for more information.)

See also

[“Accessibility” on page 8-16.](#)

Examples

In the example below, a method called **writeit** is defined.

```
public method writeit, void
    c2, @class2
proc
    open(1,o,"tt:")
    writes(1,"num="+%string(c2.m_number))
    close(1)
    mreturn
endmethod
```

MRETURN – Return control from a method

`MRETURN` [*expression*]

Arguments

expression

(optional) An expression whose type is the same as the return type of the method.

Discussion

If the current method is a constructor or destructor, or if its return type is VOID, *expression* should not be specified. Otherwise, *expression* should be present and should have a representation that is the declared return type for the method.

In VOID methods (including constructors and destructors), an MRETURN is implied at the end of the method. In non-VOID methods, you must specify at least one MRETURN statement; otherwise a compiler error will be generated.

Example

In the example below, the MRETURN statement returns control from the get method. The MRETURN for the set method is implied.

```
public class myclass
  private myfield, i4
  public property myproperty, i4
    method get
      proc
        mreturn myfield
      endmethod
    method set
      proc
        myfield = value
      endmethod
    endproperty
  endclass
```

The MRETURN in the example below does not specify an expression because the method's return type is VOID.

```
public method mymethod, void
  p1, i4
  proc
    p1 = 0
    mreturn
  endmethod
```

NAMESPACE-ENDNAMESPACE – Define a namespace

```
NAMESPACE name
  member_def
  .
  .
  .
ENDNAMESPACE
```

Arguments

name

The name of the namespace to define.

member_def

Either the declaration of a class, structure, function, or subroutine within the namespace, or another namespace declaration. You can specify as many member definitions as you want, although you must specify at least one.

Discussion

The NAMESPACE-ENDNAMESPACE statement creates a namespace, which enables you to group related entities together for use in a Synergy application.

Name can consist of multiple nested namespace names, separated by a period. For example,

```
namespace space1.space2
```

A namespace name must be unique among all other identifiers within the same namespace.

Examples

The example below creates the namespace **space1**, which defines the classes **class1** and **class2**.

```
namespace space1
  class class1
    .
    .
    .
  endclass
  class class2
    .
    .
    .
  endclass
endnamespace
```

NEXTLOOP – Terminate current loop iteration

NEXTLOOP

Discussion

The NEXTLOOP statement terminates the current iteration of a loop statement (such as DO, FOR, REPEAT, and WHILE). Execution continues with the next iteration of the current loop statement.

Examples

The following example demonstrates the difference between the NEXTLOOP, EXITLOOP, and EXIT statements.

```
for i from 1 thru 100
  begin
    if (.not. %valid(data(i)))
      begin
        if ((severity = %fixit(data(i))) .eq. FIXED)
          exit                      ;Process the data
        if (severity .eq. FATAL)
          exitloop                  ;Get out of for loop
        nextloop                    ;Start next iteration
      end
    xcall process(data(i))
  end
```


NOP – Perform no operation

NOP

Discussion

The NOP statement performs no operation and is ignored by the program. It is often used with the CASE and USING statements.

Examples

The following example uses the NOP statement where a routine that hasn't been developed yet will be placed later.

```
record
    choice          ,a2

proc
    xcall getchoice(choice)
    case choice of
        begincase
"ADD":
    xcall add_cust
"DEL":
    xcall del_cust
"LST":
    xcall list_cust
"RPT":
    nop                ;Report function not implemented yet
"UPD":
    xcall upd_cust
        endcase
    stop
end
```

OFFERROR – Disable runtime errors traps

OFFERROR

Discussion

The OFFERROR statement cancels any outstanding error traps established by a previous ONERROR statement. If you use OFFERROR, all runtime errors are subsequently considered fatal and cause immediate program termination until another ONERROR statement is executed. The exceptions are errors trapped by the I/O statement error lists and EOF labels.

See also

[“Record Locking” on page 4-243.](#)

Examples

```
onerror oops                ;Enable "bad digit" trap
number = input
offerror                    ;Disable "bad digit" trap
```

ONERROR – Enable trapping of runtime errors

ONERROR *label*

or

ONERROR (*error_list*) *label*[, (...) ...][, *catch_all*]

Arguments

label

In the unqualified form, a single statement label to which control is immediately transferred if a trappable runtime error occurs. In the qualified form, *label* is one or more statement labels to which control is transferred if any error in the associated list occurs.

error_list

A list of errors separated by commas that cause execution to be transferred to the associated label. Each list is made up of one or more compile-time expressions, which can be error numbers, error literals, or literal variables, as long as the expression can be evaluated to a number by the compiler. (**n**)

catch_all

(optional) A statement label to which control is transferred if an error that is not one of the listed errors occurs.

Discussion

The ONERROR statement sets one or more error traps to detect and handle runtime errors. The unqualified form of the statement specifies a single label. The qualified form of the ONERROR statement sets several error traps, each defined by a list of error numbers being trapped and the label to which control is transferred.

When an ONERROR statement is executed, all error traps created by any previously executed ONERROR statements are canceled. Only those traps created by the current ONERROR statement are defined. The defined error traps remain in effect until one of the following conditions occurs:

- ▶ An OFFERROR statement is executed.
- ▶ A new ONERROR statement is executed, causing the current traps to be canceled and new traps to be set.
- ▶ An external subroutine or user-defined function is called, which suspends the current error traps until the subroutine or function returns. Upon returning, the error traps are re-established.
- ▶ The program terminates.



If an ONERROR loop occurs during I/O to a server, aborting the runtime with CTRL+C or one of the Windows functions may take a long time, making the abort appear not to work.

Error catching

If no ONERROR statement or error list for an individual statement exists, a fatal traceback is generated by the runtime. However, if you are using a product (such as *x/ServerPlus*) that must be able to control all exceptions so it can provide information to the caller, you'll want to include the \$ERR_CATCH literal in your error list to enable catch functionality. Error catching enables a routine to set an ONERROR trap that is called whenever an untrapped error occurs in a routine lower in the call tree. The routine must use the %ERR_TRACEBACK function to retrieve (or clear, if not required) the error traceback information that would have been reported had the error not been trapped. If %ERR_TRACEBACK is not used, additional caught errors continue appending information to the internal catch buffers until memory is exhausted. See the second example in the Examples section below.

See also

- ▶ [“Record Locking” on page 4-243.](#)
- ▶ [%ERR_TRACEBACK on page 9-109.](#)

Examples

In this example, the OFFERROR statement ensures that the error traps are canceled at the **bad** and **what** labels. If an error occurs, the program terminates.

```
.define TTCHN          ,1
record
    alpha              ,a10
    decml              ,d5
    errmsg             ,a40

proc
    open(TTCHN, i, "tt:")
    repeat
        begin
            display(TTCHN, "Enter number (1-7 digits): ")
            reads(TTCHN, alpha)[eof=done]
            onerror ($ERR_DIGIT) bad, what
            decml = alpha
            offerror
            writes(TTCHN, "... It was valid")
            nextloop

    bad,
        offerror
        writes(TTCHN, "... It was not valid")
        nextloop

    what,
        offerror
        errmsg = %ernum, "Error XXX: "
```

```

        xcall ertxt(%ernum, errmsg(12,40))
        writes(TTCHN, errmsg)
    end

done,
    stop
end

```

The following example illustrates using the \$ERR_CATCH mnemonic to enable the catching of errors in called routines that do not have an ONERROR statement.

```

.align
record
    tty          ,i4
    sts          ,i4
record
    error_description      ,a255

proc
    onerror ($ERR_CATCH) caught
    xcall routine_to_call
    stop

caught,
    offerror
    open(tty=%syn_freechn, o, 'tt:')
    writes(tty,"error "+%string(%ernum)+" was caught")
    while (%err_traceback(error_description))
        writes(tty, %atrim(error_description))
    stop
end

subroutine routine_to_call
proc
    xcall cause_an_error
    xreturn
endsubroutine

subroutine cause_an_error
record
    d          ,i2
    zero       ,i2

proc
    d = d/zero
    xreturn
endsubroutine

```

OPEN – Open a channel

```
OPEN (channel, mode_spec, file_spec[, ALLOC : blocks][, LOCK : lock_spec][, NUMREC : number]
&    [, OPTIONS : "runtime_option"[, ...]][, POSITION : pos_spec][, RECSIZ : record_size]
&    [, SHARE : share_spec][, TEMPFILE : temp_spec][, FDL : string][, GUIWND]
&    [, BKTSIZ : bucket_size][, BLKSIZ : block_size][, BUFNUM : number][, BUFSIZ : blocks]
&    [, CONTIG : value][, DEQ : blocks][, RECTYPE : record_format]) [ [error_list] ]
```

Arguments

channel

The channel to be opened. Valid channel numbers are 1 through 1024. **(n)**

mode_spec

One of the following mode specifications:

mode[:sub] Compile-time mode

MODE:var Mode and submode are determined at runtime.

mode

The mode in which to open the channel: **(a)**

- I** Input mode. Allows read access of an existing file.
- O** Output mode. Creates a new file opened for exclusive access. Files opened in O mode on OpenVMS default to sequential submode. To change the default to stream, use the STREAM qualifier.
- U** Update mode. Allows read and write access of an existing file and locks the record when read.
- A** Append mode, which functions like output mode, except that if the file already exists, it positions to the end of the file.

sub

(optional) One of the following submodes in which to open the file: **(a)**

- S** Sequential submode. Specifies sequential access of the file.
- R** Relative submode. Specifies relative access of the file.
- I** Indexed submode. Indicates that the file is an ISAM file.
- B** Block submode. Allows direct block access to files. (See the Discussion section below for more information.)

- P** Print submodule. Used with output mode only. Creates a file for output to a printer-like device.
- C** Character submodule. Indicates that character I/O is performed on the specified channel (used for terminal I/O).

VMS

- * Wildcard. Opens the file as if the appropriate submodule for the file's actual organization had been specified (S for sequential, R for relative, or I for indexed). This submodule cannot be used with output mode (O), and using the DELETE statement is not allowed on a file opened in U:* mode.
-

var

An expression that contains the mode and optional submodule in the form *mode[:sub]* from above (for example, "i:i"). (**a**)

file_spec

The name of the file or device to be opened. The default filename extension is **.ddf**. For ISAM files, the default file extension is **.ism**. See ["File specifications" on page 1-43](#) for more information about valid file specifications. (**a**)

ALLOC

(optional) Defines the initial allocation for a file being opened for output. See [ALLOC on page 4-208](#) for a complete description.

LOCK

(optional) Specifies whether record locking is performed on the specified channel. See [LOCK on page 4-223](#) for a complete description.

NUMREC

(optional) Preallocates a number of RECSIZ records for the file being opened in output mode. The RECSIZ qualifier must also be specified. See [NUMREC on page 4-230](#) for a complete description.

OPTIONS

(optional) Specifies one or more runtime options. See [OPTIONS on page 4-231](#) for a complete description.

WIN, UNIX**POSITION**

(optional) Defines the position at which the first sequential I/O operation begins. See [POSITION on page 4-233](#) for a complete description.

RECSIZ

(optional) Sets the record size for the file being opened. With this option you can specify a record size, or you can set the record size to the length of the first record in the file. See [RECSIZ on page 4-234](#) for a complete description.

SHARE

(optional) Specifies whether the file can be accessed by other users. On an OPEN statement, you can specify the SHARE qualifier with a file open for input and/or with a sequential file type. See [SHARE on page 4-239](#) for a complete description.

TEMPFILE

(optional) Defines an intermediate file to be used when opening in output mode (as in versions prior to Synergy DBL 5). See [“Temporary files” on page 4-133](#) for more information.

FDL

(optional) Specifies an FDL or XDL string or file description. See [FDL on page 4-216](#) for a complete description.

GUIWND

(optional) Opens a new Synergy application window through which all I/O for the channel is directed. See [GUIWND on page 4-221](#) for a complete description.

VMS**BKTSIZ**

(optional) Specifies the RMS bucket size to be used on the OPEN statement. See [BKTSIZ on page 4-209](#) for a complete description.

BLKSIZ

(optional) Specifies the RMS block size in bytes. See [BLKSIZ on page 4-210](#) for a complete description.

BUFNUM

(optional) Sets the number of process local I/O buffers. See [BUFNUM on page 4-211](#) for a complete description.

BUFSIZ

(optional) Sets the buffer size. See [BUFSIZ on page 4-212](#) for a complete description.

CONTIG

(optional) Allocates the contiguity of a file. See [CONTIG on page 4-213](#) for a complete description.

DEQ

(optional) Sets the number of blocks to add when a file is automatically extended. See [DEQ on page 4-214](#) for a complete description.

RECTYPE

(optional) Defines the type of records to write to a file. See [RECTYPE on page 4-236](#) for a complete description.

error_list

(optional) An I/O error list. If any one of the specified errors occurs as part of the OPEN, control is transferred to the associated label.

Discussion

The OPEN statement opens a file or device (for example, a terminal, port, mailbox device, etc.) on a specified channel and defines the type of input and output allowed.

If a qualifier is specified that has no function under a specific system, it is ignored and no error occurs.

If the OPEN is successful, all subsequent I/O operations are able to reference the opened channel. If an error other than a “Channel is in use” error (\$ERR_CHNUSE) occurs during the OPEN, the specified channel is not assigned and is still available for a subsequent OPEN.

If an **.is2** file is detected, an OPEN on ISAM files fails with a “Failure during I/O operation” error (\$ERR_IOFAIL) from Synergy Language and a DE_OPNIDX error from the Synergy Database API.

Modes and submodes

When you specify an I/O mode, *mode* determines how a program accesses a file, and *submode* defines how a program processes a file and, in some cases, the organization of the file being accessed. The submode primarily enables you to detail the use of the file. If a submode is present on a file opened in output mode, the created file has the characteristics specified by *submode*. If a submode is present on a file opened in any mode other than output mode, the file’s characteristics are checked.

If you open a channel without specifying a submode, the default is stream submode on all systems except OpenVMS. On OpenVMS, if no submode is specified on file creation, the default is sequential (which means you can’t use GET and PUT statements on the file); if no submode is specified on existing file opens, the submode is determined by the file type.

When block (B) submode is specified, only I/O requests that are a multiple of the block or sector size of the operating environment are allowed on READ, READS, and WRITE statements. However, if you are opening to a device, the device—rather than the operating environment—may determine the block size. All operations in block submode start on a block size boundary.

The WRITES statement does *not* require that the buffer be a multiple of the block size. Sequential operations following a WRITES operation with a buffer that is not a multiple of block size start on the next block size boundary. The area between partial write and the next block is undefined.

Only certain combinations of I/O modes and submodes are allowed on certain I/O statements. The following table lists each of the I/O statements and their valid modes and submodes.

X = Allowed – = Not allowed V = OpenVMS only WU = Windows, UNIX * = Must add permission on OpenVMS

Submode and mode	ACCEPT	DELETE	DISPLAY	FIND	FORMS	GET	GETS	PUT	PUTS	READ	READ FST	READ LST	READS	REVERSE	STORE	WRITE	WRITE FST	WRITE LST EOF	WRITES
Stream I	X	–	–	X	–	X	X	–	–	X	X	X	X	–	–	–	–	–	–
Sequential stream_CR I	–	–	–	V	–	–	–	–	–	V	V	–	V	–	–	–	–	–	–
Sequential stream_LF I	–	–	–	V	–	–	–	–	–	V	V	–	V	–	–	–	–	–	–
Stream O	–	–	X	X	X	X	X	X	X	X	–	–	X	–	–	X	–	–	X
Sequential stream_CR O	–	–	V	–	–	–	–	–	–	–	–	–	V	–	–	–	–	–	V
Sequential stream_LF O	–	–	V	–	–	–	–	–	–	–	–	–	V	–	–	–	–	–	V
Stream U	–	–	–	X	–	X	X	X	X	X	–	–	X	–	–	X	–	–	X
Sequential stream_CR U ^a	–	–	–	V	V	–	–	–	–	V	–	–	V	–	–	V	–	–	V
Sequential stream_LF U ^a	–	–	–	V	V	–	–	–	–	V	–	–	V	–	–	V	–	–	V
Sequential I	–	–	–	X	–	–	–	–	–	X	X	WU	X	–	–	–	–	–	–
Sequential O	–	–	X	–	X	–	–	–	–	–	X	WU	X	–	–	–	–	–	X
Sequential U	–	–	–	V	V	–	–	–	–	V	V	–	V	–	–	V	–	–	V
Relative I	–	–	–	X	–	–	–	–	–	X	X	WU	X	X*	–	–	–	–	–
Relative O	–	–	–	X	–	–	–	–	–	X	X	WU	X	X*	–	X	X	WU	X
Relative U	–	X	–	X	–	–	–	–	–	X	X	WU	X	X*	–	X	X	WU	X
ISAM I	–	–	–	X	–	–	–	–	–	X	X	X*	X	X	–	–	–	–	–
ISAM O	–	X	–	X	–	–	–	–	–	X	X	X*	X	X	X	X	–	–	–
ISAM U	–	X ^b	–	X	–	–	–	–	–	X	X	X*	X	X	X	X	–	–	–

Submode and mode	ACCEPT	DELETE	DISPLAY	FIND	FORMS	GET	GETS	PUT	PUTS	READ	READ FST	READ LST	READS	REVERSE	STORE	WRITE	WRITE FST	WRITE LST EOF	WRITES
Block I	-	-	-	X	-	-	-	-	-	X	-	-	X	-	-	-	-	-	-
Block O	-	-	-	X	-	-	-	-	-	X	-	-	X	-	-	X	-	-	X
Block U	-	-	-	X	-	-	-	-	-	X	-	-	X	-	-	X	-	-	X
Null device I	V	-	V	-	V	-	V	-	V	-	-	-	V	-	-	-	-	-	V
Null device O	V	-	V	-	V	-	V	-	V	-	-	-	V	-	-	-	-	-	V
Null device U	V	-	V	-	V	-	V	-	V	-	-	-	-	-	-	-	-	-	-
Mailbox I	-	-	-	-	-	-	-	-	-	-	-	-	V	-	-	-	-	-	-
Mailbox O	-	-	-	-	-	-	-	-	-	-	-	-	V	-	-	-	-	-	V
Mailbox U	-	-	-	-	-	-	-	-	-	-	-	-	V	-	-	-	-	-	V
Record-oriented device I	-	-	-	X	-	-	-	-	-	X	-	-	V	-	-	-	-	-	-
Record-oriented device O	-	-	X	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X
Record-oriented device U	-	-	-	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X
Spooled network device O:P	-	-	V	-	V	-	-	-	-	-	-	-	V	-	-	-	-	-	V
Spooled network device U	-	-	-	-	V	-	-	-	-	-	-	-	V	-	-	-	-	-	V
Printer O:P	V	-	X	-	X	-	V	-	V	-	-	-	V	-	-	-	-	-	X
Print O	-	-	X	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X
Print U	-	-	X	-	X	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Terminal I	X	-	X	-	X	-	X	-	X	-	-	-	X	-	-	-	-	-	X
Terminal O	X	-	X	-	X	-	X	-	X	-	-	-	X	-	-	-	-	-	X
Terminal U	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Pipe I	WU	-	-	-	-	-	WU	-	-	-	-	-	X	-	-	-	-	-	-
Pipe O	-	-	WU	-	WU	-	-	-	WU	-	-	-	-	-	-	-	-	-	X
Pipe U	-	-	-	-	-	-	-	-	-	-	-	-	V	-	-	-	-	-	V

- Due to unpredictable results and potential side effects of the RMS implementation, we do not recommend or support opening a stream_CR or stream_LF file in update mode.
- Note that you cannot use U:* with DELETE on OpenVMS. If you want to delete records, you must open the file with an explicit submode (e.g., U:I).

If an invalid mode is specified on the OPEN statement, you'll get a "Bad mode specified" error (\$ERR_IOMODE).

An OPEN for output mode supersedes existing files, unless the FLAGS subroutine runtime option 3 is set. (See [FLAGS on page 9-126](#) for more information.) To create an ISAM file, you must specify the FDL qualifier with either an XDL file, FDL file, or XDL or FDL string.

Input mode (I:I) sets up input-only access, and restricts the channel to the use of the CLOSE, FIND, READ, and READS statements. Update mode (U:I) declares that the channel can be used for input as well as deletion and update activity. In U:I mode, all the ISAM I/O statements are available for use.

If the LOCK qualifier is not specified, subsequent I/O operations use automatic locks. In input mode, this means that no record locking is performed on the data file (**.is1**), but tree locking is performed on the index file (**.ism**) to maintain the integrity of the ISAM file's index. In update mode, this means that both record locking and tree locking are performed. Whenever locking (record or file) is used, when you open a file in I:I mode, neither the index nor data file require write permission. With U:I mode, both the index and data files need write permission.

If you specify the LOCK:Q_NO_LOCK qualifier to change the default locking state, no record or tree locking is performed in both input and update mode. The LOCK:Q_NO_LOCK qualifier overrides all I/O record locking on an ISAM file.



We recommend that you use the LOCK:Q_NO_LOCK qualifier on the OPEN statement with caution; if other processes are performing locks on the same file, use of this qualifier could cause file corruption. Use the SHARE qualifier to gain exclusive access.

An OPEN using append (A) or output (O) mode includes an implied SHARE mode of Q_EXCL_RW to gain exclusive access to the file.

Opening a terminal

A terminal device (TT:) can be opened in either input or output mode, and with or without the C submode. If a terminal is opened in input mode, output can also be performed, and vice versa.

VMS

When you open TT:, Synergy Language works in the following manner:

- ▶ If neither system option #35 nor #39 is set, Synergy Language opens SYS\$INPUT and SYS\$OUTPUT. If you're running your program from a command file or batch job, DCL maps SYS\$INPUT to SYS\$COMMAND, so that input comes from the command file, not the terminal.
- ▶ If system option #35 or #39 is set, Synergy Language opens the logical TT:. If you're running your Synergy Language program from a command file, the logical TT: is defined by DCL as the terminal, so Synergy Language opens the terminal. If you're running your program from a

batch job, a detached job, or a network process (any process for which the TNMBR subroutine returns a negative value, which currently includes just these three types), Synergy Language works as if system options #35 and #39 are not set. See the preceding paragraph.

If you want to get your input from the terminal when you're running from a command file, either set system option #35 or #39, or define SYSS\$INPUT as TT:. For example:

```
$ define/user sys$input tt:
```

Defining SYSS\$INPUT as TT: only works for the program that is started immediately following this definition, and the definition does not remain set across program chains.

Opening a port

WIN

You must include a colon at the end of the port name passed as the *file_spec* argument. For example, to open the COM1 port, your OPEN statement would look something like this:

```
open(portnum, 'o', 'com1:')
```

If the colon isn't present, "com1" is interpreted as a filename rather than a device and won't be found.

UNIX

Do not include a colon at the end of the port name. If the port name ends in a colon, it's interpreted as an environment variable. Also, you must pass both the path and the device name. For example:

```
open(portnum, 'o', '/dev/tty03')
```

Open pipes

WIN, UNIX

You can optionally execute a command through an opened pipe using the following syntax:

```
OPEN(channel, mode, command)
```

channel

The channel to be opened (1–1024). (**n**)

mode

Either I for input mode or O for output mode, with no submode specified. *Mode* determines the direction of the pipe, which might be output from the Synergy Language program to a subprocess, or input to the Synergy Language program from a subprocess. (**a**)

command

A vertical bar (|) followed by a command to be executed (up to 255 characters). *Command* shares one end of the pipe with the Synergy Language program executing the OPEN. (a)

You can get the status of the command after a CLOSE statement using the XSTAT subroutine. See [XSTAT on page 9-394](#) for more information.

On Windows, you cannot use this method for shell commands, such as **dir**. For shell commands, you must execute the program (**cmd**) and pass the command (**dir**). For example:

```
open(2,i,"|cmd /c dir")
```

VMS

You can execute a command or a series of commands through an opened pipe using this syntax:

```
OPEN(channel, mode, command)
```

channel

The channel to be opened (1–1024). (n)

mode

Either I for input mode, O for output mode, or U for a two-way pipe, with no submode specified. *Mode* determines the direction of the pipe, which might be output from the Synergy Language program to a subprocess, or input to the Synergy Language program from a subprocess, or both. (a)

command

A vertical bar (|) followed by a command to be executed (up to 255 characters). *Command* shares one end of the pipe with the Synergy Language program executing the OPEN. (a)

You can get the status of the last command executed after a CLOSE statement using the XSTAT subroutine.

Using the above open pipe syntax, an OpenVMS command can be written, output from the command can be read, and when end-of-file has been encountered on the READS, another command can be written. Your program can write ASCII(3) or ASCII(25) to stop the current program running on that pipe. (These are OpenVMS terminal driver special sequences, CTRL+C and CTRL+Y, that interrupt a program.) Sending ASCII(25) sends EOF to the pipe, which is the exit signal for many OpenVMS programs. You can do a READ from a pipe with a time-out that limits how long to wait for a command to produce output. This is useful when dealing with programs that can be run at a pipe, but go into loops or other infinite wait states.

VMS

Opening a mailbox device

If you open a mailbox device in O:P mode, the runtime generates an error.

WIN, UNIX

Temporary files

If you specify the TEMPFILE qualifier when opening a file in output mode and a file with the same name already exists, a temporary file is created. The temporary file either replaces the existing file on a CLOSE, or it is deleted if no CLOSE is performed or if a PURGE is executed. If you open an existing file in output mode *without* specifying the TEMPFILE qualifier, no temporary file is created and the existing file is deleted on the OPEN. See [TEMPFILE on page 4-241](#) for syntax.

Maximum file size

The maximum file size for each operating system is as follows:

Operating system	File type	Maximum size per physical file
Windows	FAT16	2,147,483,647 bytes
	TBYTE ISAM file on NTFS	1,099,511,627,775 bytes
	Non-TBYTE ISAM file on NTFS ^a	2,147,483,647 bytes
UNIX	TBYTE ISAM file	1,099,511,627,775 bytes or file size ulimit (whichever is smaller)
	Non-TBYTE ISAM file or non-ISAM file ^a	2,147,483,647 bytes or file size ulimit (whichever is smaller)
OpenVMS	RMS OpenVMS 6.0 or higher	1,099,511,627,775 bytes

a. On 64-bit versions of Synergy/DE, the TBYTE limit applies to non-ISAM files as well.

UNIX

Ensure the operating system setting for the file size ulimit is sufficient for the largest file size expected. This is especially important when using Synergy 7 and terabyte ISAM files. Versions prior to 6.4 automatically extended the file size ulimit to 204800 or 104 MB. Version 6.4 (and higher) does not automatically extend the ulimit file size; instead, it generates an “Output file is full” error (\$ERR_FILFUL) when a file reaches the system-imposed file size ulimit.

See also

- ▶ [“Input and Output Statement Qualifiers” on page 4-207.](#)
- ▶ [“I/O error lists” on page 6-3.](#)
- ▶ [FLAGS on page 9-126.](#)
- ▶ [XSTAT on page 9-394.](#)

Examples

The first example opens the terminal in output mode.

```
open(chn, o:c, "tt:")
```

The second example opens the file **DAT:master.ddf** for exclusive read and write access. The file is opened in relative submode with a record size of 200 characters.

```
open(chn, u:r, "DAT:master", SHARE:Q_EXCL_RW, RECSIZ:200)
```

The third example opens a pipe and executes the UNIX or OpenVMS DATE command to display the date and time. The program then reads the output of that command.

```
main
.define pipe_ch      ,12

record
.ifdef OS_VMS
    command    ,a*      , "| SHOW TIME"
.else
    command    ,a*      , "| date"
endc
    line      ,a80

proc
    open(pipe_ch, i, command)
    reads(pipe_ch, line)
    close pipe_ch
endmain
```

The last example opens an ISAM file for update mode. If the file does not exist, ISAMC is called to create it.

```
    open(ISMCHN, u:i, "intro.ism") [err=nofile]
    if (%false)
        begin

nofile,
    xcall isamc("intro.ism", 38, 1, "start=1, length=20, dups")
    open(ISMCHN, u:i, "intro.ism") [err=nofile]
end
```


PROC-END – Begin and end the procedure division

```
PROC
```

```
  .  
  .  
  .
```

```
END
```

Discussion

The PROC statement ends the data division section of a program and begins the procedure division. You must use one PROC statement per routine.

The END statement ends the procedure division. If the procedure division is part of a main routine, subroutine, function, or method definition, you can alternatively end it with ENDMAIN, ENDSUBROUTINE, ENDFUNCTION, or ENDMETHOD, respectively.

The PROC and END statements are recognized with or without a beginning period.

Examples

```
record                                ;Data division  
  dvar    ,d6  
  avar    ,a6  
  
proc                                    ;Beginning of the procedure division  
  .  
  .  
  .  
end                                    ;End of the procedure division
```

PROPERTY-ENDPROPERTY – Declaring a property

```
[access] [property_mod ...] PROPERTY name, property_type
    method_def
.
.
.
ENDPROPERTY
```

Arguments

access

(optional) One of the following access modifiers:

PUBLIC	Access is not restricted. This is the most accessible option.
PROTECTED	Access is limited to the containing class or types derived from the containing class.
PRIVATE	Access is limited to the containing type. This is the least accessible option. (default)

property_mod

(optional) One or more of the following property modifiers, with the restrictions stated in the Discussion section:

ABSTRACT	Must have a method implemented in a derived class of the same signature. Subclasses must implement this method.
SEALED	Cannot be overridden any further by inheriting classes without the NEW modifier.
NEW	Hide an inherited member of the same name.
OVERRIDE	Overrides the parent class's virtual method that has the same method signature.
STATIC	Accessed without a reference object. The method can be executed even if the class that contains the method hasn't been instantiated.
VIRTUAL	Can be overridden in a subclass.

name

The name of the property to declare.

property_type

Any valid field types. See [“Field definitions” on page 2-20](#). The only valid array syntax for a property type is [#]*@class*.

method_def

The definition of either a get or a set accessor method (METHOD GET or METHOD SET).
See [METHOD-ENDMETHOD on page 4-112](#) for more information about defining a method.

Discussion



Although a property appears to look like a field in the code, it isn't. The compiler translates it into a set of method calls. Because of this, property access is significantly slower than field access, and properties cannot be used in the same places where fields can. For example you can do this:

```
if (field = 20)
```

but you cannot do this:

```
if (property = 20)
```

The access modifier (PUBLIC, PROTECTED, PRIVATE, INTERNAL, or PROTECTED INTERNAL) can appear between the other modifiers; it does not have to come first. See [“Accessibility” on page 8-16](#) for more information about these settings.

By default, properties cannot be overridden unless they are marked as ABSTRACT, OVERRIDE, or VIRTUAL. (ABSTRACT and OVERRIDE properties are also assumed to be virtual and are mutually exclusive.) An ABSTRACT property has no implementation, which means it has a PROC and an END with no intervening statements.

The SEALED modifier can only be specified in conjunction with the OVERRIDE modifier.

If you have any questions about which modifiers can be specified together, see the METHOD table on [page 4-114](#).

You can hide a class member in a parent class with an identical signature in an inherited class by specifying a member with the same name, along with the NEW member modifier, within the inheriting class. You can also hide dissimilar members. (For example, a field or a sealed member in the parent can be hidden by a method or field of the same name marked NEW in the child.) The class members can be different kinds (for example, a method and a field).

You can optionally override the property accessibility by declaring an access modifier on the get or set accessor. The modifier on the get or set accessor must be more restrictive than the accessibility of the property itself.

The *method_def* must be a METHOD GET or METHOD SET definition to get the value of the property or set the value of the property, respectively. (See the Examples section for an example of each.) When you declare a set method for a property, an implicit **value** argument, which has the

same type as the property, represents the value to which the property should be set. (Because the keyword **value** is reserved, it cannot be used as an identifier within a set method.) Parameters are not valid on get or set methods.

To get an accessible property value from an instantiated class, use the following syntax, which causes the property's declared get method to be executed:

```
avalue = var1.myproperty
```

To set an accessible property value from an instantiated class, use the following syntax, which causes the property's set method to be executed:

```
var1.myproperty = avalue
```

Indexer properties

An indexer is a special property that allows data in a class to be accessed using array syntax. You can declare one or more indexer properties for a class by declaring a property with the name “indexer” followed by one or more array index arguments. (Note that you cannot specify arguments for nonindexer properties.) For example,

```
class myclass
private record
myarray, [100]i4 ; private field
endrecord
public property indexer, i4
    index1, i4
method get
proc
    mreturn myarray[index1]
endmethod
method set
proc
    myarray[index1] = value
    mreturn
endmethod
endproperty
endclass
```

The get accessor for an indexer has access to all the arguments of the indexer. The set accessor for an indexer has access to all those same arguments plus the implicit **value** argument.

See [“Properties and indexers” on page 8-8](#) for more information.

Examples

The example below declares a get accessor method for **myproperty** that returns the value of the property.

```
private record
    myfield, i4                ; a private field
endrecord
public property myproperty, i4
method get
proc
    mreturn myfield
endmethod
endproperty
```

The following example declares a set method that sets the value of **myproperty**.

```
private record
    myfield, i4                ;A private field
endrecord
public property myproperty, i4
method set
proc
    myfield = value
    mreturn
endmethod
endproperty
```

PURGE – Purge a channel

PURGE *channel*

Arguments

channel

The channel to purge. (**n**)

Discussion

The PURGE statement completes any pending I/O, terminates the association of a channel with a file, and deletes the file if it was opened in output mode.

When a channel is purged, all locks on the specified channel are released and the channel is available for future use. After a channel is purged, an I/O statement cannot reference that channel until an OPEN statement reactivates it.

If *channel* is a serial or terminal device, the runtime flushes the input and output buffers (thus canceling any pending output) and resumes any suspended output.



To guarantee that all data written to the device is transmitted, use the CLOSE statement.

If *channel* is opened in output mode, the PURGE statement deletes the output file. If *channel* is not opened in output mode, PURGE performs the same function as a CLOSE statement.



If you specify the TEMPFILE qualifier when opening a file in output mode, and a file with the same name already exists, the existing file is still present after a PURGE is executed. However, if you open an existing file in output mode *without* the TEMPFILE qualifier specified, the existing file is deleted on the OPEN.

See also

[TEMPFILE on page 4-241](#).

Examples

See [CLOSE on page 4-28](#) for an example of the PURGE statement.

PUT – Write fixed-length data

`PUT (channel, data_area, record) [[error_list]]`

Arguments

channel

The channel to which you want to send data. The channel must already have been opened in output or update mode with no submode specified. **(n)**

VMS

The file must have a stream record format.

data_area

An expression whose contents are to be written. The size of the result of the expression determines the number of characters transmitted. For example, an **a623** variable causes 623 characters to be written. **(a)**

record

An expression defining which record is written. If the value is less than 1, you'll get an "Illegal record number specified" error (\$ERR_RECNUM). **(n)**

error_list

(optional) An I/O error list. If any one of the specified errors occurs during the PUT, control is transferred to the associated label.

Discussion

The PUT statement performs random-access binary output. It does not interpret the data being transmitted. It assumes that each character is an 8-bit binary value.

The record number is interpreted according to the size of the data area. The position of the first character is calculated as follows:

$$((record - 1) * (size\ of\ data\_area)) + 1$$

During PUT processing, record terminators are not automatically attached or stripped.

See also

["I/O error lists" on page 6-3.](#)

Examples

The examples below use the following data division.

```
record
  area1          ,a38
  area2          ,a1967
  area3          ,a5
```

The following example sends the 157th five-byte record to the specified channel. The record starts at position 781 from the beginning of the file [$((157-1) * 5) + 1 = 781$].

```
put (CHN, area3, 157)
```

The next example sends the 1967-byte record whose number is defined by **recno** to the file opened on **CHN**. If a trappable error occurs, control immediately transfers to the statement labeled **lbl1**.

```
put (CHN, area2, recno) [err=lbl1]
```


PUTS – Write sequential fixed-length data

PUTS (*channel*, *data_area*[, *label*]) [[*error_list*]]

Arguments

channel

The channel to which you want to write the data. The channel must already have been opened in output or update mode with no submode specified. (**n**)

VMS

The channel must be open to a file that has a stream record format, or be opened to a serial or terminal device.

data_area

An expression whose contents are to be transmitted. The size of the result of the expression determines how many characters are output. For example, an **a623** variable sends the next 623 characters. (**a**)

label

(optional) A label to which control is transferred if an “End of file” error (\$ERR_EOF) occurs.

error_list

(optional) An I/O error list. If any one of the specified errors occurs during the PUTS, control is transferred to the associated label.

Discussion

The PUTS statement writes the specified data area’s next sequential fixed-length record to a channel. It does not interpret the data being transmitted. It assumes that each character is an 8-bit binary value.

During PUTS processing, record terminators are not automatically attached or stripped.

See also

[“I/O error lists” on page 6-3.](#)

Examples

The examples below use the following data division.

```
record
  area1          ,a38
  area2          ,a1967
  area3          ,a5
```

The following example sends the five bytes contained in **area3** to the specified channel. Since PUTS performs sequential access, the output begins at the current position in the file.

```
puts(CHN, area3)
```

The next example outputs a 1967-byte record to the file opened on **CHN**. If an error is trapped, control immediately transfers to the statement **lbl1**.

```
puts(CHN, area2) [err=lbl1]
```

READ – Read a specified record

```
READ (channel, data_area, key_spec[, GETRFA:new_rfa][, KEYNUM:krf_spec]
&      [, LOCK:lock_spec][, MATCH:match_spec][, NOFILL][, POSITION:pos_spec]
&      [, RFA:match_rfa][, WAIT:wait_spec]) [[error_list]]
```

Arguments

channel

The channel on which the file to be read is open. The channel must already have been opened in input or update mode. **(n)**

data_area

A variable that will receive the information. For ISAM files, this argument is used in implied key-of-reference specifications. **(a)**

key_spec

The ordinal position of the record in the file, the key value to match, or one of the following qualifiers:

^FIRST	Position to the first record in the file before reading. For ISAM files, this is relative to the key of reference.
^LAST	Position to the last record in the file before reading. For ISAM files, this is relative to the key of reference.



On Windows and UNIX, ^LAST works only on ISAM and relative files. On OpenVMS, ^LAST works only on ISAM files and only with version 6.1 and above of OpenVMS.

GETRFA

(optional) Returns the record's RFA. See [GETRFA on page 4-218](#) for a complete description.

KEYNUM

(optional) Specifies a key of reference by which to perform the READ. See [KEYNUM on page 4-222](#) for a complete description.

LOCK

(optional) Specifies whether the record is to be locked. See [LOCK on page 4-223](#) for a complete description.

MATCH

(optional) Defines how a specified key is matched. See [MATCH on page 4-227](#) for a complete description.

NOFILL

(optional) Prevents the data area from being padded with blanks when the data returned does not completely fill *data_area*. See [NOFILL on page 4-229](#) for a complete description.

POSITION

(optional) Specifies an absolute position in the file. See [POSITION on page 4-233](#) for a complete description.

RFA

(optional) Reads the record with the specified RFA. See [RFA on page 4-237](#) for a complete description.

WAIT

(optional) Specifies how long to wait for a record lock to be released. See [WAIT on page 4-242](#) for a complete description.

error_list

(optional) An I/O error list. If any one of the specified errors occurs as part of the READ, control is transferred to the associated label.

Discussion

The READ statement retrieves a specified record from a file.

By default, READ locks the record, unlocking any previous automatic lock. If *channel* is open in update mode and READ tries to access a locked record, a “Record is locked” error (\$ERR_LOCKED) is generated. If the program does not trap the error, it is considered fatal.

If the file record size is less than that of the data area, the record is placed into the data area left-justified over blanks. If the file record size is greater than the size of the data area, only the leftmost portion of the record is transferred, and an “Input data size exceeds destination size” error (\$ERR_TOOBIG) is generated. (If system option #27 is set, this error is not generated. See #27 in the “System Options” chapter of *Environment Variables and System Options* for more information about this option, which allows a larger buffer to be stored into a smaller destination.)

If you READ the last record in a file opened with block submode, and this record’s length is not a multiple of the file’s buffer size, Synergy Language generates an “End of file” error (\$ERR_EOF) and copies the file record into the buffer. A call to %RSIZE returns the actual length of the record read. Any excess characters in the buffer are not filled.

The record to position to is specified either by the POSITION qualifier, the MATCH qualifier (valid on ISAM files only), the RFA qualifier, or *key_spec* in the following order of precedence:

1. If the POSITION qualifier is specified, MATCH, RFA, and *key_spec* are ignored.
2. If the MATCH qualifier is specified, the RFA qualifier is ignored, unless MATCH:RFA is specified. *Key_spec* is also ignored if MATCH:RFA or MATCH:SEQ are specified.

3. If the POSITION qualifier is not specified, but RFA is, *key_spec* is ignored.
4. If the POSITION, MATCH, and RFA qualifiers are not specified, *key_spec* defines the record to position to.

ISAM

By default, if MATCH is not specified (which is equivalent to setting MATCH:Q_GEQ) and no record's key field begins exactly as specified by the key value, the next available record according to the collating sequence of the index of reference is located and a "Key not same" error (\$ERR_KEYNOT) occurs. The data is still transferred, and the record is still locked. If no higher record exists, an "End of file" error (\$ERR_EOF) occurs. (See [MATCH on page 4-227](#) for more information about how keys are matched.)

If you specify a key value in *key_spec* and the key value length is greater than the record's key field, only the first part of the key value locates the record. If the key's length is less than the size of the key field, the value is called a partial key. In this case, the record located is the first one having a key that begins with the specified value.

The key of reference is determined as follows:

- ▶ If the KEYNUM qualifier is specified, it defines the key of reference.
- ▶ If KEYNUM is not specified, READ uses the implied key of reference. The implied key of reference is a field within *data_area* that matches the offset and contiguous length of the key defined by *key_spec*. If a match is not found, the first key that matches the offset and has a greater length than *key_spec* is the implied key of reference.
- ▶ If KEYNUM is not specified and the conditions for an implied key of reference do not exist, READ uses the primary key as the key of reference if system option #45 is not set.

If you want to READ a segmented key that is specified by *key_spec*, you must first construct that key by concatenating each segment. You can use the %KEYVAL intrinsic function to return the extracted key value from the specified record. See [%KEYVAL on page 9-201](#) for more information.

On OpenVMS, if you specify both the RFA and KEYNUM qualifiers on the READ statement, KEYNUM must specify the primary key as the key of reference (Q_PRIMARY or 0).

NON-ISAM

On Windows and UNIX, if a non-ISAM file is opened with no submode specified and the OPEN does not specify a record size, the runtime assumes the file contains records that are each the same size as the receiving data area. Based on this size and the number of the record to be retrieved, the runtime determines the offset within the file and begins at the first character of the record.

If you attempt to read a record beyond the end of file, you'll get an "Illegal record number specified" error (\$ERR_RECNUM).



See the note under [“Record Locking” on page 4-243](#) for information about inconsistencies that may occur after an I/O error is encountered.

See also

- ▶ [“I/O error lists” on page 6-3.](#)
- ▶ [“Input and Output Statement Qualifiers” on page 4-207](#) for a complete description of each I/O statement qualifier.

Examples

The following internal subroutine searches for an ISAM record based on the primary key and a sequence number, optionally skipping records marked as purged.

```
prifind,
    read(g_datachn, g_modlog, seqnum)
    &      [KEY = notfound, EOF = notfound]
    if ((.not. g_inc_purg) .and. (status .eq. "P"))
        begin
            xcall prompt("Requested record has been purged."
                &      "Enter 'Y' to include purged"
                &      "records", char)
            if (char .eq. 'Y')      ;Toggle incl of purged records
                begin
                    g_inc_purg = (g_inc_purg .xor. 1)
                    xcall tprgmsg      ;Update the status message
                end

            while ((.not. g_inc_purg) .and. (status .eq. "P"))
                reads(g_datachn, g_modlog, notfound)
        end
    return

notfound,
    a_err = E_NOTFOUND
    return
```

The following example references a data file that contains item information that uses the item number as the reference code. The reference codes are numerically ordered, starting with item number 00001. The example gets an item code number, validates it, and then reads the appropriate record. It calls a user-defined subroutine to display the information, then repeats the whole process.

```

subroutine get_item
    a_maxitem      ,d
    a_chn          ,d      ;Channel opened in i:r mode
    a_data         ,a      ;Data area for item record
    .define TTCHN      ,1
record
    item           ,a5
    item_num       ,d5

proc
    repeat
        begin
            display(TTCHN, "Item #: ")
            reads(TTCHN, item) [eof=done]
            item_num = item
            if (item_num .gt. a_maxitem .or. item_num .le. 0)
                begin
                    display(TTCHN, "Invalid item number", $scr_mov(-1,-80),
&                        $scr_clr(line))
                    nextloop
                end
            read(a_chn, a_data, item_num)
            xcall show_item(a_data)
        end

    done,
        xreturn
endsubroutine

```

READS – Read the next sequential record

```
READS (channel, data_area[, label][, REVERSE][, DIRECTION:dir_spec]
&      [, GETRFA:new_rfa][, LOCK:lock_spec][, NOFILL][, WAIT:wait_spec])
&      [, error_list]
```

Arguments

channel

The channel on which the file to be read is open in input or update mode. **(n)**

data_area

The variable that receives the information. **(a)**

label

(optional) A label to which control is passed if an “End of file” error (\$ERR_EOF) occurs.

REVERSE

(optional) Only valid with Synergy ISAM and relative files and specifies that the record is to be read sequentially in the reverse direction.

DIRECTION

(optional) Only valid with Synergy ISAM and relative files and specifies a forward or reverse sequential read. See [DIRECTION on page 4-215](#) for a complete description.

GETRFA

(optional) Returns the record’s RFA after the READS has been performed. See [GETRFA on page 4-218](#) for a complete description.

LOCK

(optional) Only valid with Synergy ISAM files and specifies whether the record is to be locked after the READS has been performed. See [LOCK on page 4-223](#) for a complete description.

NOFILL

(optional) Prevents the data area from being padded with blanks when the data returned does not completely fill *data_area*. See [NOFILL on page 4-229](#) for a complete description.

WAIT

(optional) Specifies how long to wait for a record lock to be released. See [WAIT on page 4-242](#) for a complete description.

error_list

(optional) An I/O error list. If any one of the specified errors occurs as part of the READS, control is transferred to the associated label.

Discussion

The READS statement retrieves the next available record according to the collating sequence of the index of reference (i.e., whether the key is ordered ascending or descending). If REVERSE is not specified, READS retrieves the record that logically follows the record that was most recently read from or written to the file. The next record context is established as follows:

For file type	The context is established as
ISAM	The first record in the file for the primary key when the file is opened. If a READ is performed, the next record follows or precedes the record read (depending on the DIRECTION qualifier) in the order of the specified index. If a FIND is performed, the record read by a READS is the last record positioned to by the FIND. The next record is not affected by a STORE, WRITE, or DELET.
Relative	The first record in the file when the file is opened unless the POSITION qualifier is specified on the OPEN statement. If a READ is performed, the next record follows or precedes the record read (depending on the DIRECTION qualifier). If a FIND is performed, the record read by a READS is the last record positioned to by the FIND. The next record is not affected by a WRITE. A WRITES updates the context so the next record follows the record written by the WRITES.
Stream and sequential	The first record in the file when the file is opened unless the POSITION qualifier is specified on the OPEN statement. If a READ is performed, the next record follows or precedes the record read (depending on the DIRECTION qualifier). If a FIND is performed, the record read by a READS is the last record positioned to by the FIND. WRITE and WRITES update the context so the next record follows the record written by the WRITES, WRITE, or PUT.

By default, READS locks the record, unlocking any previous automatic lock. If *channel* is open in update mode and READS tries to access a locked record, a “Record is locked” error (\$ERR_LOCKED) is generated. If the program does not trap the error, it is considered fatal.

If the file record size is less than that of the data area, the record is placed into the data area left-justified over blanks. If the file record size is greater than the size of the data area, only the leftmost portion of the record is transferred, and an “Input data size exceeds destination size” error (\$ERR_TOOBIG) is generated. (If system option #27 is set, this error is not generated. See #27 in the “System Options” chapter of *Environment Variables and System Options* for more information about this option, which allows a larger buffer to be stored into a smaller destination.)

With the ability to read reverse, beginning-of-file (BOF) and end-of-file (EOF) are both treated as the file “end.” When we refer to EOF, we mean either end-of-file or beginning-of-file.

A READS on an empty file positions you to the end of the file, not the beginning. Therefore, if you perform a READS on an empty file followed by a STORE, the next READS statement generates an “End of file” error (\$ERR_EOF).

If you attempt to read the record following the last record in a file, you'll get an "End of file" error (\$ERR_EOF). When this error condition occurs, the current record position becomes undefined.

If REVERSE is specified but *label* is not, a comma is still required as an argument placeholder. Specifying the REVERSE keyword enables you to re-enter a file sequentially after reaching the end-of-file. You may not receive the record that was previously read, however, because the end-of-file moves as records are added to the file. A sequential READS back into the file reads the first or last record that is currently in the file.



On OpenVMS, READS REVERSE is only supported on ISAM and relative files, and only with OpenVMS version 6.1 and above.

The first READS that follows a FIND on the same channel ignores the DIRECTION and REVERSE qualifiers and retrieve the record that was positioned to by the FIND.



See the note on [page 6-2](#) for information about inconsistencies that may occur after an I/O error is encountered.

WIN

A READS from the terminal behaves differently on Windows than on other platforms. On UNIX and OpenVMS, any text that was previously displayed at the location of the READS remains displayed until the user types over it. On Windows, this text is erased for the width of the buffer passed to READS. The reason for this difference is that on Windows, the input is performed using an edit control to give the user the benefit of editing features (arrow keys, HOME/END, cut/copy/paste, and so forth). For text to be displayed within the edit control, it must also be returned to the program if the user merely presses ENTER. This is not operationally equivalent to the behavior on UNIX and OpenVMS, where the text, although displayed, is not returned in the READS buffer unless the user types it in. We therefore opted for keystroke compatibility, as opposed to visual compatibility, with other systems. The behavior on UNIX and OpenVMS is actually the less consistent of the two, but because it has historical precedent, we cannot change it without breaking existing code.

If you want to prefill an area for input, use a different input method. Possibilities include various UI Toolkit routines (I_INPUT, I_INPFLD, U_FLD) or the Synergy windowing API routine W_FLDS with the WF_INPUT subfunction.

ISAM

With READS, the records are retrieved in the order of the key of reference accessed by the last FIND or READ on the channel. If READS is the first I/O action on the channel, the input is processed by the first data record of the primary key index. You can also use READS to retrieve ISAM records with duplicate keys after a READ has located and retrieved the first record of the set.



On the READS statement for ISAM files, the REVERSE keyword is only available for consistency with Synergy Language version 4. We suggest that you use the DIRECTION qualifier instead of REVERSE.

NON-ISAM

WIN, UNIX

Characters are read from the file to the data area until one of the following conditions occurs:

- ▶ A record terminator is encountered.
- ▶ The data area is filled.
- ▶ The end-of-file is encountered.

If the data area is filled before a record terminator is encountered, an “Input data size exceeds destination size” error (\$ERR_TOO BIG) is generated, and input characters are ignored until one of the other conditions occurs. If a record terminator is encountered before the data area is filled, the remaining space is filled with blanks. (Valid record terminators in Synergy are ASCII values 10 through 12, and 13 when followed by 10 on Windows.)

VMS

For nonstream files, the next record is read into the data area.

For terminals or serial devices, READS behaves as documented in the WIN, UNIX section above, with the following additional rules:

- ▶ If the FLAGS subroutine’s position 9 is set, which terminates a field entry implicitly when the field is full, no error is generated when the data area is filled.
- ▶ Additional terminators can be defined using the ACCHR subroutine. The terminator can be found using %RTERM. The number of characters read can be found using %RDLEN.
- ▶ The terminator for READS is returned in %RTERM and is not part of the returned characters in the data area, nor is it part of the read length.



If you read the last record in a file and want to retrieve that record later, unless the file is open in exclusive mode, you can't guarantee that the same record is at the end-of-file. Use the %RECNUM intrinsic function to get the record number or to capture the RFA of the current end-of-file, then retrieve that record specifically later. See [%RECNUM on page 9-262](#) for more information.

Prefetching records to improve performance with x/Server

To improve sequential READS performance when using x/Server, a prefetch feature is available that enables the client to prefetch (or cache) sequential records for files of any type that are open in input mode, relative files that are open for update, or ISAM files that are open for update and use the LOCK:Q_NO_TLOCK option.

Prefetching is off by default. To turn prefetching on, set the SCSPPREFETCH environment variable to a value between 1 and 32, which specifies the size of the prefetch buffer in kilobytes. To turn prefetching off, set SCSPPREFETCH to 0. You can enable or disable prefetching on a file-by-file basis by using the SETLOG routine to set or clear SCSPPREFETCH prior to the OPEN statement. The value of SCSPPREFETCH is checked on every remote file open. See [SCSPPREFETCH](#) in the “Environment Variables” chapter of *Environment Variables and System Options* for more information.

Prefetching is not allowed in the following instances:

- ▶ When a READS(REVERSE) is being performed on an ISAM file on a key with duplicates.
- ▶ When a READS with a lock follows a READS(LOCK:Q_NO_TLOCK) on a channel open for update on an ISAM file on a key with duplicates.

If you use prefetching in either of these cases, a “Sequential read caching error” (\$ERR_SEQRDS) will occur. If this happens, immediately try doing a keyed READ to reset your current position.

To use prefetching without modifying your code, you can set system option #55, which automatically maps READS with LOCK:Q_NO_LOCK to READS with LOCK:Q_NO_TLOCK. See [#55](#) in the “System Options” chapter of *Environment Variables and System Options* for details and restrictions.

See also

- ▶ [“Input and Output Statement Qualifiers” on page 4-207.](#)
- ▶ [“I/O error lists” on page 6-3.](#)

Examples

The following example scans through an entire relative file looking for a specific record.

```

subroutine find_rec
    a_chn      ,d                      ;Relative file channel

    .include "REC:who.rec"             ;Record "who" includes field "name"

record
    srch_name,a20

proc
    xcall flags(4020, 1)
    display(TTCHN, "Name to find: ")
    reads(TTCHN, srch_name, done)
    find(a_chn, who, ^first)
    repeat
        begin
            reads(a_chn, who, none)
            if (name .eq. srch_name)
                exitloop
        end
        writes(TTCHN, who)

done,
    sleep 1
    display(TTCHN, $scr_clr(screen))
    xreturn

none,
    writes(TTCHN, "No record found")
    goto done
endsubroutine

```

RECORD-ENDRECORD – Define a data record

```
[access] [rec_mod] RECORD [name] [,X]
    member_def
    .
    .
    .
[ENDRECORD]
```

Arguments

access

(optional) For a class record only, one of the following access modifiers:

PUBLIC	Access is not restricted. This is the most accessible option.
PROTECTED	Access is limited to the containing class or types derived from the containing class.
PRIVATE	Access is limited to the containing type. This is the least accessible option. (default)

rec_mod

(optional) One of the following record modifiers:

LOCAL	For a routine record only, the contents of the record persist for as long as the routine is activated at one or more levels.
STACK	For a routine record only, the contents of the record are unique for every activation of the routine.
CONST	For a class record only, field values are not changed once they are initialized. CONST implies STATIC.
READONLY	For a class record only, field values can only be set during declaration or in the constructor of the class.
STATIC	For both routine and class records, the contents of the record persist throughout the life of the program.

name

(optional) The name of the entire record area to define.

,X

(optional) Indicates that you're defining an overlay for the most recent nonoverlay record or common.

member_def

Either the definition of a single field in the record or a group declaration. Each field definition has the syntax described in “[Field definitions](#)” on page 2-20. Each group declaration has the syntax described in [GROUP-ENDGROUP](#) on page 4-86. You can specify as many member definitions as you want.

Discussion

The RECORD statement defines a data structure and its component fields. A record can be declared within a class or within a routine. The record is either global or local, depending on whether or not the RECORD statement is part of a GLOBAL-ENDGLOBAL statement block.

You can specify an access modifier and the CONST or READONLY modifier only on a class record. These modifiers apply to all groups and fields in the record unless they are overridden at the field level. CONST and READONLY are mutually exclusive.

The LOCAL and STACK modifiers can be specified only on a routine record, not a class record. If you specify the LOCAL modifier, the record’s data is initialized on the associated routine’s initial activation, and any changes are guaranteed to remain in effect while the routine is activated at one or more levels. If the routine is not activated at any level, it can be removed from memory in some environments. If the routine is removed, the routine’s next activation is exactly equivalent to its initial activation. LOCAL is the default for non-re-entrant routines.

If you specify the STATIC modifier in a subroutine or function, the contents of the static record upon entry are the same as they were when the routine was last exited. For a class record, fields are shared by all instances of the class. The record’s data is initialized when the program is started, and any changes are guaranteed to be persistent until the entire program is exited. If you specify the STATIC modifier in a REENTRANT subroutine or function, the data referenced is the same on every activation level. You don’t need to use the STATIC modifier in your main routine, because data defined in the main routine is never reinitialized; all records in the main routine are implicitly static.

Initial values are not allowed on STACK records, and if you specify the STACK modifier, the data area for the associated record is never initialized. Whenever an activation level of a routine is entered, a data area unique to that activation is allocated. (Its contents are undefined and probably different each time.) Then, when that activation level is exited, the unique data area is released.

If you don’t specify STATIC or STACK, the record defaults to the default record type of the routine, which is determined by the presence of the LOCALDATA, STACKDATA, STATICDATA, or REENTRANT (implying STACKDATA) routine qualifier. Records that are not in a class default to LOCAL.

Access or record modifiers on an unnamed class record can be overridden by a modifier on one of its fields. If a field modifier is specified, it takes precedence over the record modifier. Widening accessibility is not allowed; if a record is public, a field can be specified as private, but not vice-versa.

If you don't specify *name*, the record is unnamed. If you specify *name*, the record is accessed as data type alpha. The record name must be unique within the class or routine in which it is declared. If *name* is specified on a class record, it represents an instance variable of that record type within that class and can be used to access fields in that record within all instance methods of the class. If *name* is specified on a routine record, it represents a local variable of that record type within that routine and can be used to access fields in that record within that routine. For example,

```
method mymethod, void
record myrecord
    f1, i4
proc
    myrecord.f1 = 4
    mreturn
end
```

See [“Overlaying data” on page 2-44](#) for a list of rules that apply to overlay records, which are indicated by “,X”.

At least one field or group definition must be present after the RECORD statement.

If you declare a named group within a class record, the group and its members inherit the accessibility of the class record. Accessibility cannot be widened.

The ENDRECORD statement is optional. If you don't specify ENDRECORD, the record is automatically terminated at the start of the next syntactical component.

Examples

The following example defines a global data section named **gbl** that has 11 characters and a local area of 35 characters. The global data section consists of one global record named **g1** and one unnamed alternate layout. The local area consists of one local record named **rec1** (21 characters) and one unnamed record (14 characters). Record **rec1** has one named alternate layout, or overlay, named **r1** and one unnamed alternate layout. The unnamed record has an alternate layout named **ovr**.

```
global data section gbl      ,INIT
    record g1
        g1a      ,d5,      164
        g1b      ,3d2,      1, 27, 3
    record ,x
        govr1     ,d2
        govr2     ,a1
    endglobal

record rec1
    alpha      ,a13,  "Initial value"
    beta       ,a2
    gamma      ,d6,      975126
```



```

record ,x
    r1fld1    ,a3
    r1fld2    ,a7
record r1 ,x
    r2fld1    ,a5
    r2fld2    ,a1

record
    xx1        ,a7
    xx2        ,d7,    11765

record ovr ,x
    xx10vr     ,a3
    xx1end     ,d4

```

The following example defines a class record.

```

namespace ns1
class c1
    record r1
        f1        ,i4
        f2        ,i4
    endrecord
    public record r2
        f1        ,i4    ;This field inherits public accessibility
        private f2 ,i4    ;This field is explicitly private
    endrecord
endclass
endnamespace

```

RECV – Receive a message

RECV (*message*, *label*[, *message_id*, *length*])

Arguments

message

A variable that is loaded with the received message, left-justified over blanks. (**a**)

label

A label to which control is transferred if no message is pending.

message_id

(optional) A variable that specifies the name of the message. It can be up to six characters long. (**a**)

length

(optional) A variable that will be returned with the length of the received message. (**n**)

Discussion

The RECV statement receives a message from a previously executed SEND statement in the same or a different program.

If the received message is longer than *message*, only the leftmost portion of the message is received and the remainder is ignored.

When a message is sent by the SEND statement, it is assigned a message ID. That message can only be received when the same ID is specified in a RECV statement. If *message_id* is not specified, the ID is assumed to be the name of the receiving routine (up to 31 characters, left-justified over blanks).

The receiving routine's name is defined with the MAIN, FUNCTION, or SUBROUTINE statement. If MAIN is not used in the main routine, the filename is used as the message ID.

You can also use the specially interpreted [SELF] ID as the message ID. The [SELF] ID causes the name of the message ID to be the default program name or the name set by the RCVID subroutine.

If *message_id* is specified, *length* must also be specified. If more than one message with the same ID is outstanding, the messages are received in first-in-first-out order.

If *length* initially has a negative value, the message received is not deleted from the message pool, so it can be received again. If *length* is greater than or equal to zero, the message is deleted and cannot be received again. (This is called a destructive receive.) If *length* is not specified, a destructive receive is the default action.

VMS

The “SEND/RECV message failure” error (\$ERR_MSGFAIL) on a RECV statement is normal if many messages are queued and the message controller has not yet processed them. In this case, the RECV has queued another message in sequence, but the message controller has not processed it in the time the runtime has allotted for a reply. You should retry the RECV if you are sure the message controller is actually running. This error protects the user from hanging if the message controller aborts; however, the time to wait is dependent on the processing activity and speed of the system, so you must determine if a retry is required.

See also

- ▶ [“Messaging” on page 1-38.](#)
- ▶ [SEND on page 4-165](#) for information about the [SELF] ID.

Examples

See [SEND on page 4-165](#) for an example that uses the RECV statement.

REMOVEHANDLER – Unregister a registered event handler

WIN

`REMOVEHANDLER (event, method_name)`

Arguments

event

The name of an event, or an object reference to a field that is of the type `Synnet.DotNetEvent` or inherited from `Synnet.DotNetEvent`.

method_name

The name of the method (including the instance) that handles the event.

Discussion

The REMOVEHANDLER statement can only be used with code generated by the **gennet** utility. (See “[The Gennet Utility](#)” in the “General Utilities” chapter of *Synergy Language Tools* for more information about **gennet**.)

Method_name is in the form *instance.method*. For example, if **c1** is an instance of **class1**, *method_name* could be any of the following:

- ▶ **c1.mymeth**
- ▶ **this.mymeth** (for this instance in a class)
- ▶ **parent.mymeth** (for the parent’s instance in a class)

When the event goes out of scope, all the methods registered for the event are automatically unregistered.

If you use REMOVEHANDLER for an event handler that has not been previously registered, the statement will be ignored.

See also

- ▶ [ADDDHANDLER on page 4-13.](#)
-

REPEAT – Repeatedly execute a statement

REPEAT *statement*

Arguments

statement

A single or compound statement to be repeatedly executed.

Discussion

The REPEAT statement repeatedly executes a statement until a condition occurs to transfer control to another statement.

Statement can be on the next logical line. It is executed repeatedly until control is explicitly transferred to a label outside of the scope of *statement*, or until an EXITLOOP is executed.

This statement is functionally equivalent to the DO FOREVER statement.

See also

[DO FOREVER on page 4-44.](#)

Examples

```
.define TTCHN      ,1
record
    password      ,a5
external function
    chkpass      ,d

proc
    open(TTCHN, o, "tt:")
    xcall flags(50000, 1)
    repeat
        begin
            display(TTCHN, "Enter password: ")
            reads(TTCHN, password, exit)
            if (%chkpass(password))
                exitloop
            writes(TTCHN, "Invalid Password")
            sleep 1
        end
        display(TTCHN, $scr_clr(screen), "XYZ Accounting System")
        xcall process      ;User is valid - now do something
    exit,
    stop
end
```

RETURN – Return control from a subroutine

RETURN

Discussion

The RETURN statement returns control to the statement that follows the call to the most recently executed subroutine. RETURN is normally used to return from internal subroutines, although it can be used to return from external subroutines (see note below).

If RETURN is executed with no outstanding CALL or XCALL present, a “Return with no CALL” error (\$ERR_NOCALL) is generated.



For consistency with Synergy Language version 4, a RETURN statement without a corresponding CALL in a subroutine is functionally equivalent to the XRETURN statement. We recommend that you use the XRETURN statement, instead of RETURN, to return from an external subroutine.

See also

[“Writing Subroutines and Functions” on page 7-2](#) for more information on internal subroutines.

Examples

In the following example, RETURN is used both to return from an internal subroutine and from an external subroutine.

```
subroutine mysub
record
    val      ,i4
proc
    val = 1
    call do_val      ;Call to internal subroutine
    val = val x 2    ;Point A
    return          ;Returns from subroutine back to calling
                   ; program (equivalent of xreturn)
do_val,
    val = val + 7
    return          ;Returns to Point A
endsubroutine
```

SEND – Send a message

`SEND (message, message_id[, terminal])`

Arguments

message

A variable that contains the message to be sent. **(a)**

message_id

The name that identifies the message (up to 6 characters, or 39 characters on OpenVMS). **(a)**

terminal

(optional) The number of the terminal to which the message will be sent. The number is operating system specific. **(n)**

Discussion

The SEND statement sends a message to the same or a different program.

The message controller saves the message until it is retrieved by the RECV statement. The maximum length of the message is system-dependent.

Only a RECV statement with the same message ID can receive the message.

The sending routine's name is defined with the MAIN, FUNCTION, or SUBROUTINE statement. If MAIN is not used in the main routine, the filename is used as the message ID.

WIN, UNIX

Synergy Language has two specially interpreted message IDs:

- [SELF]** The message ID is the default program name or the name set by the RCVID subroutine. [SELF] eliminates the need to update the message ID if the routine name changes and ensures that the SEND and RECV will function as intended. [SELF] sends to any program in the process chain.
- [INIT]** No message is sent. Instead, [INIT] deletes all messages pending for the terminal associated with the program. Messages associated with other programs are not affected.

The square brackets around [SELF] and [INIT] are part of the message IDs.

When [SELF] is specified in the SEND statement, the program determines the message ID to send based on the current program name. When it is specified in the RECV statement, the program determines the message ID to look for based on the current program name. Therefore, if you SEND to [SELF] in program **abc**, the message ID will be "abc". If you then chain to program **xyz** and do a RECV on [SELF], the program will look for "xyz", not "abc", and the message will not be found.

See also

[“Messaging” on page 1-38.](#)

Examples

The following example shows a message being sent to another program, which receives it using the [SELF] ID.

The file **records.def** contains the following:

```
record cust_master
name    ,a30
addr    ,2a40
city    ,a15
state   ,a2
zip     ,d5
phone   ,d10
fax     ,d10
```

The file **custmain.dbl** contains the following:

```
.include "records.def"

proc
xcall get_cusrec(cust_master)
send(cust_master, "cust")
stop "cust"
end
```

The file **customer.dbl** contains the following:

```
main cust
.include "records.def"
record
len ,d5

proc
if (%true) then

recv(cust_master, nomsg, "[SELF]", len)

else                                     ;Recv wasn't chained to
begin
nomsg,
xcall get_cusrec(cust_master)
end
end
```


SET – Assign a value to a set of variables

SET variable[, ...] = expression

Arguments

variable

One or more variables to which a value will be assigned. All variables within a single SET statement must be the same data type. (**a** or **n**)

expression

An expression whose result is assigned to each of the variables in the SET statement. (**a** or **n**)

Discussion

The SET statement assigns a single value to one or more variables.

Expression must be the same data type as *variable*.

The SET statement is useful for assigning large sets of variables to a single value.



We recommend using the SET statement rather than individually assigning the same value to multiple fields.

Examples

```
set dflag1, dflag2, dflag3 = 1
```

SLEEP – Suspend program execution

`SLEEP interval`

Arguments

interval

The number of seconds for which program execution is to be suspended. (**n**)

Discussion

The SLEEP statement suspends program execution for a specified amount of time. *Interval* can be specified as any numeric type (integer, decimal, implied-decimal, packed, or implied-packed) and as either a variable or a literal. If *interval* is an implied value, the runtime will sleep for the specified number of seconds and milliseconds. Note that the processing of the milliseconds is system dependent, and some systems are able to sleep for a finer granularity than others.

After the time specified by *interval* has elapsed, execution resumes with the statement following the SLEEP statement.

If *interval* is greater than 65,535, you'll get an "Arithmetic operand exceeds maximum size" error (\$ERR_BIGNUM). If *interval* is less than or equal to zero, no sleeping occurs.

UNIX

If system option #12 and the TBUF I/O environment variable is set, the I/O buffer is flushed before the SLEEP statement is executed.

See also

- ▶ [#12](#) in the "System Options" chapter of *Environment Variables and System Options*.
- ▶ [TBUF](#) in the "Environment Variables" chapter of *Environment Variables and System Options*.

Examples

```
sleep 5
```

```
sleep 3.2
```

```
sleep int_var
```

```
sleep imp_var
```

```
sleep 2.532           ;Note that the length of time slept could be 2 or 2.5
                      ; or 2.53 or 2.532 seconds
```

Also see [WHILE on page 4-196](#) for an example that uses the SLEEP statement.

SORT – Sort a file

```
SORT ( INPUT=filename [, RECORD=record [ / rec_switch ] ]
&      [, KEY= ( key_field [ / key_switch, ... ] ) [ / ], OUTPUT=out_file [ / ], END=end_val ]
&      [, TEMPFILE=temp_file [ / ], OPTIONS=option_string ] )
```

Arguments

INPUT

Precedes the filename to sort.

filename

The name of the input file to sort. **(a)**

RECORD

(optional) Precedes a record name that defines the file record format.

record

The name of the record that defines the file record format. If *record* is not specified, the RECSIZE qualifier must be included in the OPTIONS string. **(a)**

rec_switch

(optional) One of the following switches that identifies the record as either fixed-length or variable-length. Each switch must be preceded by its own slash (/). **(a)**

f Fixed-length records (default)

v Variable-length records

KEY

(optional) Precedes one or more fields to use as a sort key. You can specify up to 15 keys. SORT precedence is determined by the order in which the keys are specified.

key_field

A named field within *record* to use as a key. It should not reference an array element. Use a named overlay of the array element instead. If *key_field* is not specified, the KEY qualifier must be specified in the OPTIONS qualifier string. **(a)**

key_switch

(optional) One of the key switches that indicates the order of the sort. Each switch must be preceded by its own slash (/). **(a)**

f Forward (ascending) sort order (default)

r Reverse (descending) sort order

a Alpha key (no case distinction)

- d** Decimal key (numeric ordering)
- c** Character key (case sensitive ordering) (default)

OUTPUT

(optional) Precedes an output filename.

out_file

The name of the output file. If *out_file* is not specified, the input file is replaced by the resulting sorted file. If *out_file* is specified, the input file is sorted, the resulting file is *out_file*, and the input file is not modified. **(a)**

END

(optional) Indicates an end-of-file specification.

end_val

A list of alpha or decimal values to use as either maximum record counts or logical end-of-file characters for the input files. **(a or n)**

TEMPFILE

(optional) Specifies a temporary file to be used during the SORT.

temp_file

The name of a temporary file to be used during the SORT. If *temp_file* is not specified, or if it consists of a device or directory name only, SORT builds a file with a unique name in the default or specified device and directory. This file is created and deleted during the SORT. **(a)**

OPTIONS

(optional) Precedes one or more SORT options to be parsed at runtime.

option_string

Has a maximum size of 256 and the following format: **(a)**

```
[RECSIZE=size[/rec_swch]][, KEY=(key)][, OUTPUT=out_file][, TEMPFILE=temp_file]
& [, END=end_val[, ...]]
```

RECSIZE

(optional) Precedes the size of the records in the file. This qualifier must be specified if the RECORD compile-time qualifier is not specified.

size

A literal that states the size of the records in a file. For files containing variable-length records, use the maximum record length. **(n)**

rec_swch

One of the following switches. Each switch must be preceded by its own slash (/). (**a**)

f *Size* is the size of a fixed-length record. (default)

v *Size* is the size of a variable-length record.

e *Size* is the exact size of a fixed-length record, including record terminators.

KEY

(optional) Precedes the SORT keys. You can specify up to 15 keys in *key*. This qualifier must be specified if the KEY compile-time qualifier is not specified.

key

A key specification in the following form: (**a**)

pos:len[/swch, ...]

pos The position of the key field within the record (**n**)

len The length of the given key (**n**)

swch (optional) Any one of the switches listed above under *key_switch*. Each switch must be preceded by its own slash (/).

OUTPUT

(optional) Precedes an output filename (*out_file*). This qualifier is functionally similar to the OUTPUT compile-time qualifier, except that only literal values are accepted.

TEMPFILE

(optional) Precedes the name of a temporary file (*temp_file*) to use during the SORT. This qualifier is functionally similar to the TEMPFILE compile-time qualifier, except that only literal values are accepted.

END

(optional) Specifies a list of alpha or decimal literals (*end_val*) to use as either maximum record counts or logical end-of-file characters for the input files. This qualifier is functionally similar to the END compile-time qualifier, except that only literal values are accepted.

Discussion

The SORT statement sorts a file in a specified key order. You can place the result into a new file or back into the original file.

The OPTIONS qualifier overrides the corresponding SORT qualifier. It enables you to build SORT commands during program execution. This causes some minor extra runtime overhead, so we recommend that you use compile-time arguments wherever possible.

SORT always creates a temporary file, regardless of whether or not you specify the TEMPFILE qualifier. If the TEMPFILE qualifier is not specified and you’re sorting locally,

On	The temporary file is created in
Windows	One of the following directories: ▶ The directory specified by the TMP environment variable. ▶ The directory specified by the TEMP environment variable if TMP is not defined. ▶ The Windows directory if neither TMP nor TEMP is defined.
UNIX	The current directory (where the program was started, not where <i>in_file</i> or <i>out_file</i> is located).
OpenVMS	SYS\$SCRATCH, if it exists.

Make sure sufficient free disk space is available wherever temporary files are created. See [“Sorting a file remotely” on page 4-173](#) for information about the default temporary file in a client/server configuration.

You must use the **/e RECSIZE** switch when sorting records that may contain binary data, such as RFAs and integer data types.

The SORT statement opens and closes the input file.

If *end_val* is an alpha value, the first record containing the individual *end_val* character as the first character of the record denotes the end-of-file. The record containing the end-of-file character is not sorted. If *end_val* contains more than one character, only the first character is used. If *end_val* is a numeric value, it represents the number of the last record in the file to be sorted. If *end_val* is not specified, the file is sorted until the physical end-of-file is reached.

WIN, UNIX

SORT outputs a relative file (variable or fixed) regardless of the type of the input file. ISAM files cannot be sorted.

Sorting a file remotely

In a client/server configuration, the SORT statement is passed to *x/Server* which performs the sort locally on the server, rather than sorting across the network. For this reason, the filenames specified on the SORT statement *must* refer to the same machine, either all local or all remote. For example, if the input file is on the server, the output file must also exist on the server. The following SORT statement shows proper syntax:

```
SORT(input="DAT:orders", output="DAT:ordered, ...)
```

while this SORT statement generates an error:

```
SORT(input="DAT:orders", output="ordered", ...)
```

If you want to sort a file on one machine and have the resulting file copied to another machine, use XCALL COPY in conjunction with the SORT statement.

On a remote sort, the default location of the temporary file varies by operating system:

- ▶ On Windows, it is placed in the directory that TMP or TEMP is set to in the Windows registry (not the environment); if they are not set, it is placed in C:\Documents and Settings\All Users\Shared Documents (XP/2003) or C:\Users\Public\Public Documents (all other Windows platforms).
- ▶ On UNIX, it is placed in /tmp.
- ▶ On OpenVMS, it is placed in SYS\$SCRATCH.

Examples

The following example sorts a file by two keys and creates a sorted output file.

```
subroutine srtfil
record customer
    id            ,a6
    name          ,a30
    addr          ,2a30
    city          ,a15
    state         ,a2
    zip           ,a9

literal
    CLIENTS      ,a*, "DAT:clients.ddf"
    SRTFIL       ,a*, "SRT:clients.srt"

proc
    sort(input=clients, record=customer, key=(zip,name), output=srtfil)
    xreturn
endsubroutine
```

The following subroutine does the same thing as the above example, but uses the runtime option string instead of the compile-time qualifiers.

```
subroutine srtfil          ;Sort any file that needs two sort keys
                          ; and an output file.
    a_srtfile            ,a    ;File to sort.
    a_recsize            ,d
    a_key1pos            ,d
    a_key1len            ,d
    a_key1sw             ,a
    a_key2pos            ,d
    a_key2len            ,d
    a_key2sw             ,a
    a_outfile            ,a

record
    optlen                ,d3
    srtopts               ,a160

proc
    xcall s_bld(srtopts, optlen, "rec=%d, key=(%d:%d/%a, %d:%d/%a),
&                        out=%a", a_recsize, a_key1pos, a_key1len, a_key1sw,
&                        a_key2pos, a_key2len, a_key2sw, a_outfile)
    sort(input=a_srtfile, options=srtopts)
    xreturn
endsubroutine
```


STOP – Terminate program execution

STOP [*function*]

Arguments

function

(optional) One of the following termination functions to perform:

- ▶ The name of a program to which to chain. (**a**)
- ▶ The name of a command file to which to chain. (**a**)
- ▶ An exit status to send to the operating system. Synergy/DE predefines two of these in a system-independent way: (**n**)

D_EXIT_SUCCESS Operation succeeded.

D_EXIT_FAILURE Operation failed.

VMS

- ▶ A DCL command, with an optional argument list. (**a**)
-

Discussion

The STOP statement terminates program execution and passes control either to another program or the monitor.

If *function* is a program, it can be any executable program on OpenVMS, but it must be a **.dbr** file on Windows and UNIX. If the program is a bound program, *function* must be the name specified with the **-b** or **-p** compiler option.

To chain to a command file, precede the command filename with an “at” sign (@) as follows:

```
stop "@filename"
```

or

```
stop "$@filename"
```

VMS

To chain to a DCL command, precede the command with a dollar sign (\$) as follows:

```
stop "$command [arg1] [arg2] ..."
```

If *function* is not specified, no further program statements are executed and control returns to the operating system prompt. All I/O channels that are not closed before the STOP are purged.



Even though the channel is purged, pending terminal input characters are not removed during a STOP chain. You must drain the buffer before stopping to another program. For example,

```
while %ttsts(chan) accept(chan, dvar)
```

If *function* is a name but is not found, a “File not found” error (\$ERR_FNF) is generated.

System option #43 controls whether a Synergy application produces a stop message when a STOP statement is executed. See [#43](#) in the “System Options” chapter of *Environment Variables and System Options*.

Examples

The following example returns an error status to the operating system (on UNIX).

```
stop 1
```

The following example chains to the **rpt.dbr** program and the **cleanup** command file, respectively.

```
stop "rpt"  
stop "@cleanup"
```

STORE – Store a record to an ISAM file

STORE (*channel*, *data_area*[, GETRFA:*new_rfa*][, LOCK:*lock_spec*]) [[*error_list*]]

Arguments

channel

The number of the channel on which the ISAM file is open in update mode (U:I). (**n**)

data_area

The variable containing the record to be added. (**a**)

GETRFA

(optional) Returns the record's RFA. See [GETRFA on page 4-218](#) for a complete description.

LOCK

(optional) Specifies whether a manual lock is applied. Automatic locking is not supported for STORE. See [LOCK on page 4-223](#) for a complete description.

error_list

(optional) An I/O error list. If any one of the specified errors occurs as part of the STORE, control is transferred to the associated label.

Discussion

The STORE statement stores or adds a record to an ISAM file. This statement is used exclusively with ISAM files.

Once it is determined that the STORE is valid, the appropriate indexes are updated and the data is added. If the data area is longer than the defined maximum length of a record in the ISAM file, an "Invalid record size" error (\$ERR_IRCSIZ) occurs and no data is transferred to the file.

On a STORE operation, all indexes have the new record's key values inserted, unless a key is null according to the definition of the key. See [ISAMC on page 9-170](#). STORE ensures that new key values are unique for keys that don't allow duplicates. If a duplicate key is encountered, a "Duplicate key specified" error (\$ERR_NODUPS) occurs.

STORE does not change the sequential context of the file for this channel. A READS following a STORE returns the record following the last READ or FIND. By default, the newly inserted record is not locked after the STORE.

See also

- ▶ ["Input and Output Statement Qualifiers" on page 4-207.](#)
- ▶ ["I/O error lists" on page 6-3.](#)

Examples

The following subroutine gets new client information and stores it in an ISAM file.

```
subroutine getinfo
    a_sts          ,d1      ;Returned with TRUE (successful),
                        ;or FALSE (error)

record customer
    last_name      ,a20
    first_name     ,a20
    crdt_limit     ,a6
    crdt_type      ,a2

proc
    a_sts = %true
    display(TTCHN, $scr_clr(screen), $scr_pos(5, 10),
&        "Last name: ")
    reads(TTCHN, last_name, done)
    display(TTCHN, $scr_pos(7, 10), "First name:")
    reads(TTCHN, first_name, done)
    display(TTCHN, $scr_pos(9, 10), "Credit limit: ")
    reads(TTCHN, crdt_limit, done)
    display(TTCHN, $scr_pos(11, 10), "Credit type: ")
    reads(TTCHN, crdt_type, done)
    store(ISMCHN, customer)
    xreturn

done,
    a_sts = %false
    xreturn
endsubroutine
```

STRUCTURE-ENDSTRUCTURE – Define how data is laid out

```
[access] [PARTIAL] STRUCTURE name
  member_def
  .
  .
  .
[ENDSTRUCTURE]
```

Arguments

access

(optional) For a class structure only, one of the following access modifiers:

PUBLIC	Access is not restricted. This is the most accessible option.
PROTECTED	Access is limited to the containing class or types derived from the containing class.
PRIVATE	Access is limited to the containing type. This is the least accessible option.

PARTIAL

The structure declaration can be separated into two or more declarations that will be compiled together as a single structure. Partial structures may span one or more source files.

name

The name of the structure area to define.

member_def

Either the definition of a single field or a group declaration. Each field definition has the syntax described in [“Field definitions” on page 2-20](#). Each group declaration has the syntax described in [GROUP-ENDGROUP on page 4-86](#). You can specify as many member definitions as you want.

Discussion

The STRUCTURE statement defines how data is laid out. Unlike the RECORD statement, it does not allocate any data space. A structure can be declared globally or within a namespace, class, or routine.

A structure declared globally or within a namespace or class can be used as a data type for a parameter, return type, field, property, or local variable. See [“structure” on page 2-13](#) for more information.

A structure declared in an implicit main routine before any record construct is considered to be a member of the default namespace. If you want to declare a structure within an implicit main routine, add an explicit MAIN or RECORD before the structure declaration. For example,

```
structure struct1           ;Structure in default namespace
    fld1, i4
endstructure

record
structure struct2           ;Structure is declared locally to implicit main
    fld2, i4
endstructure
```

Structures within a routine can be used only as the first operand of the $\wedge M$ data reference operation or as an argument to a $\wedge SIZE$ operation. Any other type of reference generates a compilation error. The resulting data structure has an implied $[*]$ dimension specification. This single-stride dimension specification allows array access of the format $\wedge m(\text{structure}[n], \text{memory\_handle})$, where n is limited to the size of the structure divided into the size of the memory handle. For example, given a structure defined with the name **str** and a field **fld**, the following field expression is always allowed:

```
 $\wedge m(\text{str}[n].\text{fld}, \text{handle})$ 
```

$\wedge M$ of a structure that has an object field generates an error, because you can't map an object on memory.

When used in a dynamic array, an initial value declared in a structure is propagated to each element of the array

The ENDSTRUCTURE statement is optional but recommended. If not specified, the structure is automatically terminated at the start of the next syntactical component. (Note, however, that depending on what the next component is, there are rare instances, such as with the `xfMethod` attribute, when the compiler cannot terminate the structure properly, and you must specify ENDSTRUCTURE.) When declaring a global structure, explicitly specifying ENDSTRUCTURE is recommended, and the compiler issues an "ENDSTRUCTURE on global structure expected" (NOSPECL) warning if it is omitted.

See also

- ▶ ["structure" on page 2-13.](#)
- ▶ ["Using a structure with a structfield" on page 2-44.](#)

Examples

The following example declares a public structure called **mystruct** that contains two fields.

```
public structure mystruct
    field1    ,i4
    field2    ,a20
endstructure

main
    str      ,mystruct
proc
    str.field1 = 7
end
```

SUBROUTINE-ENDSUBROUTINE – Define an external subroutine

```
SUBROUTINE name[, option, ...]  
    param_def  
    .  
    .  
    .  
[ENDSUBROUTINE | END]
```

Arguments

name

The name of the external subroutine to define.

option

(optional) One or more of the following modifiers:

LOCAL LOCALDATA	Keep record contents constant for as long as the routine is activated at one or more levels.
STACK STACKDATA	Make record contents unique for every activation of the routine.
REENTRANT	Allow multiple active instances of the subroutine.
VARARGS	More than the declared number of arguments can be passed to this routine.

WIN, UNIX	
RESIDENT	Keep the routine in memory, even when not in use.
ROUND	Use rounding rules for implied-decimal data types within the subroutine.
TRUNCATE	Use truncation rules for implied-decimal data types within the subroutine.

param_def

Subroutine parameter definition. See [“Parameter definitions” on page 2-33](#).

Discussion

The SUBROUTINE statement identifies the beginning of an external subroutine.

Subroutine names of longer than 30 characters are truncated when the routine is linked (although you can specify up to 255 characters as long as the first 30 characters are unique).

The compiler performs a page feed when the SUBROUTINE statement is encountered.

The LOCAL and STACK modifiers specify the default state of an unqualified RECORD statement. For example, given the following statement:

```
subroutine fred, STACK
```

all of the unqualified RECORD statements in subroutine **fred** are treated as if they are STACK RECORD statements. You can specify either LOCAL or STACK, but not both. If you don't specify LOCAL or STACK, the default is LOCAL, unless the REENTRANT option is specified. If REENTRANT is specified, unqualified RECORD statements default to STACK data, rather than LOCAL data, for each instance of the subroutine. For example, given the following statement:

```
subroutine fred, REENTRANT
```

all of the unqualified RECORD statements in subroutine **fred** are treated as if they are STACK RECORD statements.

When a combination of LOCAL, STACK, or REENTRANT is used, the last one on the line determines how unqualified RECORD statements are processed.



The REENTRANT modifier causes all local records to be STACK records unless otherwise qualified. Fields in STACK records do not allow default or explicit initial values.

By default, an external subroutine cannot call itself; if it does, the runtime generates a "Recursive XCALL" error (\$ERR_RECEXTCAL). However, specifying the REENTRANT modifier on the SUBROUTINE statement enables you to call an external subroutine recursively. (In other words, the subroutine can call itself.)



Object handles are not allowed in local records in REENTRANT routines.

WIN, UNIX

By default, a subroutine may be removed from memory when it is no longer in the calling chain. If you specify the RESIDENT modifier, the routine stays in memory.

VARARGS is optional for unprototyped subroutines and functions, but it is required when using **-qnet** or strong prototypes (or internal routine prototyping unless **-qrelaxed:param** is set). It permits you to pass more arguments than declared.

A program by default rounds all expression results. You can change the default to truncate by setting system option #11. However, specifying the ROUND or TRUNCATE modifier on a SUBROUTINE statement overrides the default rounding behavior for that routine (including if system option #11 is set). You cannot specify both TRUNCATE and ROUND in the same statement. See [#11](#) in the “System Options” chapter of *Environment Variables and System Options* for more information about rounding versus truncation rules.

See also

[Chapter 7, “Understanding Routines.”](#)

THROW – Throw an exception

THROW [*object_handle*]

Arguments

object_handle

(optional) An object handle derived from System.Exception.

Discussion

The THROW statement unwinds the stack automatically until a handler that handles the type of exception thrown is located. This allows errors that occur deep within a series of routine calls to be handled at a higher level by a single exception handler.

The THROW statement adds a line of the format

```
" at namespace.class.member in filename:line line_num"
```

to the StackTrace property of the exception for each nested level on the unwind stack. The topmost line indicates the line where the THROW occurred.

You can rethrow the most recently caught exception and maintain the exception's original context by using the THROW command within the CATCH block without specifying *object*. You cannot rethrow outside of a CATCH block.

For example, the following will rethrow the caught exception:

```
catch (e, @Exception)
begin
    throw
end
```

See also

- ▶ [“Structured Exception Handling” on page 6-4.](#)
- ▶ [TRY-CATCH-FINALLY-ENDTRY on page 4-186.](#)

Examples

The example below throws the object returned by “new Exception(“My Error”)”.

```
throw new Exception("My Error")
```

TRY-CATCH-FINALLY-ENDTRY – Handle exceptions

```

TRY
    .
    .      ; TRY block
    .
    [CATCH (exception_var [, @exception_type])
        .
        .      ; one or more CATCH blocks
        .]
    [FINALLY
        .
        .      ; FINALLY block
        .]
ENDTRY

```

Arguments

exception_var

A CATCH block exception variable, whose scope is confined to the CATCH block in which it is declared.

exception_type

The type of the exception. The default type is System.Exception.

Discussion

The TRY block should contain all of the statements that are subject to the exception handler. It can have zero or more CATCH statements, each handling a different type of exception. A CATCH block is only entered when an appropriate exception is being caught.

When a thrown exception is being processed, the system looks through the CATCH blocks for an exception type that is equal to or in the hierarchy of the thrown exception. The first CATCH statement that meets the condition is executed. If no CATCH blocks in the current frame handle this type of thrown exception, the stack continues to unwind either until an appropriate CATCH block is found or until an unhandled exception is reported.

Exception_var must be type System.Exception, or it must inherit from it. It cannot be a statement keyword or a keyword reserved for the procedure division. You don't need to declare a CATCH block *exception_var* in the data section. If a CATCH block catches a thrown exception, the *exception_var* for the block will be assigned the instance of the thrown exception that it caught.

If a CATCH block does not rethrow the caught exception, the destructor for the exception is run automatically when the CATCH block is exited.

A call to a label in a higher scope is not allowed in a CATCH statement.

You can throw an exception using the THROW statement. (See [THROW on page 4-185](#).)

You can place any code that must be run regardless of whether or not an exception is handled in the FINALLY block. Typically the FINALLY block releases critical resources, such as file locks. At least one CATCH or one FINALLY block is required. If a FINALLY block exists, it is always run on exit from the TRY or CATCH block. Exit conditions include end of block, RETURN statements, and statements that jump to a label outside the TRY/CATCH block.

The following statements and routine calls are not allowed in a FINALLY block: XRETURN, FRETURN, MRETURN, CALL, RETURN, EXITTRY, EXITE, and DBL\$EXITERORR. Code jumps (for example, GOTO, I/O error list, EXITLOOP, READS(„eof), and so on) are also not permitted. An EXIT statement is allowed if the referenced BEGIN-END block resides entirely within the FINALLY block.

See also

- ▶ [“Structured Exception Handling” on page 6-4](#).
- ▶ [THROW on page 4-185](#).

Examples

```
subroutine TestExceptions
record
    tt          ,i4
proc
    try
    begin
        open(tt=%syn_freechn,i,"tt:")
        ;open(%syn_freechn,i,"dontfindme.txt")    ;Uncomment this line to see
                                                ; IO exception being caught
        throw new Exception("My Error Message") ;Throws the exception here
        writes(tt,"SHOULD NOT SEE THIS!")
    end
    catch (ioe, @SynIOException)                ;Catch Synergy IO exception
    begin
        writes(tt,"Caught this IO error:"+ioe.Message)
    end
    catch (e, @Exception)                        ;Catch any exception not handled
                                                ; by a previous catch
    begin
        writes(tt,"Caught this error:"+e.Message)
    end
end
```

Synergy Language Statements

TRY-CATCH-FINALLY-ENDTRY

```
    finally
    begin
        writes(tt,"Now in finally block")
        close(tt)          ;Use FINALLY block to release critical resources
    end
endtry
endsubroutine
```

UNLOCK – Release a record lock on a channel

UNLOCK *channel* [, RFA: *match_rfa*]

Arguments

channel

The channel on which the file is open. (n)

RFA

(optional) Unlocks the record with the specified RFA. See [RFA on page 4-237](#) for a complete description.

Discussion

The UNLOCK statement releases an automatic lock on the specified channel or a lock specified by the RFA.

This statement is ignored if no lock conditions exist on the specified channel.

See also

- ▶ [FREE on page 9-133](#).
- ▶ [“Record Locking” on page 4-243](#).

Examples

The following subroutine unlocks both automatic and manual locks.

```
subroutine rel_locks
  a_chn          ,n
  a_rfa          ,a

proc
  if (^passed(a_rfa)) then
    unlock a_chn, RFA:a_rfa
  else
    unlock a_chn
  xreturn
endsubroutine
```

The following example reads and locks a relative record, and then prompts the user to add a new record, change the locked record, delete the locked record, or exit the routine. If the user selects exit or presses the EOF character, the record is explicitly unlocked.

```
subroutine proc_rec
    a_chn      ,d          ;Channel of file open in update mode
    a_data     ,a          ;Data record to read
    a_recnum   ,d          ;Record number to read

    .define TTCHN      ,1
record
    ans                ,a1

proc
    read(a_chn, a_data, a_recnum);Record locked
    repeat
        begin
            display(TTCHN, "Add/Change/Delete/Exit: ")
            reads(TTCHN, ans, exit)
            case ans of
                begincase
                    "A":
                        xcall add
                    "C":
                        xcall change
                    "D":
                        xcall delete
                    "E":
                        exitloop
                endcase
            end
        end

    exit,
        unlock a_chn
        xreturn
endsubroutine
```


UPCASE – Convert lowercase characters to uppercase

UPCASE *variable*

Arguments

variable

A variable containing the characters to be converted to uppercase. (a)

Discussion

The UPCASE statement converts lowercase characters to their corresponding uppercase characters. It does not change the value of any other character.



The LOCALIZE subroutine controls the effect of the UPCASE statement on the characters [, \,], and ^ by enabling ANSI-compliant translation.

On those systems that support a user setting of the current language locale (Windows, OpenVMS, and most UNIX), the runtime automatically determines which 8-bit characters are uppercase or lowercase according to the current language locale of the user.

See also

[LOCALIZE on page 9-206.](#)

Examples

```
record
    fld1      ,a14,      "just a [demo]."
```



```
proc                                ;The results will be:
    upcase fld1(4,11)              ;"jusT A [DEMO] ."
    upcase fld1                    ;"JUST A [DEMO] ."
    stop
end
```

USING – Conditionally execute a statement

```

USING control SELECT
([match_term[, ...]]),
    statement
.
.
.
ENDUSING

```

Arguments

control

An expression whose value determines which statement is executed. (**a** or **n**)

match_term

(optional) Defines one or more conditions, enclosed in parentheses, in which *statement* is executed. *Match_term* uses any combination of the following formats:

<i>exp</i>	A match expression. A match occurs if <i>control</i> is equal to <i>exp</i> .
<i>exp_1</i> thru <i>exp_2</i>	Two expressions separated by the THRU keyword. A match occurs if <i>match_term</i> is equal to or greater than <i>exp_1</i> and less than or equal to <i>exp_2</i> .
<i>operator exp</i>	An expression preceded by a relational operator. (See “Operators” on page 3-4 for a list of operators.) <i>Control</i> is compared to <i>exp</i> by <i>operator</i> and evaluated for its truth value. If it’s true, a match occurs.
()	Null term. If no <i>match_terms</i> are matched, the null term is matched.

statement

A Synergy Language procedure division statement that are executed if one or more of the *match_term* expressions are true.

Discussion

The USING statement selects a single statement for execution from a set of statements based on the value of a specified expression. It is similar to the CASE statement, except that it provides more powerful structures. USING is most efficient when using an **i** or **d** *control* expression and all *match_terms* are compile-time literals.

Each *match_term* is evaluated from top to bottom and left to right. Once a match is found, no other conditions in the *match_term* list are evaluated..



If you use alpha labels, be aware that the comparisons are .EQ. and .NE., which will be compared for the length of the shortest field. If you would like to match against the exact label, you should use the USING statement with the label (.EQS."ABC").

See also

- ▶ [CASE on page 4-18.](#)
- ▶ [USING-RANGE on page 4-194.](#)

Examples

```
.define TTCHN,1
record
    code          ,a5

proc
    open(TTCHN, o, "tt:")
    display(TTCHN, "Enter code: ")
    reads(TTCHN, code, done)
    using code select
        ('0' thru '9'),
            xcall class1(code)
        ('A' thru 'Z'),
            xcall class2(code)
        ("99", '$'),
            xcall class3(code)
        (.gt.'z'),
            xcall ierror(code)
        (),
            xcall ferror(code)
    endusing

done,
    stop
end
```

USING-RANGE – Conditionally execute a statement with a range

```

USING control RANGE low THRU high SELECT
  (match_value[, ...]),
  statement
.
.
.
ENDUSING

```

Arguments

control

An expression or single-character alpha literal whose value determines which statement is executed. (**a** or **n**)

low

A compile-time expression that specifies the lowest value within the range.

high

A compile-time expression that specifies the highest value within the range.

match_value

(optional) Defines a condition in which *statement* is executed, using any combination of the following formats:

<i>exp</i>	A compile-time expression. A match occurs if <i>control</i> is equal to <i>expression</i> .
<i>exp_1</i> thru <i>exp_2</i>	Two expressions separated by the THRU keyword. A match occurs if <i>match_value</i> is equal to or greater than <i>exp_1</i> and less than or equal to <i>exp_2</i> .
<i>operator exp</i>	An expression preceded by a relational operator. (See “Operators” on page 3-4 for a list of operators.) <i>Control</i> is compared to <i>exp</i> by <i>operator</i> and evaluated for its truth value. If it’s true, a match occurs.
%INRANGE	A match occurs if <i>control</i> was not matched by any literal match values, but was within the specified range.
%OUTRANGE	A match occurs if <i>control</i> is outside the specified range.

statement

A Synergy Language procedure division statement that is executed if one or more of the *match_value* expressions is true.

Discussion

The USING-RANGE statement selects a single statement for execution from a set of statements based on the value of a specified expression.

USING-RANGE builds a dispatch table at compile time and is faster than the USING statement. It is similar to the CASE statement, except that it provides more powerful structures.



Note the following difference between USING-RANGE and USING: implied values used by *control*, *low*, *high*, and *match_value* are always rounded or truncated in the USING-RANGE statement.

Any arithmetic that is done within a compile-time expression (*low*, *high*, or *match_value*) is whole number arithmetic and has the normal order of precedence. If the evaluated expression does not match one of the *match_values*, the statement immediately following the USING-RANGE statement is executed.

Not all of the values from *low* to *high* need to be specified. You can use the functions %INRANGE and %OUTRANGE to match any values that are not specified. A null *match_value* list (empty parentheses) matches any selection value within the range that is not specified or any value outside the range. A null list is equivalent to the *match_value* list (%INRANGE, %OUTRANGE).

Each *match_value* is evaluated from top to bottom and left to right. If the *match_value* falls outside of the range specified from low to high, the compiler warnings “Select value is outside defined range” (OUTOFRANGE) and “Statement can never be executed” (NOTEXE) are generated, and no match is set up for that value. If you specify an alpha literal longer than one character as your *match_value*, the compiler generates an “Only one character allowed” warning (ONEALPHA).

Once a match is found, no other conditions in the *match_value* list are evaluated.



If you use alpha labels, be aware that the comparisons are .EQ. and .NE., which will be compared for the length of the shortest field. If you would like to match against the exact label, you should use the USING statement with the label (.EQS.“ABC”).

See also

- ▶ [CASE on page 4-18.](#)
- ▶ [USING on page 4-192.](#)

WHILE – Repeatedly execute a statement

`WHILE condition [DO] statement`

Arguments

condition

A conditional expression that is evaluated before each repetition of the specified statement.

statement

A single or compound statement to be executed repeatedly.

Discussion

The WHILE statement executes a statement repeatedly as long as a specified condition is true. *Statement* is executed while *condition* is true. When the *condition* becomes false, control transfers to the logical statement after the WHILE.

Statement can be on the next logical line from WHILE condition. If DO is present, it must be on the same logical line as WHILE.



Statement is not executed if *condition* is false at the initial iteration.

Examples

The following example performs password validation.

```
.define TTCHN          ,1
record
    password           ,a5

proc
    open(TTCHN, o, "tt:")
    xcall flags(50000, 1)           ;Turn off input echo
    while (password .ne. "DEMO ") do
        begin
            display(TTCHN, "Enter password: ")
            reads(TTCHN, password, exit)
            if (password .ne. "DEMO")
                begin
                    writes(TTCHN, "Invalid password")
                    sleep 1
                end
        end
    end
```

```
        display(TTCHN, $scr_clr(screen), "XYZ Payroll System")
        xcall process          ;Start the package

exit,
    stop
end
```

WRITE – Write a record to a file

```
WRITE (channel, data_area[, record][, POSITION:pos_spec][, GETRFA:new_rfa]
&      [, RFA:match_rfa]) [ [error_list] ]
```

Arguments

channel

The channel on which the file is open. The channel must already have been opened in output or update mode. **(n)**

data_area

An expression that contains the information to be written. **(a)**

NON-ISAM

record

(optional) The ID of the record to write or one of the following options. The record ID is a numeric variable whose value is the ordinal number of the desired record. *Record* must be passed when writing to a file other than an ISAM or relative file.

^FIRST Evaluate to the first record currently in the file.

^LAST Evaluate to the last record currently in the file.

^EOF Evaluate to the nonexistent record starting at the end-of-file.

POSITION

(optional) Specifies an absolute position in the file. See [POSITION on page 4-233](#) for a complete description.

GETRFA

(optional) Returns the record's RFA after the WRITE has been performed. See [GETRFA on page 4-218](#) for a complete description.

NON-ISAM

RFA

(optional) Writes the record at the specified RFA. See [RFA on page 4-237](#) for a complete description.

error_list

(optional) An I/O error list. If any one of the specified errors occurs as part of the WRITE, control is transferred to the associated label.

Discussion

The WRITE statement outputs a record to a file. If the specified record already exists in the file, it is replaced. On file types other than ISAM, if the record doesn't exist, it is created.

NON-ISAM

If you specify a WRITE with no record number to a relative file, the runtime writes the current record.

ISAM

The record being modified must be the record most recently retrieved (locked) by a FIND, READ, or READS statement. If it is not locked, a "Record not locked" error (\$ERR_NOCURR) occurs. If you try to modify the value of a key that was not defined as modifiable when the ISAM file was created, a "Key not same" error (\$ERR_KEYNOT) occurs. If the data area is longer than the defined maximum length of a record in the ISAM file, an "Invalid record size" error (\$ERR_IRCSIZ) occurs, and no data is transferred to the file.

You can't modify the primary key with the WRITE statement. To modify the primary key or any key that is not modifiable, you must delete the existing record and then STORE the modified record with a new primary key.

After the WRITE, any automatic record locks are released for the record. The WRITE appends a record terminator to the data area written.

A WRITE to ^EOF extends the file, and the record just written (the last record in the file) becomes the current record.

On OpenVMS, ^LAST is only supported for ISAM and relative files.

When using RFAs from a file with data compression or variable-length records, you may need to use the GETRFA option to retrieve the potentially new RFA. This isn't necessary if the file has STATIC_RFA defined.

See also

- ▶ ["Input and Output Statement Qualifiers" on page 4-207.](#)
- ▶ ["I/O error lists" on page 6-3.](#)

Examples

The following subroutine updates a modification tracking record.

```

subroutine writerec
    a_datachn      ,d      ;Channel for data file (modlog file)
    a_seqnum       ,d      ;Current sequence number
    a_orig_seqnum  ,d      ;Original sequence number
    a_modlog       ,a      ;Modification record
    .define TTCHN      ,1

proc
    if (a_seqnum .eq. a_orig_seqnum) then
        begin
            write(a_datachn, a_modlog, a_seqnum)[key = badkey]
            display(TTCHN, $scr_pos(5,22),
&                "Sequence number ", ^a(a_seqnum),
&                " has been updated")
            read(a_datachn, a_modlog, a_seqnum)
&            [key = badkey] ; reread for update
        end
    else
        ;Don't let them change the sequence number
        writes(TTCHN, "Sequence number may not be modified")
    return

badkey,
    writes(TTCHN, "Invalid key specified")
    return
endsubroutine      ;End of subroutine writerec

```

The following example references a data file that contains item information and uses the item number as the reference code. The reference codes are numerically ordered starting with item number 00001. The example gets an item code number, validates it, reads the appropriate record, updates the information, then writes the updated record back to the file.

```

subroutine get_item
    a_maxitem      ,d
    a_chn          ,d      ;Channel opened in i:r mode
    a_data         ,a      ;Data area for item record
    .define TTCHN      ,1
record
    item           ,a5
    item_num       ,d5

```

```
proc
  repeat
    begin
      display(TTCHN, "Item #: ")
      reads(TTCHN, item) [eof=done]
      item_num = item
      if (item_num .gt. a_maxitem .or. item_num .le. 0)
        begin
          display(TTCHN, "Invalid item number", $scr_mov(-1,-80),
&          $scr_clr(line))
          nextloop
        end
      read(a_chn, a_data, item_num)
      xcall show_item(a_data)
      xcall update_item(a_data)
      write(a_chn, a_data, item_num)
    end
  done,
  xreturn
endsubroutine
```

WRITES – Write the next sequential record to a channel

NON-ISAM

`WRITES (channel, data_area[, label][, GETRFA:new_rfa]) [[error_list]]`

Arguments

channel

The channel on which the file is open. The channel must already have been opened in output or update mode. **(n)**

data_area

An expression that contains the information to be written. **(a)**

label

(optional) The label to which control is transferred if the output device is full.

GETRFA

(optional) Returns the record's RFA after the WRITE has been performed. See [GETRFA on page 4-218](#) for a complete description.

error_list

(optional) An I/O error list. If any one of the specified errors occurs as part of the WRITES, control is transferred to the associated label.

Discussion

The WRITES statement writes the record immediately following the most recent record that was read from or written to the output file to a specified channel. It also appends a record terminator to the data area written.

After the WRITES, any automatic record locks are released.

See also

[“I/O error lists” on page 6-3.](#)

Examples

The following example scans through a file, extracting the **name** field into a separate list file.

```
subroutine find_rec
    a_chn          ,d          ;Input file channel

    .include "REC:who.rec"      ;Record "who" includes field "name"
    .define TMPCHN          ,99
    record
        list_file          ,a*,    "WRK:name.lst"

    proc
        xcall flags(4020, 1)
        open(TMPCHN, o, list_file)
        repeat
            begin
                reads(a_chn, who, done)
                writes(TMPCHN, name)
            end
        done,
            close TMPCHN
            xreturn
    endsubroutine
```

XCALL – Call an external subroutine or function

`XCALL subroutine ([argument, ...])`

Arguments

subroutine

The name of an external subroutine or function to which control will be transferred.

argument

(optional) A list of one or more arguments that are passed to the subroutine or function.

(**a** or **n**)

Discussion

The XCALL statement transfers control to an external subroutine or user-defined function. (It cannot be used to call system-supplied functions.) Upon return from the subroutine or function, control is transferred to the statement following the XCALL statement.



Any ONERROR traps set in the calling program are disabled while the external subroutine is being processed. Once control returns to the calling program, the error traps are re-enabled.

See also

[“Writing Subroutines and Functions” on page 7-2.](#)

Examples

See [USING on page 4-192](#) for an example that uses the XCALL statement.

XRETURN – Return control to a calling routine

XRETURN [*return_value*]

Arguments

return_value

(optional) A ^VAL value to be returned to the caller. In an external subroutine, this value can be retrieved using XSTAT. It can also be retrieved by declaring the subroutine as a ^VAL function and returning *return_value* as the function return value. A C routine can also retrieve the return value from XRETURN. **(n)** (Synergy standard only)

Discussion

The XRETURN statement returns control to the calling routine from an external subroutine. Any internal CALLs in the subroutine are disregarded; in other words, you don't have to RETURN from them before XRETURNing to the calling routine.

XRETURN cannot be used in a main routine or a function. If the compiler encounters an XRETURN statement outside of a subroutine, it generates an error.

See also

- ▶ [XSTAT on page 9-394](#) for an example of retrieving the value in the optional *return_value* argument.
- ▶ [“Writing Subroutines and Functions” on page 7-2](#) for more information about external subroutines.

Examples

The following example shows an external subroutine that uses XRETURN.

```
subroutine choice
    a_data          ,a
external function
    decide          ,d

proc
    repeat
        begin
            case %decide(a_data) of
                begincase
                    writes(TTCHN, "processing result #1")
                    writes(TTCHN, "processing result #2")
                    writes(TTCHN, "processing result #3")
                endcase
```

```
        else
            writes(TTCHN, "... invalid result")
        end
    xreturn
endsubroutine
```


Input and Output Statement Qualifiers

Many of the I/O statements have qualifiers that enable you to control the function of the statement. These qualifiers are discussed on the pages that follow.

Each I/O statement qualifier is followed by an expression. The valid values for each statement qualifier are defined by the compiler (and in the **dbl.def** file included with your distribution, which you can use as a reference). If you do not want these definitions available, you can specify the global definitions compiler option (**-g** on Windows and UNIX or **/nogbldefs** on OpenVMS), which instructs the compiler not to define them. See [“Invoking the compiler”](#) in the “Building and Running Synergy Language Programs” chapter of *Synergy Language Tools* for more information about this compiler option.

For your reference, we have listed the available identifiers for each I/O statement qualifier, followed by the equivalent value for each identifier.



When specifying a qualifier, you can either use a colon before the expression (for example, `KEYNUM:krf_spec`) or an equal sign (for example, `KEYNUM=krf_spec`).



The KRF qualifier is not listed in this section, but it is available for consistency with Synergy Language version 4. KRF is functionally equivalent to KEYNUM. We suggest that you use KEYNUM instead of KRF for new development.

ALLOC – Pre-allocate a number of blocks

ALLOC : blocks

Arguments

blocks

The number of 512-byte blocks to allocate for the file. (**n**)

Discussion

The ALLOC qualifier defines the initial allocation size for a file being opened for output.

If the file is not opened in output mode, this qualifier is ignored, except on OpenVMS or unless system option #15 is set.

See also

[OPEN on page 4-124.](#)

BKTSIZ – Set bucket size

VMS

BKTSIZ : *bucket_size*

Arguments

bucket_size

The bucket size. (**n**)

Discussion

The BKTSIZ qualifier specifies the RMS bucket size to be used on the OPEN for output statement.

See also

[OPEN](#) on page 4-124.

BLKSIZ – Set block size

VMS

BLKSIZ : *block_size*

Arguments

block_size

The block size. (**n**)

Discussion

The BLKSIZ qualifier specifies the RMS block size argument in bytes.



We recommend that you only use BLKSIZ for MAGTAPE devices or to alter the record size of a B submode output file. Use BUFNUM to alter the performance of sequential files. (Values less than the OpenVMS default of 16 cause a reduction in performance.)

See also

[OPEN on page 4-124.](#)

BUFNUM – Set number of process local I/O buffers

VMS

BUFNUM : *number*

Arguments

number

The number of process local I/O buffers in the range 0 through 127. (**n**)

Discussion

The BUFNUM qualifier notifies RMS of the number of process local I/O buffers.

If BUFNUM is not specified or has a value of zero, the process default for the particular file organization is used. If the process default is also zero, the system default is used.

See also

[OPEN on page 4-124.](#)

BUFSIZ – Set buffer size

VMS

BUFSIZ : *blocks*

Arguments

blocks

The number of blocks to allocate within the range 1 through 127. (**n**)

Discussion

The BUFSIZ qualifier specifies the number of blocks to allocate for the I/O transfer buffer. If specified, BUFSIZ overrides any buffer size specified in the proc statement.

See also

[OPEN on page 4-124.](#)

CONTIG – Allocate contiguity of a file

VMS

CONTIG: *value*

Arguments

value

One of the following values: (**n**)

- 0** The file is not allocated contiguously.
- 1** The file is allocated contiguously. If sufficient space is not available, the file is not created and an “Output file is full” error (\$ERR_FILFUL) occurs.
- 2** The file is allocated on a best-try basis. If sufficient space is not available, the file is created noncontiguously and no error occurs.

Discussion

The CONTIG qualifier enables you to specify the desired contiguity of a file. This qualifier is used in conjunction with ALLOC and is only used if the file is opened for output.

See also

[OPEN on page 4-124.](#)

DEQ – Set number of blocks to add

VMS

DEQ : *blocks*

Arguments

blocks

The number of blocks to add. (**n**)

Discussion

The DEQ qualifier specifies the number of blocks to add when a file is automatically extended. Automatic extension occurs whenever a program performs a WRITE, WRITES, or STORE, and the space that is currently allocated is exhausted.

In some cases, DEQ reduces overhead by reducing the number of times a file must be extended. In other cases, DEQ makes it possible to extend a file when available space is less than the system default extend quantity.

When creating a new file, the DEQ qualifier becomes the permanent default for the file. When opening an existing file, you can use DEQ to override the file's default during the time the file is open.

See also

[OPEN on page 4-124.](#)

DIRECTION – Set direction of a READS

DIRECTION : *dir_spec*

Arguments

dir_spec

One of the following identifiers or corresponding values: (**n**)

Q_IGNDIR or **0** Ignore the DIRECTION qualifier.

Q_FORWARD or **1** Read the next sequential record in a forward direction. (default)

Q_REVERSE or **2** Read the next sequential record in a reverse direction.

Discussion

The DIRECTION qualifier specifies the direction in which to perform the READS.



The REVERSE keyword, which is functionally similar to the DIRECTION:Q_REVERSE qualifier, is available on the READS statement. However, on the READS statement for ISAM files, we suggest that you use DIRECTION instead of REVERSE.

The DIRECTION qualifier is ignored if it follows a FIND on the same channel.

See also

[READS on page 4-150.](#)

FDL – Specify an FDL or XDL string or file specification

FDL : *string*

Arguments

string

The FDL or XDL string or file specification. (**a**)

Discussion

The FDL qualifier enables you to specify an FDL (File Description Language) or XDL (ISAM definition language) string or file specification. (XDL is a subset of FDL.)

When *string* is an FDL or XDL string (not a filename), it is the contents of an FDL or XDL file stripped of any comments and recomposed with a semicolon between each line instead of a CR-LF.

WIN, UNIX

The FDL qualifier is only allowed for creating an ISAM file. If *string* is a file specification, the default extension is **.xdl**.

VMS

The FDL qualifier always specifies an FDL string or file specification. If *string* is a file specification, the default extension is **.fdl**.

OpenVMS uses the FDL description to provide default RMS file attributes for the file. These default values are superseded by other OPEN statement qualifiers that affect RMS attributes (such as BUFNUM, NUMREC, ALLOC, and so on). The file type in the FDL overrides the type specified in the OPEN statement mode (for example, O:R changes to O:S if the FDL type is sequential).

If the first character of *string* is an “at” sign (@), the remainder of the string is treated as the file specification containing the FDL or XDL description; otherwise, the entire string is treated as an FDL or XDL description.

If a filename is specified in the OPEN statement, the file will be created with that name regardless of what name was defined in the FDL or XDL description. If no filename is specified in the OPEN statement, the file will be created with the name specified in the FDL or XDL description.

If the filename specified in the OPEN statement is a remote file, and the FDL qualifier specifies an FDL file specification, that FDL file must exist on the remote system and cannot be an *x/Server* file specification (containing an “at” sign). Any logical translation is handled on the remote system.

See also

- ▶ [OPEN](#) on page 4-124.
- ▶ “ISAM Definition Language” in the “Synergy DBMS” chapter of *Synergy Language Tools*.

Examples

The following example specifies a file called **[DKA0:]myfile.ism** that contains the FDL or XDL description:

```
FDL:[DKA0:]myfile.ism
```

The following example specifies an actual FDL string:

```
FDL:FILE;  BEST_TRY_CONTIGUOUS  yes;  BUCKET_SIZE      5;  CLUSTER_SIZE 5;
```

GETRFA – Return a record’s RFA

GETRFA : *new_rfa*

Arguments

new_rfa

One of the following:

rfa The returned RFA. (**a6**)

grfa The returned global RFA (GRFA), which must be a 4-byte record CRC and a 6-byte RFA of the current record. (**a10**)

Discussion

The GETRFA qualifier returns the record’s RFA or GRFA after the operation has been performed.

In some cases an RFA is returned even following an error; \$ERR_KEYNOT and \$ERR_TOOBIG generated by a READ or READS still transfer a data record and its RFA.



See the note on [page 6-2](#) for information about inconsistencies that may occur after an I/O error is encountered.

Using GRFAs can help you implement optimistic locking. (See “[Optimistic locking technique](#)” on [page 4-244](#) for more information.) Using the GETRFA qualifier with a GRFA enables a READ or READS statement to read a record along with its CRC. If you save off the GRFA that is returned, a subsequent READ or READS with the RFA qualifier can attempt to reacquire the same record. If a record has changed between reads, it will still be read, but a “Record not same” error (RECNOT) will result because the CRC no longer matches.



GETRFA:*grfa* cannot be used on the FIND statement, on the READ, READS, and WRITE statements for file types other than ISAM and relative, and on the WRITES statement for file types other than relative. The CRC portion of the GRFA is ignored on the UNLOCK statement and the POSRFA routine for ISAM and relative files.

The following built-in types are available when declaring an RFA or GRFA variable.

- D_RFA_TYPE (defined as **a6**)
- D_GRFA_TYPE (defined as **a10**)

GETRFA:*rfa* is functionally equivalent to the GETRFA subroutine.

See also

- ▶ [GETRFA on page 9-150.](#)
- ▶ [FIND on page 4-57.](#)
- ▶ [READ on page 4-145.](#)
- ▶ [READS on page 4-150.](#)
- ▶ [STORE on page 4-177.](#)
- ▶ [WRITE on page 4-198.](#)
- ▶ [RFA on page 4-237.](#)
- ▶ [“Record Locking” on page 4-243.](#)
- ▶ [“Record File Addresses \(RFAs\)” on page 4-246.](#)

Examples

The following routines illustrate the use of GRFA to implement optimistic locking. A record is read from an ISAM file and then parts are updated. The new record is then written back to the file after it has been determined not to have changed. If the record *has* changed, the update is attempted again after notifying the user. A brief record lock only occurs when the on-disk record before the WRITE is the same as it was before the update.

```
.define RECORD_UPDATED      ,1
.define RECORD_NOT_UPDATED ,2
.define USER_CANCEL        ,0
.define USER_RETRY         ,1
.define RECSIZ              ,256

function optimistic_update ,i
    ch                ,n
    rec               ,a
    key               ,a

record
    grfa              ,D_GRFA_TYPE ;Or a10
    tmprec            ,a RECSIZ

proc
    read(ch, rec, key, GETRFA:grfa, LOCK:Q_NO_LOCK)

    xcall update_totals(rec)

    read(ch, tmprec, RFA:grfa, WAIT:Q_WAIT) [$ERR_RECNOT=was_changed]
    write(ch, rec)                          ;Update and unlock the record
    freturn(RECORD_UPDATED)
```

```
was_changed,
    rec = tmprec                ;Note: The lock has been released
    freturn(RECORD_NOT_UPDATED) ; as a result of the error
end

subroutine process_record
    ch          ,n
    rec         ,a
    key         ,a
record
    st          ,i1

proc
    while (%optimistic_update(ch, rec, key) .eq. RECORD_NOT_UPDATED)
        begin
            xcall notify_user(rec, key, st)
            if (st .eq. USER_CANCEL)
                exitloop
                sleep 1
            end
        xreturn
    end
```

GUIWND – Open a new application window

WIN

GUIWND

Discussion

The GUIWND qualifier opens a new Synergy application window through which all I/O for the channel is directed.

You can use GUIWND when opening the console (TT:).

See also

[OPEN](#) on page 4-124.

KEYNUM – Specify a key of reference

KEYNUM : *krf_spec*

Arguments

krf_spec

Either an alpha expression that represents the name of the key, or one of the following identifiers or corresponding values: (**a** or **n**)

Q_PRIMARY or **0** The primary key is the key of reference. (default)

Q_ALT1 or **1** The first alternate key is the key of reference.

Q_ALT2 or **2** The second alternate key is the key of reference.

Q_ALT3 or **3** The third alternate key is the key of reference.

Q_ALT4 or **4** The fourth alternate key is the key of reference.

Q_ALT5 or **5** The fifth alternate key is the key of reference.

Q_ALT6 or **6** The sixth alternate key is the key of reference.

Q_ALT7 or **7** The seventh alternate key is the key of reference.

Although only eight keys are predefined by the compiler, you can add definitions in your own header files to handle however many keys you use.

Discussion

The KEYNUM qualifier specifies a key of reference by which to perform the operation.

If *krf_spec* is an alpha expression, it must be a valid key name defined in the ISAMC subroutine or the **bldism** utility. The maximum key name length is 15 for Synergy ISAM. (RMS ISAM has no maximum length, but because ISAMC accepts a maximum of 32 characters, your key name shouldn't be any longer than 32 characters.)

If KEYNUM is specified, *krf_spec* defines the key of reference. If KEYNUM is not specified, the I/O statement uses the implied key of reference. If KEYNUM is not specified and the conditions for an implied key of reference do not exist, the primary key is used as the key of reference.

See also

- ▶ [FIND on page 4-57.](#)
- ▶ [READ on page 4-145.](#)

LOCK – Control record locking

LOCK : *lock_spec*

Arguments

lock_spec

One of the following identifiers or corresponding values: (**n**)

Q_NO_LOCK or 0	Do not lock the record.
Q_AUTO_LOCK or 1	Perform automatic locking on the record. (default)
Q_MANUAL_LOCK or 2	Perform manual locking on the record.
Q_NO_TLOCK or 3	Do not lock the record or probe the index tree.

Discussion

The LOCK qualifier specifies whether the operation requires a record lock.

Manual locking is similar to automatic locking, except that manual locking blocks all update reads until either the channel is closed, an explicit UNLOCK with the RFA qualifier specified is performed, or the FREE subroutine is called.

Q_NO_LOCK allows you to READ a locked record when it's opened in update mode. The read record is not locked.

WIN, UNIX

Q_NO_TLOCK causes a READS statement on a file open for update to perform like READS on a file open for input. By default, READS on a file open for update probes the index tree and locks the record. With Q_NO_LOCK, it probes the tree but doesn't lock the record. With Q_NO_TLOCK, it does not probe the tree *or* lock the record. On an active file (one with concurrent updates occurring), some records may not be seen at all by a READS using Q_NO_TLOCK without an intervening READ statement.

Q_NO_TLOCK and READ/READS on a channel open for input produce the same results: a cached copy of the current index blocks is searched for the record requested (with READ) or the next record (with READS). As a result, any new records that were stored (or any existing records whose keys changed in the current index) that affect the current cached index blocks will not be seen. These changes are always seen by default on an update channel, because every operation rereads the index block(s). (Of course, a deleted record is not returned.)

For performance reasons (especially locking performance), use Q_NO_TLOCK instead of Q_NOLOCK unless you explicitly want to see concurrent changes by other users.

Q_NO_TLOCK and Q_NO_LOCK are synonymous on OpenVMS systems.

On the OPEN statement for both non-ISAM and ISAM files, the Q_NO_LOCK and Q_AUTO_LOCK are the only valid options for the LOCK qualifier. In general, LOCK:Q_NO_LOCK does not grant access to files opened with an exclusive SHARE mode (either Q_EXCL_RW, Q_EXCL_RO, or files open in append and output mode) except on some UNIX platforms. (See [SHARE on page 4-239](#).)



We do not recommend using Q_NO_LOCK on OPEN, except for files on read-only systems.

On the STORE statement, Q_NO_LOCK and Q_MANUAL_LOCK are the only valid options for the LOCK qualifier.

See also

- ▶ [“Record Locking” on page 4-243](#).
- ▶ [OPEN on page 4-124](#).
- ▶ [STORE on page 4-177](#).
- ▶ [#55](#) in the “System Options” chapter of *Environment Variables and System Options*.

MASK – Specify activation characters

`MASK:act_char`

Arguments

act_char

An array of integers with a total size of 32 bytes and which must start on an aligned **i4** boundary. **(i)**

Discussion

The MASK qualifier specifies activation characters at the statement level, to enable you to replace those specified by the ACCHR, ACESC, DACHR, or DAESC subroutines if you have opened a serial or terminal device other than the console (TT:). MASK is used with the GETS statement.

The *act_char* argument is a series of 256 bits corresponding to the ASCII character codes 0-255. If a bit is set, that character is an activation character.

To terminate on a CR or LF, for example, do the following:

1. Create an array of 256 bits:

```
actchar      , [32]i1
```

(Each **i1** is 8 bits, and 8 x 32 = 256.)

2. Determine the values of CR and LF (13 and 10, respectively).
3. Because the lowest order bit corresponds to 0 rather than 1, subtract 1 from each value (resulting in 12 and 9, respectively).
4. Set bit 12 and bit 9 in the array (bits 5 and 2 of the second byte, or values 16 and 2, which makes 18).
5. Set `actchar[2] = 18`.

See also

[GETS on page 4-77.](#)

Examples

The following routine sets the correct bit in the array for any specified character:

```

subroutine setchars
;
; Description:  Routine to set the appropriate bits in the activation
;               character mask for desired terminators
;
; Arguments:
;
;   a_actchars      , [32]i1      ; Array of activation bit characters
;   a_char          , n           ; 0 or more character values
;
; .define D_BITS_IN_CHAR      , 8      ; Number of bits in a character

; .align
stack record
;   argnm           , i4           ; Argument counter
;   charval         , i4           ; Character value
;   ndx             , i4           ; Index into array
;   group dbits     , d D_BITS_IN_CHAR ; "Bit flags" for a character
;   dbit            , [D_BITS_IN_CHAR]d1
;   endgroup
;   abits           , a D_BITS_IN_CHAR @dbits      ; Union as alpha

proc
  for argnm from ^argnum(a_char) thru %numargs ; For all "a_char" args
    if (^passed(^arg(argnm)))
      begin
        charval = ^argn(argnm) ; Get the character value
        decr charval ; Make 1 = 0
        ndx = (charval / D_BITS_IN_CHAR) ; Base 0 array index
        charval -= (ndx * D_BITS_IN_CHAR) ; Modulo array element size
        clear dbits
        dbit[D_BITS_IN_CHAR - charval] = 1 ; Set the "bit"
        incr ndx ; Make the array base 1
        a_actchars[ndx] = a_actchars[ndx] .bor. %b(abits)
        ; Mask in the bit

      end

    xreturn
  endsubroutine

```

MATCH – Define how a record is located

ISAM

MATCH: *match_spec*

Arguments

match_spec

One of the following identifiers or values: (n)

Q_GEQ or 0	Locate the record that matches the key value greater than or equal to the key specification. If no equal match is found, the next greater record is read and a “Key not same” error (\$ERR_KEYNOT) is generated. If no greater record exists, an end-of-file occurs. (default)
Q_EQ or 1	Locate the record that matches the key value equal to the key specification. If no match is found that is equal to the key specification, a “Record not found” error (\$ERR_RNF) occurs.
Q_GTR or 2	Locate the record that matches the key value greater than the key specification. No error occurs unless end-of-file is reached, in which case, an “End of file” error (\$ERR_EOF) occurs.
Q_RFA or 3	Locate the record with the RFA specified by the RFA qualifier.
Q_GEN or 4	Locate the record that matches the key value that is equal or next in sequence to the key specification. An “End of file” error (\$ERR_EOF) is generated if there are no matches nor any higher records in sequence.
Q_SEQ or 5	Locate the next sequential record following the current record, ignoring the key specification. No error occurs unless end-of-file is reached.

Discussion

The MATCH qualifier defines how a record is located in an ISAM file. It uses relative relationships (for example, greater than and greater than or equal to) to make this determination.

The record that is read is the first one that logically follows *key_spec* according to the collating sequence of the index of reference (i.e., whether the key is sorted ascending or descending). For example, if you have a key ordered ascending, a greater-than match retrieves the next higher key value. However, if the key is ordered descending, a greater-than match retrieves the next lower key value.



The record retrieved by a READS after a READ with MATCH:Q_EQ that returns a “Record not found” error (\$ERR_RNF) is different on OpenVMS and non-OpenVMS systems and therefore should not be assumed. Use READ or FIND to re-establish context on all systems.

See also

[FIND on page 4-57.](#)

NOFILL – Prevent the data area from being padded with blanks

NOFILL

Discussion

If the file record size is less than the size of the data area that is passed to the READ or READS statement, by default, READ or READS will place the record into the data area and then fill the remainder of the data area with spaces. Specifying the NOFILL qualifier causes the data to be transferred to the user's data area without padding the data area, leaving the remainder of the user's data area unchanged.

See also

- ▶ [READ on page 4-145.](#)
- ▶ [READS on page 4-150.](#)

NUMREC – Preallocate a number of records

NUMREC : *number*

Arguments

number

The number of records to preallocate for the file. (**n**)

Discussion

The NUMREC qualifier preallocates a number of records for the file being opened. If the file is not opened in output mode, this qualifier is ignored, except on OpenVMS or unless system option #15 is set.

See also

[OPEN on page 4-124.](#)

OPTIONS – Specify runtime options

OPTIONS: "*runtime_option*[, ...]"

Arguments

runtime_option

An expression, in quotation marks, that contains one or more of the following options separated by commas: **(a)**

- /encrypt** Specify that a client application should request data packet encryption from the server if the server is enabled with “slave” encryption. If the file is not a remote file via *xfServer*, this option is ignored. (It is also ignored if master encryption is enabled on the server, because all data is encrypted anyway.) See “[Using Encryption](#)” in the “Configuring *xfServer*” chapter of the *Installation Configuration Guide* for more information.
- /ex** Specify exclusive access for the file being opened.
- /fl=length** Define the record format as fixed-length with *length* as the specified record size.
- /io=mode[:submode]** Redefine the OPEN processing mode, where *mode* is the I/O mode and *submode* is the I/O submode in which you want to open the file. This option overrides the OPEN mode specified. The options **/io=O** and **/io=A** open a file for exclusive access, equivalent to the compile-time output and append modes.
- /nl** Specify no record locking is to be performed on the file being opened.
- /si=size** Set the initial allocation size of the file to *size*.
- /share=value** See [SHARE](#) on page 4-239.
- /sequential** Optimize sequential record access. On Synergy ISAM, this option optimizes sequential read performance but may cause adjacent records stored by other users during your read to be missed. In versions prior to 6.1, this was the default behavior.
- /stream** Specify that the file type is stream when opened for output on an OpenVMS server system.
- /tf=filename** See [TEMPFILE](#) on page 4-241.

WIN	
/guiwnd	See GUIWND on page 4-221.
WIN, UNIX	
/cache	Open an ISAM file with cached access. See #3 in the “System Options” chapter of <i>Environment Variables and System Options</i> for details.
/po	Specify the start position. See POSITION on page 4-233.
UNIX	
/nodelay	Open a serial device with the UNIX O_NDELAY option. This causes the OPEN to not be blocked until the device is ready or available. The O_NDELAY flag is turned off after the device is opened so that I/O to the device does not automatically generate a “Failure during I/O operation” error (\$ERR_IOFAIL).
VMS	
/alloc=value	See ALLOC on page 4-208.
/bufnum=value	See BUFNUM on page 4-211.
/bufsiz=value	See BUFSIZ on page 4-212.
/deq=value	See DEQ on page 4-214.
/rectype=value	See RECTYPE on page 4-236.

Discussion

The OPTIONS qualifier enables you to specify one or more runtime options. These runtime options are evaluated when the OPEN is processed during program execution. Each runtime option must be enclosed in quotation marks.

The runtime options set by the OPTIONS qualifier take precedence over the other I/O qualifiers.

See also

[OPEN](#) on page 4-124.

Examples

```
open(ch, o, "fred", OPTIONS:"/stream")
```

POSITION – Define the file position

POSITION : *pos_spec*

Arguments

pos_spec

One of the following identifiers or corresponding values: (**n**)

Q_IGNPOS or **-1** Ignore the POSITION qualifier.

Q_FIRST or **0** Position to the first record in the file (relative to the key of reference, if it's an ISAM file).

WIN, UNIX

Q_LAST or **1** Position to the last record in the file (relative to the key of reference, if it's an ISAM file).

Q_EOF or **2** Position to the end of the file without establishing a current record.

Q_BOF or **3** Position to the beginning of the file without establishing a current record (valid on FIND only).

Discussion

The POSITION qualifier sets the file position before performing the I/O operation.

By default, the OPEN statement positions to the first record in the file. OPEN only recognizes the Q_FIRST and Q_EOF identifiers.

The FIND statement only recognizes the Q_FIRST, Q_LAST, Q_BOF identifiers.

The READ statement only recognizes the Q_FIRST and Q_LAST identifiers.

NON-ISAM

The WRITE statement only recognizes Q_FIRST, Q_LAST, and Q_EOF identifiers.

See also

- ▶ [FIND on page 4-57.](#)
- ▶ [OPEN on page 4-124.](#)
- ▶ [READ on page 4-145.](#)
- ▶ [WRITE on page 4-198.](#)

RECSIZ – Specify a record size for a file

NON-ISAM

RECSIZ : *record_size*

Arguments

record_size

One of the following values: (**n**)

- n* Set the record size to *n* for the file being opened.
- 0** Ignore the RECSIZ qualifier.
- 1** Set the record size to the length of the first record in the file.

Discussion

The RECSIZ qualifier sets a record size for the file being opened.

WIN, UNIX

If you use the RECSIZ qualifier, the records of the file are accessed as fixed-length records of the specified size. By default, the record size of the file being opened is equal to the size of the destination data area on the first I/O operation. Before the first I/O, the record size is undefined.

VMS

Record_size is the record size for the file being opened; the values 0 and -1 are not valid. If you're creating a relative file on OpenVMS, the RECSIZ qualifier is required. If you're opening a relative file for input, RECSIZ is checked and an "Invalid record size" error (\$ERR_IRCSIZ) occurs if the values do not match. If you are creating a sequential file, this qualifier sets the maximum record size allowed for the file, unless system option #35 is set. If option #35 is set, Synergy Language accesses the file's records as fixed-length.



If the file is empty, do not use RECSIZ:-1 to set the record size to the length of the first record in the file.



If the records in the file possibly contain integer data and you want to use RECSIZ:-1, you can put a "dummy" record at the beginning of the file that does not have any integer data. This enables Synergy Language to determine the correct record size for the file.

See also

[OPEN on page 4-124.](#)

RECTYPE – Define the type of records to write to a file

VMS

RECTYPE : *record*

Arguments

record

One of the following values: (**n**)

0	Undefined record format
1	Fixed-length records
2	Variable-length records
3	VFC (variable-fixed control)
4	Stream format records
5	Stream format records with line-feed carriage control
6	Stream format records with carriage-return carriage control

Discussion

The RECTYPE qualifier enables you to specify the type of records that are written to a file.

By default, OpenVMS creates variable-length records for sequential and print files, and fixed-length records for ISAM, relative, and block mode files.

See also

[OPEN on page 4-124.](#)

RFA – Locate record with the specified RFA

RFA:match_rfa

Arguments

match_rfa

One of the following:

rfa The RFA. (**a6**)

grfa The global RFA (GRFA), which must be a 4-byte record CRC and a 6-byte RFA of the current record. (**a10**)

Discussion

The RFA qualifier causes the FIND, READ, and WRITE statements to position to the specified RFA. However, the RFA qualifier on an UNLOCK statement allows a specified lock to be released.

Rfa is an **a6** variable that has been obtained either by the GETRFA qualifier or the GETRFA subroutine.

Using GRFAs can help you implement optimistic locking. (See [“Optimistic locking technique” on page 4-244](#) for more information.)

If you’ve used GETRFA:*grfa* on a previous READ or READS statement and then saved off the GRFA that is returned, a READ or READS with the RFA qualifier can attempt to reacquire the same record. (For a relative file, you can use WRITE to write a record by GRFA without rereading it.) If a record has changed between reads (or before it is written), a “Record not same” error (\$ERR_RECNOT) will result because the CRC no longer matches. Thus, on a READ the record will still be read, but on a WRITE the record will not be written.



RFA:*grfa* cannot be used on the FIND, READ, and WRITE statements for file types other than ISAM and relative. The CRC portion of the GRFA is ignored on the FIND statement, the UNLOCK statement, and the POSRFA routine for ISAM and relative files.

The following built-in types are available when declaring an RFA or GRFA variable.

- ▶ D_RFA_TYPE (defined as **a6**)
- ▶ D_GRFA_TYPE (defined as **a10**)

See also

- ▶ [FIND on page 4-57.](#)
- ▶ [READ on page 4-145.](#)
- ▶ [WRITE on page 4-198.](#)

- ▶ [GETRFA on page 4-218.](#)
- ▶ [“Record Locking” on page 4-243.](#)
- ▶ [“Record File Addresses \(RFAs\)” on page 4-246.](#)

SHARE – Control file access

SHARE : *share_spec*

Arguments

share_spec

One of the following identifiers or corresponding values: (n)

- Q_EXCL_RW** or **0** Allow only exclusive read and write access for the file. With this option set, other users cannot access the file.
- Q_EXCL_RO** or **1** Allow read-only access for the file. With this option set, other users can read but not write to the file.
- Q_NO_EXCL** or **2** Allow other users to both read and write to the file.

Discussion

The SHARE qualifier specifies whether a file can be accessed by other users.

To achieve the exclusive access of Q_EXCL_RW and Q_EXCL_RO, the systems file permissions must allow write access to the file.

The default *share_spec* value depends on the mode in which the file was opened: If the file was opened in input or update mode, both read and write access is allowed by default (that is, Q_NO_EXCL or 2); however, if the file was opened in output or append mode, only exclusive access is allowed (that is, Q_EXCL_RW or 0).

UNIX

Except on Sun systems, setting the LOCK:Q_NO_LOCK qualifier overrides an exclusively locked file that was opened using an exclusive SHARE qualifier (either Q_EXCL_RW or Q_EXCL_RO).

Except on Sun systems with system option #33 set, opening a file with Q_EXCL_RO generates a “File in use” error when another process has the same file open with Q_EXCL_RO, regardless of the OPEN mode.

VMS

Opening a file Q_EXCL_RW minimizes OpenVMS lock activity.

The SHARE qualifier has no effect on an OPEN statement where the OPEN mode specified is either append (A) or output (O).

See also

- ▶ [OPEN](#) on page 4-124.
- ▶ [INTRAFILELOCKS](#) in the “Environment Variables” chapter of *Environment Variables and System Options*.
- ▶ [#33](#) in the “System Options” chapter of *Environment Variables and System Options*.

TEMPFILE – Define an intermediate file for output mode

TEMPFILE[:*temp_spec*]

Arguments

temp_spec

(optional) The file specification for the intermediate file to be used for output. It can have any one of the following formats: (a)

TEMPFILE: "*logical*:"

TEMPFILE: "*file*"

TEMPFILE: "*logical:file*"

Discussion

The TEMPFILE qualifier defines an intermediate (or temporary) file to be used for opening a file in output mode. The intermediate file's directory can be specified in the form of a logical name or a directory specification.

If *temp_spec* is not specified, the intermediate filename is **DLnnnnnn**, where *nnnnnn* is six unique characters, and the file is created in the current directory on UNIX and OpenVMS or in one of the following directories on Windows:

- ▶ The directory specified by the TMP environment variable
- ▶ The directory specified by the TEMP environment variable if TMP is not defined
- ▶ The Windows directory if neither TMP nor TEMP is defined

Make sure sufficient free disk space is available wherever intermediate files are created. Note that TEMPFILE always creates a unique filename if no filename is specified.

When the channel is closed, the intermediate file is renamed to the filename specified in the OPEN statement. If this filename is blank, the temporary file is not renamed on the CLOSE.

Using the TEMPFILE qualifier instructs Synergy Language to supersede an existing file on the CLOSE instead of the OPEN.

VMS

Temp_spec is ignored unless it is blank because a new version is used for a new temporary file of the same name. Be careful of directory version limits when using the TEMPFILE qualifier.

See also

- ▶ [OPEN on page 4-124.](#)
- ▶ [SORT on page 4-169.](#)

WAIT – Specify wait time for an I/O statement to finish processing

`WAIT:wait_spec`

Arguments

`wait_spec`

One of the following identifiers or corresponding values: (**n**)

Q_NOWAIT or **0** Don't wait if I/O processing can't be completed (unless system option #35 is set; see below for details).

Q_WAIT or **-1** Wait until I/O processing is completed.

n Wait up to *n* number of seconds for I/O processing to complete.

Discussion

The WAIT qualifier specifies how long to wait for an I/O statement to complete processing.

For files, I/O processing can't complete if the record is locked. If you don't specify the WAIT qualifier, Synergy Language defaults to the setting of the DFLAG subroutine runtime option flag 3. If you do specify WAIT, it overrides the DFLAG setting.

If a record is locked and you specify WAIT:Q_NOWAIT, you'll get a "Record is locked" error (\$ERR_LOCKED). If a record is locked and you specify WAIT:Q_WAIT or WAIT:*n*, program execution is suspended until the record is available or until *n* seconds have passed. If the record is not available within *n* number of seconds, you'll get a "Record is locked" error (\$ERR_LOCKED).

For terminals, WAIT:Q_WAIT is the default. If you specify WAIT:Q_WAIT or WAIT:*n*, the program execution is suspended until input is received from the terminal. If input is not received within *n* number of seconds, you'll get a "Terminal input operation" error.

If system option #35 (VAX DIBOL-compatible functionality) is set and you specify a value of zero for `wait_spec`, it indicates "wait forever" instead of "don't wait."

See also

- ▶ [DFLAG on page 9-98.](#)
- ▶ [#35](#) in the "System Options" chapter of *Environment Variables and System Options*.
- ▶ [ACCEPT on page 4-11.](#)
- ▶ [FIND on page 4-57.](#)
- ▶ [GETS on page 4-77.](#)
- ▶ [READ on page 4-145.](#)
- ▶ [READS on page 4-150.](#)

Record Locking

Because Synergy Language programs function in multiuser environments, files opened in update mode are automatically subject to record-level locking controls. No record locking is performed when a file is opened in input mode.

When a record is retrieved from a file opened in update mode, the data record is locked to prevent access to the record by another channel. Synergy Language provides two types of locking that can be used at the same time on a channel: automatic and manual. Automatic locking releases a lock before locking another record; only one automatic lock exists on a channel. Manual locking locks a record until it is explicitly unlocked; multiple records can be manually locked on the same channel.



Manual record locking is available for ISAM files only on UNIX and Windows. OpenVMS allows manual locks on RMS ISAM and relative files.

The default record locking state is automatic locking. If you specify the LOCK:Q_NO_LOCK on the OPEN statement to change the default record locking behavior, no record locking is performed regardless of whether the LOCK qualifier is specified on any following I/O statements.

WIN, UNIX

Using LOCK: Q_NO_LOCK on an OPEN operation can cause corrupt data to be read as locking is required to guarantee index consistency when reading the index. Using Q_NO_LOCK on a READ operation on an update channel acts the same as if the channel were opened for input.

An automatic record lock is released when one of the following conditions occurs:

- ▶ A different record is locked by a READ, FIND, READS, GET, or GETS statement.
- ▶ A STORE or WRITES statement is used.
- ▶ The FREE subroutine is called, which releases all automatic and manual locks.
- ▶ The locked record is deleted with the DELETE statement. (The DELETE statement is used with ISAM files only.)
- ▶ An UNLOCK statement unlocks the channel.
- ▶ The file is closed with the CLOSE statement.

A manual record lock is released when one of the following conditions occur:

- ▶ The FREE subroutine is called, which releases all automatic and manual locks.
- ▶ The locked record is deleted with the DELETE statement.
- ▶ An UNLOCK statement explicitly unlocks the record using the RFA I/O qualifier.
- ▶ The file is closed with the CLOSE statement.

When using manual locking on a file with data compression or variable-length records, all record updates (using the `WRITE` statement) must also update the saved RFA for that record (using `GETRFA` on the `WRITE` statement) because the record can be physically moved, thereby changing its file position. Alternatively, you can add the `STATIC_RFA` file option, which guarantees an RFA that remains static. (We recommend the latter.)

You can override the default automatic record locking behavior by specifying the `LOCK` qualifier. For example, if a file is opened in update mode, you can use the `LOCK:Q_NO_LOCK` qualifier to read a record without locking it, instead of having to perform an `UNLOCK` after the record is automatically locked.



If you're reading a record that could be locked and you don't plan to immediately update or delete it (for example, you're displaying the record for examination), we suggest that you either use the `LOCK:Q_NO_LOCK` qualifier or execute an `UNLOCK` statement to allow other users to access the record.

WIN, UNIX

To retrieve information about locked records, you can run the **chklock** utility. For more details, see [chklock](#) in the “Synergy DBMS” chapter of *Synergy Language Tools*. Depending on which options you specify, **chklock** returns the following types of information:

- ▶ The first byte position of every record that is locked in a file
 - ▶ The process ID that holds each lock (UNIX only)
 - ▶ The length of each lock (UNIX only)
 - ▶ Whether the file is opened in input or update mode (UNIX only)
 - ▶ Whether the file is locked for exclusive access
-

Synergy Language record-locking behavior may differ on your system. Refer to the [Professional Series Portability Guide](#) for information about record-locking behavior on your system.

Optimistic locking technique

Synergy Language also makes it possible for you to take advantage of optimistic record locking. Optimistic locking is a technique that cuts down on lock contention by shortening the duration of record locks during update operations. An application using an optimistic locking model during transactions will have two types of I/O operations: a get operation and an update operation. The get operation either reads a record without a lock (`LOCK:Q_NO_LOCK`) or reads it with a lock and then immediately unlocks it. The update operation occurs at the end of the transaction, by rereading the same record with a lock and then immediately writing out the new record and unlocking it.

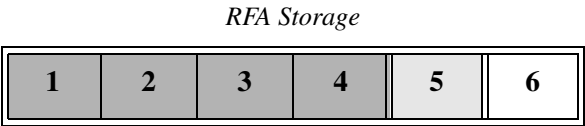
Determining whether or not the record was changed is a key requirement of optimistic locking. Using GRFAs is one way to fill that requirement. When you initially read the record, you should save the GRFA that is returned, so that subsequently when you reread the record in your update phase, you can compare the current GRFA with the one that you saved. If they are different, another process has modified the record, and you can adjust your processing as appropriate. (See [“Record File Addresses \(RFAs\)” on page 4-246](#) for more information about GRFAs.)

Record File Addresses (RFAs)

WIN, UNIX

An RFA is a binary identifier that uniquely identifies the location of a record in a file. It can be used to access previously read records without the use of keys or record numbers. On ISAM files, access is direct without having to read the index.

An RFA in 32-bit form is stored as an alpha or decimal variable consisting of six bytes, as shown in the following diagram:



The first four bytes are the absolute position in the file and are represented as a portable integer. The fifth byte is the RFA use counter, which is a byte that is incremented each time an RFA position is restored by a STORE or WRITE statement. The sixth byte is currently unused.

On terabyte files (created with the TBYTE option to the **bldism** utility or the ISAMC routine), the first five bytes are the file position represented as a portable integer, and the sixth byte is the RFA use counter.

Static RFAs

ISAM

When creating a file, you can also specify whether each record in that file has a unique, or static, RFA throughout the life of that file. Synergy Language returns the RFA to your program each time the record is retrieved or accessed. Your program can either ignore the RFA or keep it for subsequent, direct retrieval of records.

Static RFAs are primarily intended for applications that rely heavily on RMS-compatible RFA access, use manual locks together with record compression or variable-length records, or use *x/ODBC*.

Normally, Synergy Language reuses released data space, thereby eliminating the need to reorganize your ISAM files. Files that use static RFAs, however, may contain deleted data space that is not reusable; you can use the **isutl** utility to find out how many unused records exist in your data partition. (See **isutl** in the “Synergy DBMS” chapter of *Synergy Language Tools* for more information.) If you try to retrieve or access a deleted record by its saved RFA, you’ll get a “Deleted Record” error (\$ERR_DELREC).

If you don't use static RFAs, the following situations may occur:

- ▶ An RFA may change on a WRITE statement.
- ▶ A new record may have the same RFA as a record whose use count exceeds 128.
- ▶ Unlocking a manual lock by RFA after an RFA has changed will either not work or will unlock the wrong record.
- ▶ Use of temporary indexes by *xj*ODBC may result in incorrect data being returned.

When you want to free up any unusable file space used by the static RFAs for a file, you can use **isutl -ro** to reorganize an ISAM file. (See **isutl** in the “Synergy DBMS” chapter of *Synergy Language Tools* for more information.) If you reorganize your ISAM file, the static RFAs for that file are no longer valid.



Using static RFAs adds six bytes of additional storage space per record.

Global RFAs (GRFAs)

A GRFA is a 10-byte binary identifier that uniquely identifies the location and contents of a record in a file. A GRFA consists of a 4-byte record CRC and a 6-byte RFA. GRFAs are used to access previously read records without the use of keys or record numbers and also detect when the record has changed.

Because the GRFA includes a record CRC, it provides a way for transactions to be validated before being committed. A CRC is derived from the record in order to detect changes; any change to the record generates a different CRC.

5

Preprocessor and Compiler Directives

Types of Directives 5-2

Discusses preprocessor directives and compiler directives and how they are both used.

Directive Syntax 5-3

Contains specification pages for the following directives:

.ALIGN – Align the data location counter to a boundary	5-4
.DEFINE – Define an identifier.....	5-7
.IDENT – Include a module identifier string.....	5-13
.IF-.ELSE-.ENDC – Perform conditional processing	5-16
.IFDEF-.ELSE-.ENDC – Perform conditional processing.....	5-18
.IFNDEF-.ELSE-.ENDC – Perform conditional processing.....	5-20
.IFT, .IFF, .IFTF – Control conditional processing.....	5-22
.INCLUDE – Include external source code.....	5-25
.LIST – Control the listing of source code	5-35
.NODEBUG – Ignore breakpoints in debug mode.....	5-36
.NOLIST – Control the listing of source code.....	5-37
.NOPROTO-.PROTO – Turn prototype generation off and on.....	5-38
.PAGE – Advance program listing to top of new page	5-39
.REGION-.ENDREGION – Specify a collapsible block of code	5-40
.START – Control the listing of source code	5-41
.TITLE – Specify the listing page header.....	5-43
.UNDEFINE – Remove definition of replacement identifier.....	5-44

Types of Directives

All preprocessor and compiler directives begin with a period and must be the first non–white space element on a logical line.

Preprocessor directives

A preprocessor directive is an instruction for the preprocessor, which is executed before any code is actually compiled. They can precede or follow any logical line, and they apply to the remainder of the source file until other compilation control directives are specified that change the instructions to the compiler.

The preprocessor directives are

- ▶ `.LIST`
- ▶ `.NOLIST`
- ▶ `.PAGE`
- ▶ `.START`
- ▶ `.TITLE`
- ▶ `.UNDEFINE`

Compiler directives

Compiler directives instruct the Synergy compiler and are processed only when a program is compiled. They classify and delimit program structure and are position-dependent within program source code. They are not included in the resulting executable program.

The compiler directives are

- ▶ `.ALIGN`
- ▶ `.DEFINE`
- ▶ `.IDENT`
- ▶ `.IF-.ELSE-.ENDC`
- ▶ `.IFDEF-.ELSE-.ENDC`
- ▶ `.IFNDEF-.ELSE-.ENDC`
- ▶ `.IFT, .IFF, .IFTF`
- ▶ `.INCLUDE`
- ▶ `.NODEBUG`
- ▶ `.NOPROTO-.PROTO`

Directive Syntax

The following pages contain detailed specifications for the preprocessor and compiler directives.

.ALIGN – Align the data location counter to a boundary

`.ALIGN[ boundary]`

Arguments

boundary

(optional) One of the following keywords or an expression that specifies the alignment boundary:

BYTE 1 byte

WORD 2 bytes

LONG 4 bytes

QUAD 8 bytes

PAGE 512 bytes

expression An expression in the range 0 through 9 (**n**)

Discussion

The `.ALIGN` compiler directive aligns the data location counter to a specified boundary for the next data element to be processed. It can help increase an application's speed by properly aligning integer variables. You can specify `.ALIGN` before a record or single variable.

`.ALIGN` looks at the address of the next memory location and adds as many filler bytes as necessary to cause the next memory location to fall on the desired boundary. If you specify the **BYTE**, **WORD**, **LONG**, **QUAD**, or **PAGE** keyword, the data location counter is aligned to an address that is the next multiple of the number of bytes for that keyword. For example, if the next memory location is 21 and longword alignment is desired, three bytes are added to update the next memory location to 24 (the next higher number that is evenly divisible by 4), thus placing the next variable on a longword-aligned memory boundary.

The default is to align to the natural “int” size on your machine. This is usually longword alignment (4-byte boundaries), except on 64-bit machines, where it is quadword alignment (8-byte boundaries).

If you specify *expression*, the data location counter is aligned to an address that is the value of 2 raised to the power of *expression*.



If `.ALIGN boundary` is used on any system, the alignment is only guaranteed up to the natural register size of that machine. If *boundary* is larger and memory alignment cannot be guaranteed, a warning (NOTALIGNED) is issued.

General rules of thumb are as follows:

- ▶ **i1** variables need not be aligned.
- ▶ `.ALIGN WORD` should precede the first of a sequence of **i2** variables.
- ▶ `.ALIGN LONG` should precede the first of a sequence of **i4** variables.
- ▶ On OpenVMS, `.ALIGN` without a qualifier (which aligns to a quadword boundary) is recommended.
- ▶ For most other RISC processors (IBM RS/6000, MIPS), `.ALIGN QUAD` should be used, and all integers should be aligned.

If you specify `.ALIGN` before a declaration that doesn't define data (such as an overlaid data declaration statement, an `EXTERNAL COMMON` statement, an `EXTERNAL LITERAL` statement, or any field in an `EXTERNAL COMMON` declaration), the compiler generates an "Alignment not performed" warning (NOALIGN).



The `.ALIGN` directive can cause empty bytes to be allocated between successive data fields. Be careful when redefining the record that contains the `.ALIGN` directive or when subscribing across an `.ALIGN` directive, because the empty bytes can be accessed!

Examples

In the following example, **data2** is longword-aligned.

```
record
    data1          ,d3
    .align LONG
    data2          ,d8
```

In the next example, **data5** is aligned to a page boundary and **data6** through **data8** are aligned at the 128-byte boundary with room for future growth.

```
.align PAGE
record
    data5          ,a90
    .align 7
    data6          ,a90
    .align 7
    data7          ,a90
    .align 7
    data8          ,a90
```

It's important to understand the difference between the placement of the .ALIGN statement in these two examples:

record alpha	record alpha
fld1 ,a1	fld1 ,a1
record beta	.align long
.align long	record beta
fld2 ,i4	fld2 ,i4
record gamma, X	record gamma, X
fld3 ,i4	fld3 ,i4

The example on the left causes **fld2** to be aligned on a longword boundary and allows **fld2** to be accessed as efficiently as possible. The problem with this approach is not always apparent. The .ALIGN LONG has increased the size of record **beta**, and **fld3** no longer overlays **fld2** properly. You would see this easily if ^SIZE(**beta**) were displayed. In this case, the size of record **beta** is 7, and only the fourth byte of **fld3** coincides with **fld2**, which is not the desired effect.

When the .ALIGN LONG is moved before the record, as in the example on the right, the size of record **beta** is 4 as desired. But what about the size of record **alpha**? In the left example, the size of record **alpha** is quite obviously 1. If we displayed ^SIZE(**alpha**) in this example, we'd see it's still 1. So now we wonder, is **fld2** really aligned as desired, and where are the 3 bytes needed for alignment? The answer is that **fld2** *is* aligned, and the extra 3 bytes have been added to the data division between the two records and not as part of either record. ^SIZE(**alpha**) is 1, ^SIZE(**beta**) is 4, ^SIZE(**gamma**) is 4, and **fld3** and **fld2** coincide as desired.

.DEFINE – Define an identifier

`.DEFINE identifier [, replacement]`

or

`.DEFINE macro (arg, ...) [replacement2]`

Arguments

identifier

The name of the replacement identifier to be defined.

replacement

(optional) The text to replace the identifier. The comma is optional unless *replacement* begins with a “(”.

macro

The name of the macro to be defined.

arg

One or more replacement arguments.

replacement2

(optional) The text to replace the macro.

Discussion

The `.DEFINE` compilation control directive defines an identifier and optionally assigns it replacement text. You can define parameterized macros using the second form of the `.DEFINE` directive as shown above. See [“Parameterized macros” on page 5-8](#) for more information.

Whenever *identifier* is encountered within a subsequent logical line, *replacement* is substituted for *identifier*. The specified replacement text begins with the first non–white space character after the comma (or after *identifier* if the comma is omitted) and ends with the last non–white space character in the logical line, excluding comments. If *replacement* is not specified, *identifier* is removed from each following logical line in which it is encountered.

Replacement can contain another replacement identifier name. If that identifier hasn’t been processed within the current replacement operation, it is also replaced. (See example 5 in the Examples section below.)

Identifier is not replaced with the specified text if the identifier is encountered within an `.IFDEF`, `.IFNDEF`, or `.UNDEFINE` compiler directive line. (See example 2 in the Examples section below.)

By default, if a `.DEFINE` directive that redefines a previously defined identifier is encountered, the compiler generates a “Symbol already defined” warning (SYMDEFD), and the second definition supersedes the previous one. You can change this default by using the `.DEFINE` warning compiler

option. If you specify this option, the compiler does not generate a warning each time you redefine a symbol or `.UNDEFINE` something that isn't already defined. (See “[Invoking the compiler](#)” in the “Building and Running Synergy Language Programs” chapter of *Synergy Language Tools* for more information about this compiler option.)

Replacement identifiers remain defined until the entire file is compiled, unless the **-qrelaxed:end** compiler option is set. If **-qrelaxed:end** is set, replacement identifiers are cleared at the end of the routine, when the `END` statement is encountered, instead of at the end of the file.



We recommend that you uppercase all replacement identifiers defined by the `.DEFINE` directive so that you can easily differentiate them from fields. We also recommend that you use the `.DEFINE` directive to define constant values instead of hard-coding them throughout your application. The `.DEFINE` directive enables you to more easily and reliably develop and maintain your applications by localizing constant values.

Parameterized macros

Each *arg* is a replacement identifier and only has meaning within the scope of a single definition (in other words, it must be unique within all the arguments of a given definition, but can be duplicated across definitions).

The *replacement2* content starts with the first nonblank character following the closing “)” of the arguments, and continues until the last nonblank character on the logical line.

When a parameterized macro is invoked within code, the text associated with each of the defined arguments is inserted into the `.DEFINE`'s replacement text. Compilation of the line containing the parameterized define then continues. Nested parameterized macros of the same name are not permitted.

Note that continuation lines within a parameterized macro definition are allowed, but each continuation line reduces to a single space in the replacement text. Thus, the replacement text for

```
.define a(b) x (b)
```

and

```
.define a (b)  
  & x (b)
```

are not the same. The replacement text for the first is “x (b),” whereas it is “<space>x (b)” for the second. This generally is not a problem, except where a definition will be used as a function.

Using the above definitions, the code

```
var = %a(num)
```

generates an error using the second definition because it expands to

```
var = % x(num)
```

which is invalid syntax. (A space is not allowed between the “%” and the “x”.) This can be solved by making sure a space is valid in an expansion. For example, the second form above can be fixed by using this form:

```
.define a (b) x
    & (b)
```

because the expansion of

```
var = %x (num)
```

is valid syntax.

When invoking a macro, *arg* is considered to be the first nonblank character following the leading “(” or “,” up to the last nonblank character preceding the trailing “,” or “)”.

There is a special case of the above. If the first nonblank character encountered is a “<”, the replacement text begins with the first character following the “<” and continues until the last character preceding the matching “>”. In other words, the surrounding angle brackets define the text but are stripped when used. Thus,

```
.define abc(arg) xyz(D_STORE, arg)
```

when used by

```
abc(<1,3, "hello">)
```

expands to

```
xyz(D_STORE, 1,3, "hello")
```

String concatenation is supported in the replacement text by using the following rule: If an unquoted accent grave (‘) immediately precedes or follows a replacement argument, it is stripped from the final text form. In a quoted string, two accents grave preceding and a single one following a replacement argument are stripped. For example:

```
.define BASEREG(id,cls) id = `%cls`_cid ("DEF_``cls`_REC")
```

when used by

```
BASEREG(class_id, DSPOBJ)
```

expands to

```
class_id = %DSPOBJ_cid("DEF_DSPOBJ_REC")
```

Macro expansions within a string work only at a single level. For example,

```
.define MYNAME earl
.define name(t) "My name is ``t`"
main
record
    earl, a*, "Earl"
proc
    open(1,o,"TT:")
    writes(1,name(MYNAME)    ; This will display My name is MYNAME
end
```



We recommend that you follow these guidelines when using parameterized macros:

- ▶ Because white space and continuations are replaced exactly as defined in the macro, don't insert spaces or continuations before a function name in a .DEFINE definition.
- ▶ Include a comma if you are defining a literal value. If you are defining a parameterized macro, do not include a comma.
- ▶ When referencing a parameterized macro, do not continue a line in the middle of a macro argument. (For example, if the argument to a macro is also a function call, the entire function call should be on one line.) You can continue *between* arguments but not within an argument.
- ▶ If the macro declaration contains parentheses after the name, arguments must be specified. For example, the following is invalid:

```
.define MACRO() expansion
```
- ▶ If you want to substitute a macro within a string that is part of another macro expansion, you have to break it into expressions that do string concatenation. For example in the above example, you can define the macros like this:

```
.define MYNAME earl
.define name(t) "My name is "+t
```

Examples

1. In this example, two simple text replacement operations use the .DEFINE directive.

```
.define TTCHN      ,1                ;Text replacement does not
.define FLAGS      ,xcall flags      ; include these comment lines.

proc
    FLAGS(4020)                      ;This statement becomes
                                      ; xcall flags(4020).
    open(TTCHN, o, "tt:")            ;This statement becomes
                                      ; open(1, o, "tt:").
```

2. In this example, TTCHN is not replaced in the .IFNDEF directive (in other words, the directive does not become .IFNDEF 1). Since TTCHN is already defined, the .DEFINE directive within the .IFNDEF conditional is not processed.

```
.define TTCHN      ,1
.
.
.
.ifndef TTCHN
.define TTCHN      ,1                ;This statement is not processed
.endc
```

3. In this example, the compiler generates a “Symbol already defined” warning (SYMDEFD) on the second .DEFINE statement unless

- ▶ you specified the .DEFINE warning compiler option to suppress these warnings.
- ▶ you ended the routine with END instead of .END and specified **-qrelaxed=end** on the command line.

If you ended the routine with END and specified **-qrelaxed=end**, the identifier ONE is undefined after the main routine is compiled, and no error occurs.

```
.define ONE      ,1

proc
.
.
.
.end

.define ONE      ,1
subroutine sub

proc
.
.
.
endsubroutine
```

4. This example shows how replacement text can span across more than one physical line. See [“Continuation lines” on page 4-6](#) for more about the rules for splitting alpha literals across continuation lines.

```
.define SYNERGEX      ,"Synergex International
    & Corporation"

; The replacement text is "Synergex International Corporation".
```

5. This example shows recursive text replacement. In the WRITES statement, the first SYNERGEX is replaced with EMPLOYER, then EMPLOYER is replaced with SYNERGEX. Since SYNERGEX has already been processed once in this replacement operation, it is not be replaced again. The second SYNERGEX (in the WRITES statement) is a separate replacement operation; therefore, it is replaced just as the first SYNERGEX was replaced. "SYNERGEXSYNERGEX" is written to the terminal.

```
record
    synergex ,a8,      "SYNERGEX"

.define TTCHN          ,1
.define SYNERGEX        ,EMPLOYER
.define EMPLOYER        ,SYNERGEX

proc
    open(TTCHN, o, "tt:")
    writes(TTCHN, SYNERGEX + SYNERGEX)
end
```

6. In this example, the parameterized macro SUBTOTAL is used to output various subtotals following their descriptions.

```
.define SUBTOTAL(desc, amount) writes(pchan,
    &          "Subtotal for ``desc`: " + %string(amount))

SUBTOTAL(Apples, apples_total)
SUBTOTAL(Oranges, oranges_total)
SUBTOTAL(Mangoes, mangoes_total)
```

Resulting in the expansion:

```
writes(pchan, "Subtotal for Apples: " + %string(apples_total))
writes(pchan, "Subtotal for Oranges: " + %string(oranges_total))
writes(pchan, "Subtotal for Mangoes: " + %string(mangoes_total))
```

Since the names of the total variables are prefixed with their descriptions, the macro can be further simplified:

```
.define SUBTOTAL(desc) writes(pchan,
    &          "Subtotal for ``desc`: " + %string(desc`_total))

SUBTOTAL(Apples)
SUBTOTAL(Oranges)
SUBTOTAL(Mangoes)
```

which results in the semantically identical expansion:

```
writes(pchan, "Subtotal for Apples: " + %string(apples_total))
writes(pchan, "Subtotal for Oranges: " + %string(oranges_total))
writes(pchan, "Subtotal for Mangoes: " + %string(mangoes_total))
```

.IDENT – Include a module identifier string

`.IDENT string`

Arguments

string

The module identifier string to be included. This must be an alpha expression.

Discussion

The .IDENT compilation control directive defines a module identifier string to be included in an object record of a routine's object module. Only the first 15 characters specified in *string* are used.

Only the first .IDENT encountered before or within a routine is used. Additional .IDENT directives within a given routine will generate a warning.

If you specify the .IDENT directive in a file that contains more than one routine, all routines following the routine with the .IDENT directive use the same module identifier string.

The Synergy librarian displays the .IDENT string when the contents of a library are listed.

You must specify the .IDENT compiler directive for a routine before the compiler emits the object module header. If the compiler encounters an .IDENT compiler directive after it has already written the object module header, it generates an “.IDENT ignored” error (IDIGN). (This may happen if you specify .IDENT at the end of a large routine.)

UNIX

An .IDENT string is called a “what” string. If you include an .IDENT string in your object file, you can enter:

```
what file.dbo
```

at your UNIX prompt to get the .IDENT information. “What” strings are available on UNIX systems that have Source Code Control System (SCCS). For more information about “what” strings, refer to your UNIX manual.

VMS

.IDENT strings are called “module version” strings. If you include an .IDENT string in your object file, you can enter the following at your OpenVMS prompt to get the .IDENT information:

```
ANALYZE/OBJ FILE.OBJ
```

For more information about “module version” strings, refer to your OpenVMS Linker manual.

Examples

In the following example, the string “V5.1 - 040392” is included in the object file for the main routine, and “V5.1 - 031092” is included in the **sub** subroutine’s object file.

```
main
.ident "V8.1 - 040392"
proc
    .
    .
    .
endmain

subroutine sub
.ident "V8.1 - 031092"
proc
    .
    .
    .
endsubroutine
```

If the second .IDENT directive is not specified, the first string “V5.1 - 040392” is included in the object files for both routines.

In our example, if you enter

```
what file.dbo
```

at the UNIX prompt, the following information is printed:

```
V8.1 - 040392
V8.1 - 031092
```

If you enter:

```
analyze/obj/mhd file.obj
```

at the OpenVMS prompt on an Alpha machine, the following information is printed. (On an I64, the fifth line of text would say, “This is an OpenVMS IA64 (Elf format) object file.”)

```
Analyze Object File
DKA600: [NIGEL]FILE.OBJ;1
ANALYZ A05-19
```

20-FEB-2006 17:05:41.51 Page 1

```
analyze/obj/mhd file.obj
```

```
This is an OpenVMS Alpha object file
```


1. MODULE HEADER (EOBJ\$C_EMH), 64 bytes


```

      structure level: 2
      maximum record size: 8192
      module name: "MAIN$PROGRAM"
      module version: "V8.1 - 040392"
      creation date/time: 20-FEB-2006 17:02
      
```
2. LANGUAGE PROCESSOR HEADER (EMH\$C_LNM), 23 bytes


```

      Textual information:
      "Synergy DBL 8.1"
      
```
3. TITLE HEADER (OBJ\$C_HDR_TTL), 18 bytes


```

      Textual information:
      "MAIN$PROGRAM"
      
```
6. END OF MODULE (EOBJ\$C_EEOM), 24 bytes


```

      severity successful (0)
      Highest conditional linkage index: 0
      psect: 0
      value: 0 (%X'00000000')
      transfer address flags:
          (0) EEOM$V_WKTFR 0
      
```

Analyze Object File
 DKA600:[NIGEL]FILE.OBJ;1
 ANALYZ A05-19

20-FEB-2006 17:05:41.51 Page 2

SUMMARY STATISTICS:

Record Type	Count	Total Bytes
OBJ\$C_DBG	0	0
OBJ\$C_TBT	0	0
EOBJ\$C_EMH	3	105
EOBJ\$C_EEOM	1	24
EOBJ\$C_EGSD	0	0
EOBJ\$C_ETIR	0	0
EOBJ\$C_EDBG	0	0
EOBJ\$C_ETBT	0	0
Totals	4	129

The analysis uncovered NO errors.

.IF-.ELSE-.ENDC – Perform conditional processing

```
.IF expression
[...
source_1
...]
[.ELSE
...
[source_2
...]]
.ENDC
```

Arguments

expression

One of the following values:

- ▶ A string, number, or implied-decimal number
- ▶ Any compile-time expression, except one containing ^SIZE or ^ARGNUM. (See [“Compile-Time Expressions and Optimizations” on page 3-41](#) for a complete list of functions that are evaluated at compile time.)
- ▶ A macro variable that expands to one of the above

source_1

(optional) One or more lines of source code.

source_2

(optional) One or more lines of source code.

Discussion

The .IF compilation control directive indicates that the statements that follow are to be compiled if the specified expression is true.

The *source_1* lines of code are compiled only if *expression* is true. If the .ELSE directive is present, the *source_2* lines of code are compiled only if *expression* is false.

You can use one or more of the .ELSE, .IFT, .IFF, and .IFTF directives within an .IF conditional block. These conditional compilation control directives can be nested.

The .ENDC directive must always close an .IF conditional block. If .ENDC is not present, a warning is generated.

Examples

This example shows a simple conditional compilation control operation.

```
.include "GBL:users.def"

function authenticate
record
    user_ok    ,i4

proc

.if ^defined(user.password)
    user_ok = %userauth(user.password)    ;Authenticate the user
.else
    user_ok = %true                        ;Authentication is disabled
.endc
    freturn(user_ok)
endfunction
```

.IFDEF-.ELSE-.ENDC – Perform conditional processing

```
.IFDEF data_element
[...
source_1
...]
[.ELSE
...
[source_2
...]]
.ENDC
```

Arguments

data_element

One of the following:

- ▶ A defined macro
- ▶ A data declaration in the current routine
- ▶ A data declaration in a class (imported or otherwise)
- ▶ A data declaration in a structure, defined at any level and visible at that moment in processing

source_1

(optional) One or more lines of source code.

source_2

(optional) One or more lines of source code.

Discussion

The .IFDEF compilation control directive indicates that the statements that follow are to be compiled if the specified data element has already been defined.

The *source_1* lines of code are compiled only if the specified variable or identifier is defined prior to the .IFDEF directive. If the .ELSE directive is present, the *source_2* lines of code are compiled only if the specified variable or identifier is *not* defined prior to the .IFDEF directive.

You can use one or more of the .ELSE, .IFT, .IFF, and .ITF directives within an .IFDEF conditional block. These conditional compilation control directives can be nested.

The .ENDC directive must always close an .IFDEF conditional block. If .ENDC is not present, a warning is generated.

Examples

The following example shows a simple conditional compilation control operation.

```
.define TTCHN      ,1
.define DEBUG

proc
    open(TTCHN, o, "tt:")
    .
    .
    .

.ifdef DEBUG
    writes(TTCHN, "Entering mod1");This code will be compiled.
.endif
```

The following example shows nesting of the .IFDEF conditional compilation control directive.

```
.define SHOWTABLES
.define SHOWTBLB
.define TTCHN      ,1
proc
    open(TTCHN, o, "tt:")
    .
    .
    .
.ifdef SHOWTABLES
.ifdef SHOWTBLA
    writes(TTCHN, "Table A")          ;This code won't be compiled.
.endif
.ifdef SHOWTBLB
    writes(TTCHN, "Table B")          ;This code will be compiled.
.endif
.endif
```

.IFDEF-.ELSE-.ENDC – Perform conditional processing

```
.IFDEF data_element
[...
source_1
...]
[.ELSE
...
[source_2
...]]
.ENDC
```

Arguments

data_element

One of the following:

- ▶ A defined macro
- ▶ A data declaration in the current routine
- ▶ A data declaration in a class (imported or otherwise)
- ▶ A data declaration in a structure, defined at any level and visible at that moment in processing

source_1

(optional) One or more lines of source code.

source_2

(optional) One or more lines of source code.

Discussion

The .IFDEF compilation control directive indicates that the statements that follow are to be compiled if the specified data element has not previously been defined.

The *source_1* lines of code are compiled only if the specified variable or identifier is *not* defined prior to the .IFDEF directive. If the .ELSE directive is present, the *source_2* lines of code are compiled only if the specified variable or identifier is defined prior to the .IFDEF directive.

You can use one or more of the .ELSE, .IFT, .IFF, and .ITF directives within an .IFDEF conditional block. These conditional compilation control directives can be nested.

The .ENDC directive must always close an .IFDEF conditional block.

Examples

The following example shows a simple conditional compilation control operation.

```
.define TTCHN      ,1

proc
    open(TTCHN, o, "tt:")
.ifndef NO_STATS
    writes(TTCHN, "some stats")    ;This code will be compiled.
.else
    .
    .                               ;Don't write stats
    .
.endifc
```

The next example shows nesting of the .IFDEF conditional compilation control directive.

```
.define TTCHN      ,1
.define SHOWTBLA
    .
    .
    .
proc
    open(TTCHN, o, "tt:")
    .
    .
    .
.ifndef NO_TABLES                               ;This code won't be compiled.
.ifndef SHOWTBLA
    writes(TTCHN, "No information available for Table A")
.else
    writes(TTCHN, "Table A")                    ;This code will be compiled.
.endifc
.endifc
    .
    .
    .
end
```

.IFT, .IFF, .IFTF – Control conditional processing

```
directive
[...
source_1
...]
[.IFT
...
[source_2
...]]
[.IFF
...
[source_3
...]]
[.IFTF
...
[source_4
...]]
.ENDC
```

Arguments

directive

The .IF, .IFDEF, or .IFNDEF compiler directive and the specified data element.

source_1

(optional) One or more lines of source code.

source_2

(optional) One or more lines of source code.

source_3

(optional) One or more lines of source code.

source_4

(optional) One or more lines of source code.

Discussion

The .IFT, .IFF, and .IFTF compilation control directives are used within the .IF, .IFDEF, and .IFNDEF directives to control conditional processing. You can use any number of .IFT, .IFF, .IFTF directives within a conditional block, in any order.

The .IFT, .IFF, and .IFTF directives perform processing based on the following conditions:

.IFT	Instructs the compiler to process the statements that follow if the original condition of the current conditional block is true.
.IFF	Instructs the compiler to process the statements that follow if the original condition of the current conditional block is false.
.IFTF	Instructs the compiler to process the statements that follow regardless of whether the original condition of the current conditional block is true or false.

The *source_1* lines of code are compiled only if the specified conditional directive (.IF, .IFDEF, or .IFNDEF) is true. If the .IFT directive is present, the *source_2* lines of code are also compiled only if the specified conditional directive (.IF, .IFDEF, or .IFNDEF) is true. If the .IFF directive is present, the *source_3* lines of code are compiled only if the specified conditional directive (.IF, .IFDEF, or .IFNDEF) is false. If the .IFTF directive is present, the *source_4* lines are compiled regardless of whether the specified conditional directive is true or false.

You can use one or more of the .ELSE, .IFT, .IFF, and .IFTF directives within an .IF, .IFDEF, or .IFNDEF conditional block of code. The .ELSE and the .IFF directives perform equivalent functions.

See [.IF-.ELSE-.ENDC on page 5-16](#), [.IFDEF-.ELSE-.ENDC on page 5-18](#), and [.IFNDEF-.ELSE-.ENDC on page 5-20](#) for more information about specifying a conditional block of code.

Examples

In the following example, the .ELSE, .IFT, .IFF, and .IFTF directives are all processed according to the result of the original condition of the current conditional block (in this case, .ifdef unix).

```
.define UNIX
.
.
.
record
.ifdef unix                                ;This code will be compiled.
    sys ,a,                                "UNIX"
.else
.ifdef openvms                            ;This code won't be compiled.
    sys ,a,                                "OpenVMS"
.endc
```

Preprocessor and Compiler Directives

.IFT, .IFF, .IFTF

```
.ift                                ;This code will be compiled.
    buf ,a1024
.iff                                ;This code won't be compiled.
    buf ,a512
.iftf                               ;This code will be compiled.
    name,a40
.endc
```

Since UNIX is defined, the following code will be compiled:

```
record
    sys ,a,          "UNIX"
    buf ,a1024
    name,a40
```

.INCLUDE – Include external source code

`. INCLUDE filename`

or

`. INCLUDE "filename" [LIBRARY ["lib_name"]]`

or

`. INCLUDE "structure" REPOSITORY ["rpsfile_log"][, type_spec][, qualifier]`

or

`. INCLUDE "enumueration" REPOSITORY [,enum_access] ENUM`

Arguments

filename

The name of a source file to be included. The default filename extension is **.dbl**. (a)

LIBRARY

(optional) Tells the compiler to search for *filename* in the specified location and include the file in the program.

lib_name

(optional) A logical defined as a search list that specifies the library location. If *lib_name* has an extension, it is stripped. (a)

structure

A structure or alias structure from the S/DE Repository. (a)

REPOSITORY

Tells the compiler to get the structure or enumeration from the S/DE Repository.

rpsfile_log

(optional) A logical defined with the names of the repository files. (a)

type_spec

(optional) One of the following types of data structures to create:

COMMON[= *name*][,x]

EXTERNAL COMMON[= *name*][,x]

GLOBAL COMMON[= *name*][,x]

LITERAL[= *name*][,x]

EXTERNAL LITERAL[= *name*][,x]

GLOBAL LITERAL[= *name*][,x]

LOCAL LITERAL[= *name*][,x]

RECORD[= *name*][,x]

STATIC RECORD[= *name*][,x]

[*access*] [**STATIC**] **RECORD**[= *name*]

LOCAL RECORD[= *name*][,x]

STACK RECORD[= *name*][,x]

NORECORD

STRUCTURE[= *name*][,x]

[*access*] **STRUCTURE**[= *name*]

GROUP[= *name*][,x]

[*access*] [**STATIC**] **GROUP**[= *name*]

[*opt_req*] [*direction*] **GROUP**[= *name*]

where

name

(optional) The name of the data structure to create. (This may be the same name as the repository structure or a different name.) Omitting the name creates an unnamed data structure. (**a**)

x

(optional) The overlay indicator. This is not compatible with the access modifier.

access

(optional) For class members only, one of the following access modifiers:

PUBLIC Access is not restricted.

PROTECTED Access is limited to the containing class or types derived from the containing class.

PRIVATE Access is limited to the containing type. (default)

opt_req

(optional) One of the following modifiers:

OPTIONAL Argument is optional.

REQUIRED Argument is required.

direction

One of the following directional modifiers:

IN	Read only.
INOUT	Read and write.
OUT	Write only.

qualifier

(optional) One or more of the following qualifiers:

NOPRECISION	Indicates that all implied-decimal fields for this structure will be included without their fractional precision. (For example, a d8.2 field will be included as d8 .)
NOFIELDS	Indicates that no fields will be included. Instead, one unnamed alpha field will be included and its size will be the size of the entire record.
PREFIX = “ <i>prefix</i> ”	Specifies that the fields in this structure will have a prefix appended to the beginning of each field’s name.
END	For COMMON , LITERAL , RECORD , and STRUCTURE only, specifies that the corresponding END keyword (ENDCOMMON , ENDLITERAL , etc.) will be generated at the end of the included fields.
DIMENSION = “[<i>n,n,...</i>]”	For GROUP only, passes dimension information where <i>n</i> is either dimension elements or *.

enumeration

The name of the enumeration from the S/DE Repository. (**a**)

enum_access

(optional) For class members only, one of the following access modifiers:

PUBLIC	Access is not restricted.
PROTECTED	Access is limited to the containing class or types derived from the containing class.
PRIVATE	Access is limited to the containing type. (default)

ENUM

Indicates that an enumeration data structure should be created.

Discussion

The `.INCLUDE` compilation control directive instructs the compiler to include source code from another source file, library, or repository. Synergy Language supports up to 20 nested `.INCLUDE` source levels.

If *filename* is not in quotation marks, the name begins with the first non–white space character after `.INCLUDE` and ends with the last non–white space character before the end of the line, ignoring comments. This filename is automatically upcased for processing. For example, the compiler searches for **LOG:FILE.INC** if you enter

```
.include LOG:file.inc
```

If you don't want the compiler to convert filenames to uppercase letters (and you don't want to add quotation marks around the filenames), use the `DBLCASE` logical to specify the case of your filename specifications. In our example, the compiler searches for **LOG:file.inc** if you set

```
dblcase=u:l
```

See [DBLCASE](#) in the “Environment Variables” chapter of *Environment Variables and System Options* for more information.



Any white space characters between text elements is ignored. For example,

```
.include name  
is treated the same as  
.include na me
```

If *filename* is in quotation marks, you must either specify a valid `DICTIONARY`, `REPOSITORY`, or `LIBRARY` specification or not have any text other than white space and comments after the `.INCLUDE` directive.

If you `.INCLUDE` a source file (as opposed to a repository structure or enumeration), the source lines from the included file are compiled until the end-of-file character is reached. Once the end-of-file character is reached, compilation continues with the line that follows the `.INCLUDE` statement.

If you don't specify the `LIBRARY` qualifier, *filename* begins with the first non–white space character after `.INCLUDE` and ends with the first non–white space character before the end of the line, excluding comments.

If you specify the `LIBRARY` qualifier, *filename* must be an alpha expression. If *lib_name* is omitted, the compiler searches for *filename* in the directories specified by the `DBLLIBRARY` logical, as if you typed the following:

```
.include "DBLLIBRARY:filename"
```

You can't `.INCLUDE` a prototype. Use the `IMPORT` statement instead.

Including from a repository

When you include a structure or an enumeration from a repository, the compiler produces a temporary file with the specified repository contents. This temporary file is included into the source file when the source file is compiled, and then it is deleted. (If the compiler generates a listing, the temporary include file is generated to the listing file.)

You can include enumerations globally or within a class or namespace. Enumeration `.INCLUDEs` always terminate automatically.

For structures, you can include fields in an argument group, class record, routine record, common record, or global data section. You can specify any of the appropriate modifiers as qualifiers based on the location of the `.INCLUDE`. For example, if the `.INCLUDE` is being performed on an argument group, any of the argument modifiers can be specified as qualifiers.

The `rpsfile_log` argument is the name of a logical that has been set to the repository main and text filenames from which to include the specified repository structure. If you don't specify `rpsfile_log`, the default logical is `DBLDICTIONARY`. If `DBLDICTIONARY` is not defined, the compiler looks for `RPSMFIL` and `RPSTFIL` to determine the locations of the repository files. We recommend that you use `RPSMFIL` and `RPSTFIL` instead of `DBLDICTIONARY`, because those environment variables are also used by S/DE Repository. Information about defining `DBLDICTIONARY`, `RPSMFIL`, and `RPSTFIL` can be found in the “[Environment Variables](#)” chapter of *Environment Variables and System Options*.



To avoid an error, we recommend that you uppercase the logical that defines a repository location in your `.INCLUDE` statements.

When you include a structure from a repository, only fields that do not have the “Excluded by Language” flag set are included by name. Fields that have the “Excluded by Language” flag set are included as unnamed fields of the appropriate size. (Overlay fields flagged as “Excluded by Language” are not included.) See “[Defining Fields](#)” in the “Repository” chapter in the *Repository User's Guide* for more information.

If your repository structure includes a group that has a “Member prefix” defined, this prefix is only appended to the beginning of the group's member fields if the “Use by compiler” flag is set. See “[Defining Fields](#)” in the “Repository” chapter in the *Repository User's Guide* for more information.

If you specify the `PREFIX` qualifier, the prefix added by the compiler is in addition to any prefixes specified for group members (fields) if the “Use by compiler” flag is set in Repository.

The `PREFIX` and `NOPRECISION` qualifiers can be specified at the same time, in any order. `NOFIELDS`, on the other hand, cannot be specified in conjunction with any other qualifier except `END`.

If you don't specify `type_spec`, `RECORD` is the default data structure type. If `NORECORD` is specified, only fields are created; there is no `RECORD` statement.

Group `.INCLUDEs` always terminate automatically. Record and structure `.INCLUDEs` in a class terminate automatically; you do not need to specify the corresponding `END` qualifier. (See [page 4-180](#) for an exception to this rule.) For global structure `.INCLUDEs`, the `END` qualifier is recommended, but not required; if it is not present, the compiler generates an “`ENDSTRUCTURE` on global structure expected” warning (`NOSPECL`).

If your repository structure includes overlay fields, the compiler specifies them differently depending on whether the record is named or unnamed in your code. If the record is *named*, the compiler specifies the overlay field with dot notation using the record name, group names (if there are any), and field name. For example, if you have this structure in your repository:

```
employee
  id          ,a4
  startdate   ,d8
  day         ,d2 @startdate
  month       ,d2 @startdate+2
  year        ,d4 @startdate+4
  salary      ,d10
```

and you `.INCLUDE` the structure like this in your code:

```
.include "employee" repository, record="emp_rec"
```

you will see this in the code listing:

```
RECORD emp_rec
  id          ,a4          ; (1,4)
  startdate   ,d8          ; (5,12)
  day         ,d2 @emp_rec.startdate ; (5,6)
  month       ,d2 @emp_rec.startdate+2 ; (7,8)
  year        ,d4 @emp_rec.startdate+4 ; (9,12)
  salary      ,d10         ; (13,22)
```

If the record is *unnamed* (i.e., the `NORECORD` qualifier is used), the compiler specifies the overlay fields with an offset relative to the overlay field (not the overlaid field). Hence, the offset will be a negative number. For example, if you `.INCLUDE` the above structure like this in your code:

```
.include "employee" repository, norecord
```

you will see this in the code listing:

```
  id          ,a4          ;
  startdate   ,d8          ;
  day         ,d2 @-8      ;
  month       ,d2 @-6      ;
  year        ,d4 @-4      ;
  salary      ,d10         ;
```




Because the compiler uses a named reference when the record is named, an “Ambiguous symbol” error could occur if there is another symbol with the same name in your code. You can use the NORECORD qualifier to eliminate this possibility for error. If you still want to be able to refer to the record by name, you can do this:

```
record emp_rec
.include "employee" repository, norecord
```



We recommend that you declare any data that will be used in more than one routine in an include file or in the S/DE Repository, thereby ensuring that all routines have exactly the same record definitions. Also, if you need to change the record layout of a data file, the .INCLUDE directive enables you to make the changes in one place (the include file or repository), instead of having to modify each routine that contains the record layout.

Examples

Source file examples

- A. In the following example, the compiler searches for **file.dbl**.

```
.include "file"
```

- B. In the following example, the compiler searches for **file.inc** in the directories specified by the LIB logical.

```
.include "LIB:file.inc"
```

- C. Assuming that DBLCASE is not defined, the following example searches for **FILE.DBL** in the directories specified by the LIB logical.

```
.include LIB:file
```

- D. In the following example, the compiler searches for **file.dbl** in the directories specified by the DBLLIBRARY logical.

```
.include "file.dbl" LIBRARY
```

- E. In the example below, the compiler searches for **file.dbl** in the directories specified by the MYLIBS logical.

```
.include "file.dbl" LIBRARY "MYLIBS"
```

- F. In the following example, the compiler generates an “Alpha expression expected” error (ALPHAXP) because **file** is not an alpha expression and the LIBRARY qualifier is specified.

```
.include file LIBRARY
```

Repository examples

Below are examples that specify the REPOSITORY option. In these examples, the ORDER structure has the following fields:

```
ord_id      ,d6
cust_id     ,d6
```

- A.** In this example, because we don't specify *rpsfile_log*, the compiler uses the DBLDICTIONARY logical to get the repository filenames. If DBLDICTIONARY is not defined, the compiler uses the RPSMFIL and RPSTFIL logicals.

```
.include "order" REPOSITORY
```

The compiler reads the ORDER element from the repository and create a temporary include file with the following contents.

```
record
  ord_id      ,d6
  cust_id     ,d6
```

Because we didn't specify a data structure type, the default, RECORD, is used.

- B.** In this example, the compiler again reads from the repository files specified by the default logical, DBLDICTIONARY and generates a record named "prev_order".

```
.include "order" REPOSITORY, RECORD = "prev_order"
```

The compiler produces a temporary file with the following contents:

```
record prev_order
  ord_id      ,d6
  cust_id     ,d6
```

- C.** In this example, the compiler uses the repository files specified by the DDNAME logical to generate an unnamed global common.

```
.include "order" REPOSITORY "DDNAME", GLOBAL COMMON
```

The compiler produces a temporary include file with the following contents:

```
global common
  ord_id      ,d6
  cust_id     ,d6
```

- D.** In this example, the compiler reads from the repository files specified by the default logical, DBLDICTIONARY to generate a literal named “order”.

```
.include "order" REPOSITORY, LITERAL = "order" ,x
```

The compiler produces a temporary file with the following contents:

```
literal order ,x
    ord_id      ,d6
    cust_id     ,d6
```

- E.** In this example, the compiler again reads from the repository files specified by the default logical, DBLDICTIONARY.

```
.include "order" REPOSITORY, NORECORD
```

No data structure type is generated. The compiler generates a temporary file with the following contents:

```
ord_id      ,d6
cust_id     ,d6
```

- F.** In this example, the compiler again reads from the repository files specified by the default logical, DBLDICTIONARY and generates a group named “order_info”.

```
.include "order" REPOSITORY, GROUP = "order_info"
```

The compiler produces a temporary file with the following contents:

```
group order_info
    ord_id      ,d6
    cust_id     ,d6
endgroup
```

- G.** In this example, the compiler again reads from the repository files specified by the default logical, DBLDICTIONARY, creating a structure with each field prefixed by “new_”.

```
.include "order" REPOSITORY, STRUCTURE="order", PREFIX="new_"
```

The compiler produces a temporary file with the following contents:

```
structure order
    new_ord_id  ,d6
    new_cust_id ,d6
```

- H. The example below defines a dimensioned group.

```
.INCLUDE "rps_struct" REPOSITORY, GROUP = "mygrp_array", DIMENSION = "[100]"
```

The compiler produces a temporary file with the following contents:

```
group mygrp_array, [100]a
    ord_id      ,d6
    cust_id     ,d6
endgroup
```

- I. The example below includes the **colors** enumeration from the repository.

```
.include "colors" REPOSITORY, ENUM
```

The compiler produces a temporary file, which might look like this:

```
enum colors
    red      ,3
    blue     ,5
    green    ,7
endenum
```

.LIST – Control the listing of source code

.LIST

Discussion

The .LIST compilation control directive, together with the .NOLIST and .START directives, control the listing of source code.

The .LIST directive increments a listing counter, which has an initial value of zero. Listing is enabled only if the counter is greater than or equal to zero and listing has been enabled at the command line. A .LIST line that turns listing on won't appear in the listing.

If listing is not enabled at the command line, the compiler ignores all .LIST directives. See the [“Building and Running Synergy Language Programs”](#) chapter of *Synergy Language Tools* for more information on how to enable listing at the command line.

Examples

The following example assumes that listing has been enabled at the command line and that the listing counter is currently zero.

```
.nolist                ;The listing counter will be decremented.
.include "file"        ;No statements from this include file will
                        ; be listed.
.list                 ;The listing counter will be incremented,
                        ; thus listing will be enabled again and
                        ; the rest of the source lines will be
                        ; written to the listing file.

proc
.
.
.
end
```

In this example, if a second .NOLIST exists in the .INCLUDEd file, the listing counter is decremented to -2. Since the .LIST following the .INCLUDE statement only increments the counter to -1, listing remains disabled throughout the file.

.NODEBUG – Ignore breakpoints in debug mode

.NODEBUG

Discussion

The .NODEBUG compiler directive prevents the debugger from stopping at breakpoints, watchpoints, or anything else that causes the program to pause and the debugger to be entered in the routine running the program in debug mode. In other words, if the program is being debugged (**dbr -d** *program*) and any attempt to break is made in the routine where this compiler directive exists, the break is ignored and program execution continues.

.NOLIST – Control the listing of source code

.NOLIST

Discussion

The .NOLIST compilation control directive, together with the .LIST and .START directives, control the listing of source code.

The .NOLIST directive decrements a listing counter, which has an initial value of zero. Listing is enabled only if the counter is greater than or equal to zero and listing is enabled at the command line.

If an error is encountered while listing is disabled, the compiler lists the statement containing the error and the error message.

If listing is not enabled at the command line, the compiler ignores all .NOLIST directives. See the [“Building and Running Synergy Language Programs”](#) chapter of *Synergy Language Tools* for more information about enabling listing at the command line.

Examples

See [.LIST on page 5-35](#) for an example that uses the .NOLIST directive.

.NOPROTO-.PROTO – Turn prototype generation off and on

```
.NOPROTO
.
.
.
[ .PROTO]
```

Discussion

The .NOPROTO compiler directive turns off prototype generation for a block of source code in a program, and the .PROTO compiler directive turns it back on.

If you specify .NOPROTO in your source file, the Synergy prototype utility (**dblproto**) ignores the section of code from that point on, to either the matching .PROTO directive or the end of the source file, whichever comes first. All code after the matching .PROTO is processed as normal.



When the Synergy prototype utility is running, the parser does not see the code in a .NOPROTO-.PROTO block, as if you had specified the compiler directives .IF 0 and .ENDC. Definitions or declarations within a .NOPROTO block are also not included. As a result, code that compiles correctly with the Synergy compiler may not compile when the Synergy prototype utility is run against it. If this happens, make sure the .NOPROTO section does not contain code that is needed in the .PROTO section.

We do not recommend using .NOPROTO-.PROTO around a class specification.

.PROTO cannot be specified without a matching .NOPROTO.

Examples

The example below creates a prototype for sub2 but not for sub1.

```
.nopproto

subroutine sub1
proc
    xreturn
end

.proto

subroutine sub2
proc
    xreturn
end
```


.PAGE – Advance program listing to top of new page

.PAGE

Discussion

The .PAGE compilation control directive terminates the current listing page and begins a new one. This directive is ignored if there is no listing file.

The compiler doesn't list the line containing the .PAGE directive. The first line on the next page is the line immediately following the .PAGE directive.

Examples

```
proc
    .
    .
    .
end                ;This will be the last line listed
                   ; on the current page.
.page             ;This line won't be listed.
subroutine sub     ;This will be the first line listed
                   ; on the next page.

proc
    .
    .
    .
endsubroutine
```

.REGION-.ENDREGION – Specify a collapsible block of code

```
.region name
.
.
.
.endregion
```

Arguments

name

The name of the region.

Discussion

The .REGION-.ENDREGION directives enable you to hide code from view by delineating a block of code that Visual Studio .NET or Workbench will collapse and mark with a small box containing a plus sign. The classes, fields, groups, etc., defined within that region are not visible until you click on the box to expand it. When the region is expanded, a minus sign is displayed in the box.

Examples

In the example below, the region named My Region will be collapsed so that the class **fred** is hidden until you click on the plus sign to expand it. The region named Record Fields will also be collapsed, with the fields and groups defined within records in that region hidden until you click the plus sign.

```
namespace ns
.region "My Region"
    public class fred
        ; Class fields/methods
        ...
    endclass
.endregion
endnamespace

main
record myrecord
.region "Record Fields"
    ; Fields/groups within records
    ...
.endregion

proc
nop
end
```

.START – Control the listing of source code

`.START[ [NO]COND][ [NO]LIST][ [NO]OFFSETS][ [NO]PAGE][ [NO]SUMMARY]`

Arguments

`[NO]COND`

(optional) Turns the listing of false conditional blocks of source code on or off. By default, false conditional blocks of code are listed (COND).

`[NO]LIST`

(optional) Turns the listing of source code on or off. By default, the listing of source code is turned on (LIST).

`[NO]OFFSETS`

(optional) Turns the listing of the symbol table offsets on or off. Symbol table offsets list all the referenced symbols for each routine. By default, the listing of these offsets is turned off (NOOFFSETS).

`[NO]PAGE`

(optional) Turns the starting of a new page in a listing on or off. By default, the starting of a new page is turned off (NOPAGE).

`[NO]SUMMARY`

(optional) Turns the listing of a memory usage summary table on or off. By default, the listing of this table is turned off (NOSUMMARY).

Discussion

The `.START` compilation control directive, together with the `.LIST` and `.NOLIST` directives, control the listing of source code.

Specifying `.START LIST` increments the listing counter, which has an initial value of 0. Specifying `.START NOLIST` decrements the listing counter. Listing is enabled only if the counter is greater than or equal to 0 and listing has been enabled at the command line.

If listing is not enabled at the command line, the compiler ignores all `.START` directives. You can use listing options at the command line to override one or more of the `.START` listing options. See the [“Building and Running Synergy Language Programs”](#) chapter of *Synergy Language Tools* for more information about enabling listing at the command line.

Specifying `.START` without any options defaults to a page break, unless the `.START` is the first line on the page, in which case a new page is not generated. If any options are specified, the default is no page break. A `.START` line that turns listing on won't appear in the listing.

See [Appendix A](#) in *Synergy Language Tools* for more information about the symbol table offsets and memory usage summary tables.

Examples

The following example assumes that listing has been enabled at the command line.

```
.start offsets summary          ;Print the symbol table offsets and
                                ; memory usage tables at the end of each
                                ; routine. Don't generate a new page.

main main_proc
proc
    .
    .
    .
endmain

.start                          ;Generate a new page.
subroutine sub
proc
    .
    .
    .
endsubroutine

.start nocond nooffsets        ;Do not write code within false
                                ; condition blocks to the listing file,
                                ; generate symbol table offsets, or
                                ; generate new page. Generate memory
                                ; usage summary table.

function func
proc
    .
    .
    .
endfunction
```

.TITLE – Specify the listing page header

`.TITLE[ header]`

Arguments

header

(optional) The alpha expression or text to be used as the listing page header (up to 65 characters).

Discussion

The `.TITLE` compilation control directive specifies the listing page header.

The `.TITLE` directive is the first source line listed on a new page. If the listing is already at the beginning of a page when `.TITLE` is encountered, the compiler won't generate a new page.

The compiler uses the specified header string or text in the listing page header for all pages until it encounters a new `.TITLE` directive. The header string is moved to the listing page header according to the rules for moving alpha data to an alpha destination. (See [“Assignment Statements” on page 3-26](#) for more information about these rules.) If *header* is not specified, the listing page header is filled with spaces.

Examples

In the following example, the compiler generates a new page in the listing, the page header is “This is the title,” and this source line is the first line of the new page.

```
.title "This is the title"
```

In the following example, the compiler generates a new page, the page header is blank, and this source line is the first line on a new page.

```
.title
```

In the following example, the compiler generates a new page, the page header is “Program name: A_PROG,” and this source line is the first line on the new page.

```
.title "Program name: A_PROG"
```

.UNDEFINE – Remove definition of replacement identifier

`.UNDEFINE identifier`

Arguments

identifier

The replacement identifier to be removed.

Discussion

The .UNDEFINE compilation control directive removes the definition of a replacement identifier.

If *identifier* is not defined in a prior .DEFINE directive line, the compiler generates an “Identifier not defined” warning (NOTDEF). You can suppress this warning by specifying the .DEFINE warning compiler option. See “[Compiling a Synergy Language Routine](#)” in the “Building and Running Synergy Language Programs” chapter of *Synergy Language Tools* for more information about this compiler option.

Examples

```
.define DEBUG                ;This directive defines DEBUG in
.                            ; the symbol table.
.
.
.ifdef DEBUG                 ;The following code will be compiled
.                            ; because DEBUG is defined.
.
.
.endif
.undefine DEBUG              ;This directive removes DEBUG
.                            ; from the symbol table.
.
.
.ifdef DEBUG                 ;The following code won't be compiled
.                            ; because DEBUG is no longer defined.
.
.
.endif
.undefine DEBUG              ;This directive generates a compiler warning
.                            ; because DEBUG is not defined.
```

6

Error Handling

This chapter discusses the two error handling capabilities that Synergy Language provides: ONERROR handling and structured exception handling. You cannot use both types in the same routine.

Error Trapping 6-2

Describes the error trapping capabilities provided by the ONERROR and OFFERROR statements and explains how I/O error lists are used with the I/O statements to trap and handle errors.

Structured Exception Handling 6-4

Explains how to handle exceptions using TRY-CATCH blocks.

Exception Classes 6-6

Provides the Synergy exception class hierarchy and the errors that map to each exception class.

Error Trapping



Synergy Language is restricted to the characteristics of the underlying index subsystem (Synergy DBMS, RMS, and so forth). If an error occurs on an ISAM I/O statement such as FIND, READ, or READS (other than “Key not same” on an ISAM file), do *not* assume that the context or contents of the return buffers are the same on all subsystems.

For example, if the first operation on a file is a FIND returning EOF, Synergy DBMS positions the next record as EOF, while RMS positions the next record as BOF—and a READS yields different results on each. Therefore, if you want to perform a READS after an erroneous operation, you must perform a FIND without error before performing the READS operation.

Also, following a READ or READS error condition, the contents of the passed data record is undefined and should not be relied upon for subsequent operations. For example, on an ISAM file, if **key** is a field within the passed data record and an “End of file” error occurs, the **key** value must be reloaded before **key** can be used by the next READ.

Additionally, an RFA retrieved via the GETRFA qualifier, GETRFA routine, or GETFA(“RFA”) routine should be disregarded immediately following an I/O error.

When Synergy Language begins execution, all runtime errors are considered fatal. However, you can designate a subset of all possible runtime errors as trappable. This subset contains error conditions the program can resolve so that processing need not terminate.

A program can recognize and handle any subset of defined trappable errors with I/O error lists in the I/O statements and with the ONERROR statement. If an I/O error list exists, it supersedes any traps established by an ONERROR statement. For example, if both an I/O error list and ONERROR establish a label for the “End of file” error (\$ERR_EOF), the label in the I/O error list is used.

When a trappable runtime error occurs, the runtime checks to see if the program has defined its own trap for the error. If it hasn’t, the program ends with the appropriate error message. If the program has defined a trap for the error, the operation causing the error is not completed and control is immediately transferred to the label associated with the error trap.

After an error is trapped, the program can determine which error caused the trap either by using the %ERNUM intrinsic function or calling the ERROR subroutine. The ERROR subroutine or the %ERLIN intrinsic function returns the line number on which the error occurred. You can obtain the text of an error using the ERTXT subroutine.

The precedence of error traps is as follows:

- Specific I/O error trap
- General I/O error trap ([ERR=*label*])
- READS or GETS end-of-file label
- Qualified ONERROR statement
- Unqualified ONERROR statement

I/O error lists

An I/O error list can be present in each Synergy Language I/O statement. This list specifies labels to which control can be transferred if errors occur. Each particular error can be associated with a different label. This provides a simple but effective way to discriminate between possible errors an I/O statement might encounter.

An I/O error list is in effect only during the execution of the statement associated with the list. While the statement is being executed, any error specified in the list has precedence over the ONERROR list currently in effect.

An I/O error list has the following format:

element=label, ...

element

One or more of the following items:

<i>err_literal</i>	Any trappable error literal.
DUP	Handle the “Duplicate key specified” error (\$ERR_NODUPS).
EOF	Handle past the end of the input file or overflow of output file errors (\$ERR_EOF).
ERR	Handle any error not anticipated by other elements in the list.
KEY	Handle the ISAM “Key not same” error (\$ERR_KEYNOT).
LOCK	Handle record locking conflict errors (\$ERR_LOCKED).
LONG	Handle the non-ISAM “Input exceeds size of input data area” error (\$ERR_TOOBIG).

label

A label to which control is transferred if a specified error occurs.



We recommend that you use the error literal form in an I/O error list instead of specifying DUP, EOF, ERR, KEY, LOCK, or LONG.

The following example transfers control to the **errlbl** label if an “Invalid record size” error (\$ERR_IRCSIZ) occurs on the READS statement:

```
reads(TTCHN, rec) [$ERR_IRCSIZ=errlbl]
```

Error literals are listed in the “[Error Messages](#)” chapter of *Synergy Language Tools*.

Structured Exception Handling

Structured exception handling means that an exception is thrown every time an error is encountered. This causes the stack to unwind until the appropriate exception handler for that type of exception is found. `System.Exception` is the parent class for all exceptions. A class called `Synergex.SynergyDE.SynException` extends this parent class with Synergy error information.

The main syntax elements of structured exception handling are the TRY-CATCH-FINALLY statement and the THROW statement. The TRY-CATCH-FINALLY statement is designed to handle any thrown exceptions within the context of a TRY block. An exception can be thrown one of two ways:

- ▶ Implicitly by the Synergy Runtime when an error is encountered
- ▶ Explicitly by the application with the THROW statement

To use structured exception handling, your Synergy Language code must contain TRY-CATCH block syntax that includes code to throw a `System.Exception`. (See [TRY-CATCH-FINALLY-ENDTRY on page 4-186](#) for more information.) Each CATCH block should specify the type of exception to be caught and a local variable to use as the exception instance variable within the block.

A Synergy exception enables you to retrieve the following information:

- ▶ Error message
- ▶ Call stack at the time the current exception was thrown

After you've added TRY-CATCH blocks to your code and compiled, you can use the compiled application to test your exception handling code. Any exception that is thrown inside the scope of the TRY block is tested against the declared types for each CATCH block. If the declared exception to be caught is in the hierarchy tree of the thrown exception, that CATCH block will be executed. Otherwise, the call stack is unwound until either the exception is caught or the main routine is reached and the exception is not caught.

Customizing error information

To customize errors, you can extend an exception class to include additional information:

1. Create a new exception class that extends `System.Exception` or `System.ApplicationException`.



For compatibility with .NET, we recommend that you extend `System.ApplicationException`.

2. Add methods or fields and/or override inherited methods or fields in the new exception class as needed.

3. Add a **THROW** statement to your code to throw the new exception. (See [THROW on page 4-185.](#))
4. Add a **FINALLY** block to your **TRY-CATCH** code.

The **FINALLY** block runs on successful completion of the **TRY** block or when the **CATCH** block exits on an exception. Exit conditions include end of block, return statements, and statements that jump to a label outside the **TRY-CATCH** block. The destructors for any objects that go out of scope as the stack unwinds will be called.

If a thrown exception is not caught by any error handler, the application terminates, providing the error information and traceback information that is currently being provided by Synergy. **FINALLY** blocks and destructors are *not* executed on an uncaught exception.

Exception Classes

So that you don't have to catch hundreds of individual exceptions, the exception classes related to Synergy/DE runtime errors are organized in a hierarchy that groups common exceptions and like items together. You can use the classes in the hierarchy like this:

```
proc
try
    open(1, i, 'LOC:myfile.tst')
catch (e, @Synergex.SynergyDE.NoFileFoundException)
    writes(15, "File not found")
catch (e, @Synergex.Synergyde.FileIOException)
    writes(15, "Other file open error - Error " + %string(e.errno))
catch (e)
    writes(15, "Other system exception")
endtry
```

When catching the exception, you can determine the underlying Synergy/DE error number with the **Errno** field of the exception variable. For example,

```
catch (e, Synergex.SynergyDE.BadDigitException)
    writes(15, %string(e.Errno))
```

Synergy exception class hierarchy

The following exception class hierarchy is supported. Each indented class is derived from its parent (the lesser indented) class. See [“Synergy exception class error mapping” on page 6-8](#) for a list of errors that map to each exception class.

- ▶ System.Exception
 - ▶ System.SystemException
 - ▶ System.ArgumentException
 - ▶ System.ArgumentNullException
 - ▶ System.ArgumentOutOfRangeException
 - ▶ System.ArithmeticException
 - ▶ System.DivideByZeroException
 - ▶ System.OverflowException
 - ▶ System.OutOfMemoryException
 - ▶ System.InvalidCastException
 - ▶ System.IndexOutOfRangeException
 - ▶ System.MemberAccessException
 - ▶ System.FieldAccessException

- ▶ System.MethodAccessException
- ▶ System.MissingMemberException
 - ▶ System.MissingFieldException
 - ▶ System.MissingMethodException
- ▶ System.InvalidOperationException
- ▶ System.NullReferenceException
- ▶ System.ApplicationException
- ▶ Synergex.SynergyDE.SynException
 - ▶ Synergex.SynergyDE.AXException
 - ▶ Synergex.SynergyDE.BadDigitException
 - ▶ Synergex.SynergyDE.InterruptException
 - ▶ Synergex.SynergyDE.ObjectException
 - ▶ Synergex.SynergyDE.RoutineNotFound
 - ▶ Synergex.SynergyDE.SQLException
 - ▶ Synergex.SynergyDE.SynArgumentException
 - ▶ Synergex.SynergyDE.SynDataException
 - ▶ Synergex.SynergyDE.SynIOException
 - ▶ Synergex.SynergyDE.DeviceNotReadyException
 - ▶ Synergex.SynergyDE.DeviceInUseException
 - ▶ Synergex.SynergyDE.FileIOException
 - ▶ Synergex.SynergyDE.DeletedRecordException
 - ▶ Synergex.SynergyDE.DuplicateException
 - ▶ Synergex.SynergyDE.EndOfFileException
 - ▶ Synergex.SynergyDE.FileFullException
 - ▶ Synergex.SynergyDE.FileInUseException
 - ▶ Synergex.SynergyDE.FileNameException
 - ▶ Synergex.SynergyDE.FileReplaceException
 - ▶ Synergex.SynergyDE.IllegalRecordNumberException
 - ▶ Synergex.SynergyDE.KeyNotSameException
 - ▶ Synergex.SynergyDE.NoCurrentRecordException
 - ▶ Synergex.SynergyDE.NoFileFoundException
 - ▶ Synergex.SynergyDE.NoSpaceException
 - ▶ Synergex.SynergyDE.ProtectionViolationException
 - ▶ Synergex.SynergyDE.RecordLockedException

- ▶ Synergex.SynergyDE.RecordNotFoundException
- ▶ Synergex.SynergyDE.RecordNotSameException
- ▶ Synergex.SynergyDE.RecordTooBigException
- ▶ Synergex.SynergyDE.OperationTimeoutException
- ▶ Synergex.SynergyDE.SynServerException
 - ▶ Synergex.SynergyDE.NetworkException
 - ▶ Synergex.SynergyDE.NoConnectionException
 - ▶ Synergex.SynergyDE.ServerNotRunningException
 - ▶ Synergex.SynergyDE.UnknownHostException
 - ▶ Synergex.SynergyDE.UsernameException
- ▶ Synergex.SynergyDE.WNDEException

Synergy exception class error mapping

Synergy/DE maps the following errors to the exception classes listed below.

- ▶ **Synergex.SynergyDE.AXException**

\$ERR_AXERR	Error while processing an ActiveX control
\$ERR_AXNOLOAD	Could not load ActiveX control
\$ERR_AXNOSUB	Could not find subroutine or function
\$ERR_AXNOFIND	ActiveX parameter not found
\$ERR_AXUNSUP	Unsupported feature
- ▶ **Synergex.SynergyDE.BadDigitException**

\$ERR_DIGIT	Bad digit encountered
- ▶ **Synergex.SynergyDE.DeletedRecordException**

\$ERR_DELREC	Deleted record
- ▶ **Synergex.SynergyDE.DeviceInUseException**

\$ERR_DEVUSE	Device in use
- ▶ **Synergex.SynergyDE.DeviceNotReadyException**

\$ERR_DEVNOTRDY	Device not ready
- ▶ **Synergex.SynergyDE.DuplicateException**

\$ERR_NODUPS	Duplicate key specified
- ▶ **Synergex.SynergyDE.EndOfFileException**

\$ERR_EOF	End of file

- ▶ **Synergex.SynergyDE.FileFullException**
 - \$ERR_FILFUL Output file is full
- ▶ **Synergex.SynergyDE.FileInUseException**
 - \$ERR_FINUSE File in use by another user
- ▶ **Synergex.SynergyDE.FileIOException**
 - \$ERR_DEADLOCK Operation would cause deadlock
 - \$ERR_FILOPT Invalid operation for file type
 - \$ERR_FILORG Invalid file organization
 - \$ERR_INVALRFA Invalid record's file address
 - \$ERR_IRCSIZ Invalid record size
 - \$ERR_ISINFO ISINFO error
 - \$ERR_INVKVAL Invalid key value
 - \$ERR_INVPKEY Invalid partial key
 - \$ERR_NOTISM Not an ISAM file
 - \$ERR_OPNERR OPEN error
 - \$ERR_RECLNG Invalid record length
 - \$ERR_RELREC Invalid relative record
 - \$ERR_RMSERROR Unexpected RMS error
- ▶ **Synergex.SynergyDE.FileNameException**
 - \$ERR_FILSPC Bad filename
- ▶ **Synergex.SynergyDE.FileReplaceException**
 - \$ERR_REPLAC Cannot supersede existing file
- ▶ **Synergex.SynergyDE.IllegalRecordNumberException**
 - \$ERR_RECNUM Illegal record number specified
- ▶ **Synergex.SynergyDE.InterruptException**
 - \$ERR_INTRPT Interrupt character detected
- ▶ **Synergex.SynergyDE.KeyNotSameException**
 - \$ERR_KEYNOT Key not same

- ▶ **Synergex.SynergyDE.NetworkException**
 - \$ERR_NETCRYPT File requires network encryption
 - \$ERR_NETPROB Network problem reaching server
- ▶ **Synergex.SynergyDE.NoCurrentRecordException**
 - \$ERR_NOCURR No current record
- ▶ **Synergex.SynergyDE.NoConnectionException**
 - \$ERR_XFNOCONN No connection to remote host
- ▶ **Synergex.SynergyDE.NoFileFoundException**
 - \$ERR_FNF File not found
- ▶ **Synergex.SynergyDE.NoSpaceException**
 - \$ERR_NOSPAC No space exists for file on device
- ▶ **Synergex.SynergyDE.ObjectException**
 - \$ERR_CLSMTCH Class mismatch between routines
 - \$ERR_DUPEVENT Event code already specified
 - \$ERR_INCPTCLS Incompatible classes
 - \$ERR_INVCLASS Class is invalid for operation
 - \$ERR_INVCLSHND Invalid class handle
 - \$ERR_INVNETHND Invalid network handle
 - \$ERR_INVNAMHND Invalid name space handle
 - \$ERR_INVOBJHND Invalid object handle
 - \$ERR_NOOBJ No object for handle
 - \$ERR_NOTCLSHND Handle is not a class handle
 - \$ERR_NOTNAMHND Handle is not a namespace handle
 - \$ERR_NOTNETHND Handle is not a network handle
 - \$ERR_NOTOBJHND Handle is not an object handle
 - \$ERR_NOTOBJID Not an object identifier
 - \$ERR_NOTOHND Both source and destination must be object handles
 - \$ERR_OHNDREQ Object handle required
- ▶ **Synergex.SynergyDE.OperationTimeoutException**
 - \$ERR_TIMEOUT Terminal input operation timeout

- ▶ **Synergex.SynergyDE.ProtectionViolationException**
 - \$ERR_PROTEC Protection violation
- ▶ **Synergex.SynergyDE.RecordNotFoundException**
 - \$ERR_RNF Record not found
- ▶ **Synergex.SynergyDE.RecordLockedException**
 - \$ERR_LOCKED Record is locked
- ▶ **Synergex.SynergyDE.RecordNotSameException**
 - \$ERR_RECNOT Record not same
- ▶ **Synergex.SynergyDE.RecordTooBigException**
 - \$ERR_TOOBIG Input data size exceeds destination size
- ▶ **Synergex.SynergyDE.RoutineNotFound**
 - \$ERR_FNOTFOUND Function not found
 - \$ERR_NOMETHOD Method's routine not found
 - \$ERR_RTNNF Cannot access external routine
- ▶ **Synergex.SynergyDE.ServerNotRunningException**
 - \$ERR_NOSEVER Synergy server is not running
- ▶ **Synergex.SynergyDE.SQLErrorException**
 - \$ERR_BADDATATYP SQL: Invalid data type for this operation
 - \$ERR_CURSERR SQL: Error on cursor
 - \$ERR_NOMORECURS SQL: No more available open cursors
 - \$ERR_NOSQL SQL: Connection installation error or DBLOPT 48 not set
 - \$ERR_SQLDYN SQL: ^M variable still bound/defined on dynamic memory deletion
 - \$ERR_SQLERR SQL Connection error
 - \$ERR_SQLSTACK SQL: Stack variable still bound/defined on routine exit
- ▶ **Synergex.SynergyDE.SynArgumentException**
 - \$ERR_ALPHARG Alpha argument required
 - \$ERR_AORDXP Alpha or decimal variable expected
 - \$ERR_AORIARG Integer or alpha argument required
 - \$ERR_ARGDIG Numeric digit(s) expected in argument

\$ERR_ARGMIS	Argument missing
\$ERR_ARGORD	Arguments out of order for PAK or UNPAK
\$ERR_ARGREC	PAK/UNPAK fields not in record
\$ERR_ARGSIZ	Argument specified with wrong size
\$ERR_BDIGXP	Binary digits expected in argument
\$ERR_HDIGXP	Hexadecimal digits expected in argument
\$ERR_IDXP	Implied data type required
\$ERR_INTARG	Integer argument required
\$ERR_INVARG	Invalid argument
\$ERR_NUMXP	Numeric argument expected
\$ERR_OPTINV	Invalid option
\$ERR_WROARG	Incorrect number of subroutine arguments
\$ERR_WRTLIT	Writing to a literal or missing argument

► **Synergex.SynergyDE.SynDataException**

\$ERR_SUBSCR	Invalid subscript specified
\$ERR_ADDRSIZ	Invalid address size
\$ERR_ARGDIGPT	Numeric digit(s) and at most one decimal point expected
\$ERR_DATACRYPT	Error encrypting/decrypting data field: %s
\$ERR_INVDSCR	Invalid descriptor
\$ERR_INVDIM	Invalid number of dimensions
\$ERR_INVEXFTYP	Invalid external function data type
\$ERR_INVHDL	Invalid memory handle
\$ERR_INVPRC	Invalid fractional precision
\$ERR_IORDXP	Only integer and decimal operands allowed
\$ERR_IRNDVAL	Invalid round value for integer operand
\$ERR_HSZERR	Map outside bounds of field or handle
\$ERR_ODIGXP	Octal digits expected in argument
\$ERR_RNDVAL	Invalid round value
\$ERR_SETTYP	SET data types must be the same

► **Synergex.SynergyDE.SynException**

\$ERR_BADADDR	Bad address detected: %s
\$ERR_BADDBGPORT	Invalid debug port number: %s. Must be an integer within the range 1024 to 65535, inclusive
\$ERR_BADDBGTMOT	Invalid remote debug timeout value
\$ERR_BADFORMAT	Bad format string
\$ERR_BADHANDLE	Bad DLL handle
\$ERR_BIGALPHA	Alpha temporary result exceeds 65535
\$ERR_DBGCLOSED	Remote debug client closed the connection; continuing without debug
\$ERR_DBGNOCONN	No debug client connection was established
\$ERR_DBGNOSOCK	Unable to attach to remote debug port
\$ERR_DBGSOCKER	Remote debug socket error; continuing without debug
\$ERR_DLLCLSERR	DLL could not be closed
\$ERR_DLLOPNERR	DLL could not be opened
\$ERR_DLLOPNMOD	Associated DLL not in path or not found
\$ERR_GBLNF	Cannot access named global
\$ERR_INVACT	Invalid action for XCALL FATAL
\$ERR_INVCLLSEQ	Invalid calling sequence
\$ERR_INVDATE	Invalid date
\$ERR_INVFORENT	Invalid entry to FOR loop
\$ERR_INVPNHAND	Invalid pen handle
\$ERR_INVRCBHND	Invalid RCB handle
\$ERR_INVRPTHND	Invalid report handle
\$ERR_LIBMAX	Exceeded maximum open libraries
\$ERR_MAXPRC	Too many processes
\$ERR_MSGNOTFND	Error message number %d not found or internal failure
\$ERR_NOCALL	Return with no CALL or XCALL
\$ERR_NODBGPORT	Debug port number not specified
\$ERR_NODBLOPT	DBLOPT %d is obsolete, use %s

Error Handling

Exception Classes

\$ERR_NOFORK	Cannot fork
\$ERR_NOTPNHND	Handle is not a pen handle
\$ERR_NOTRCBHND	Handle is not an RCB handle
\$ERR_NOTRPTHND	Handle is not a report handle
\$ERR_NOXCAL	Undefined XCALL referenced
\$ERR_PURGE	DCL purge error
\$ERR_RECEXTCAL	Recursive XCALL
\$ERR_UNDEFERR	Undefined error
\$ERR_UNSUP	Unsupported command
\$ERR_WNFCERR	Windows API function failure
\$ERR_WINRSRC	Windows resource exhausted

► Synergex.SynergyDE.SynIOException

\$ERR_BACKPEND	Backup mode is On
\$ERR_BADKEY	Illegal key specified
\$ERR_BADXDLF	Bad XDL file
\$ERR_BADXDLS	Bad XDL string
\$ERR_BLKSIK	Invalid value specified for BLKSIK
\$ERR_CHNEXC	Too many files open
\$ERR_CHNUSE	Channel is in use
\$ERR_EXQUOTA	Exceeded quota
\$ERR_ILLCHN	Illegal channel number specified
\$ERR_IOFAIL	Failure during I/O operation
\$ERR_IOMODE	Bad mode specified
\$ERR_LPQERR	LPQUE failed
\$ERR_MAXIF	Too many input files open
\$ERR_MRGERR	Merge error
\$ERR_MSGFAIL	SEND/RECV message failure
\$ERR_NOFDL	Invalid open mode for FDL usage
\$ERR_NOOPEN	Channel has not been opened

\$ERR_NOTAVL	Device not available
\$ERR_ONLYWR	Attempt to open output device in input mode
\$ERR_OUTRDO	Output to read-only device
\$ERR_QUEUEENOTAV	Invalid queue specified on LPQUE
\$ERR_SMERR	SORT or MERGE error
\$ERR_SRTFAI	SORT failure
\$ERR_SYNSOCK	Synsock error %d
\$ERR_UPDNFD	Update of non-file device

► **Synergex.SynergyDE.SynServerException**

\$ERR_CLNTERR	Client server error
\$ERR_NETCONFIG	Local network configuration error
\$ERR_SRVLICERR	Licensing error on server %s
\$ERR_SRVLICTIMO	Licensing timed out on server %s
\$ERR_SRVRLICNS	Server license limit reached on %s
\$ERR_XFBADARRAY	Error mapping array element
\$ERR_XFBADMTHID	Method ID too long
\$ERR_XFBADPKTID	Incorrect packet identifier
\$ERR_XFBADPKT	Packet format error
\$ERR_XFBADTYPE	Invalid parameter type
\$ERR_XFHALT	Fatal error occurred on server
\$ERR_XFINCALL	Remote call already in progress
\$ERR_XFIOERR	File I/O error occurred on server
\$ERR_XFMETHCRYPT	Method requires encryption
\$ERR_XFMETHKNF	Method key not found
\$ERR_XFNOCALL	No current call in progress
\$ERR_XFNOCDT	Unable to open method catalog file
\$ERR_XFNOCMPDT	Unable to open method parameter file
\$ERR_XFNOELB	Unable to open ELB file
\$ERR_XFNOINIT	RX_DEBUG_START called without RX_DEBUG_INIT

<code>\$ERR_XFNOSPT</code>	Feature not supported in this version
<code>\$ERR_XFNUMPARMS</code>	Invalid parameter count
<code>\$ERR_XFREQPARM</code>	Required parameter not sent
<code>\$ERR_XFRTNNF</code>	Cannot access remote routine
<code>\$ERR_XFUNKERR</code>	Unknown error reported by xfServerPlus

▶ **Synergex.SynergyDE.UnknownHostException**

<code>\$ERR_BADHOST</code>	Unknown host "%s" in server spec

▶ **Synergex.SynergyDE.UsernameException**

<code>\$ERR_BADUSER</code>	Bad username, login rejected on %s

▶ **Synergex.SynergyDE.WNDException**

<code>\$ERR_BADFONTID</code>	Invalid font ID specified
<code>\$ERR_BADFONTNAM</code>	Invalid font name specified
<code>\$ERR_BADWNDID</code>	Window %d bad or no longer open
<code>\$ERR_DUPFONTNAM</code>	Duplicate font name specified
<code>\$ERR_INVWNDHND</code>	Invalid window handle
<code>\$ERR_NOTWNDHND</code>	Handle is not a window handle
<code>\$ERR_WNDERR</code>	Window Manager error

▶ **System.ArgumentNullException**

<code>\$ERR_NULARG</code>	Improper use of null argument

▶ **System.IndexOutOfRangeException**

<code>\$ERR_OUTRNG</code>	Value out of range

▶ **System.DivideByZeroException**

<code>\$ERR_DIVIDE</code>	Attempt to divide by zero

▶ **System.OutOfMemoryException**

<code>\$ERR_NOMEM</code>	Not enough memory for desired operation

▶ **System.OverflowException**

<code>\$ERR_BIGNUM</code>	Arithmetic operand exceeds maximum size

▶ **System.InvalidCastException**

<code>\$ERR_INVCAST</code>	Invalid cast

- ▶ **System.MemberAccessException**
\$ERR_MEMACC Cannot access class member
- ▶ **System.FieldAccessException**
\$ERR_FLDACC Cannot access class field
- ▶ **System.MethodAccessException**
\$ERR_METHACC Cannot access class method
- ▶ **System.MissingMemberException**
\$ERR_MISSMEM Trying to access a missing class member
- ▶ **System.MissingFieldException**
\$ERR_MISSFLD Trying to access a field that does not exist
- ▶ **System.MissingMethodException**
\$ERR_MISSMETH Trying to access a method that does not exist
- ▶ **System.InvalidOperationException**
\$ERR_INVOPER Call is invalid for object's current state
- ▶ **System.NullReferenceException**
\$ERR_NULLREF Attempt to dereference a null object reference

7

Understanding Routines

Writing Subroutines and Functions 7-2

Explains how to write and call Synergy Language subroutines, functions, and ^VAL functions.

System-Supplied Subroutines and Functions 7-9

Briefly describes the routines that are distributed as part of Synergy Language.

Argument Type Mapping 7-12

Explains how the data type of an argument may change when it is passed to a routine.

How Arguments and Parameters Are Handled 7-14

Discusses actual and activation arguments and formal parameters, and explains how they are handled when making calls to Synergy™ routines.

Writing Subroutines and Functions

Synergy Language routines can be divided into three categories: subroutines, functions, and ^VAL functions. This section describes the differences between these three types of routines, how to write them, and how to call them.

For information on the subroutines and functions that are supplied with Synergy Language, see [“System-Supplied Subroutines and Functions” on page 7-9](#).

Subroutines

A subroutine is a module of code called by a main routine, another subroutine, or a function. Synergy Language has two kinds of subroutines: *external* and *internal*.

External subroutines

An external subroutine is a routine that is separate from the routine that calls it. An external subroutine can exist in the same source file as the calling routine or in a separate source file.

To create an external subroutine, use the SUBROUTINE statement. Use either the RETURN or XRETURN statement to return to the calling routine. We recommend using XRETURN because it overrides any nested internal subroutine calls. If there isn't an explicit exit control statement, a STOP statement is implied at the end of a routine. See [SUBROUTINE-ENDSUBROUTINE on page 4-182](#) and [XRETURN on page 4-205](#) for more information.

External subroutine parameters are listed immediately after the SUBROUTINE statement using a format similar to that of field declarations. The subroutine defines the data type of each parameter, but the calling program defines the argument size. You cannot define initial values in parameter declarations. See [“Parameter definitions” on page 2-33](#) for more information.

The example below shows the external subroutine **mult**.

```
subroutine mult
    a_result ,n                ;Subroutine data division
    a_arg1   ,n
    a_arg2   ,n

proc
    a_result = a_arg1 * a_arg2 ;Set the return result
    xreturn   ;Return to the calling routine
endsubroutine
```

External subroutines are called using the XCALL statement. You can also call them as functions (i.e., in the form *%subroutine*), provided the first argument being passed contains the result of the operation and the subroutine is declared as an external function in your code. A subroutine whose first parameter is an alpha cannot be used as a function.



You can call a function (or a subroutine that has been defined as a function in an **EXTERNAL FUNCTION** statement) without using the leading %. Be careful however, because if the function does not exist but is the name of a local variable, the local variable will be used instead.

In the example below, the subroutine **mult** is called using the **XCALL** statement:

```
main
record
    retval    ,i4

proc
    xcall mult(retval, 3, 4)      ;Call mult subroutine as a subroutine
endmain
```

In the following example, the subroutine **mult** is called as a function. Note that we declare the subroutine as a function in order to call it as one.

```
main
record
    retval    ,i4
external function
    mult      ,i          ;Declare subroutine as function

proc
    retval = %mult(3, 4)    ;Call mult subroutine as a function
    stop
endmain
```

An external subroutine can be called from a C routine if that C routine was called from a Synergy Language routine. It can be called directly on OpenVMS; on Windows and UNIX, it can be called using the C interface. See [chapter 21, “Synergy Language C Interface,”](#) for more information.

Internal subroutines

An internal subroutine is located in the same function or external subroutine in which it is called (hence the designation “internal”). It resides between the calling routine’s **PROC** statement and its corresponding **END** statement, and it begins with a label and ends with the **RETURN** statement. (See [RETURN on page 4-164](#) for more information.) No arguments are passed to an internal subroutine; it shares the calling routine’s data.

Internal subroutines are called using the **CALL** statement. Upon returning from the call, processing continues at the line that follows the **CALL** statement. (See [CALL on page 4-16](#) for more information.)



When using TRY-CATCH-FINALLY or BEGIN-END blocks with local data declarations, calls to a prior scope are not allowed and generate a “Label *label* out of scope” error (LBLSCOPE). Only calls within the current scope or to a nested scope are allowed.

The example below shows the internal subroutine *square*.

```
main main
record work          ;Main routine's data division
    result    ,d5
    val       ,d3, 19

proc
    call square      ;Call to internal subroutine
    stop             ;Logical end of program
square,              ;Start of internal subroutine square
    result = val * val
    return           ;End of subroutine, return to line after call
endmain
```

Functions

A function is a module of code that returns a value. It can be called by a main routine, another function, or a subroutine. Function names are preceded by a percent sign (%).

You can use a function in an expression any place a literal can appear. For example:

```
value=%function(arguments)
```

```
if (%function(arguments))
```

```
xcall subroutine(arg1, %function(arguments), arg3)
```

A function can return a value of any data type directly to the calling program. However, what appears to be a return value to the left of the equal sign is actually passed as an additional implied argument in front of the actual arguments. That is, Synergy Language functions use the notation of a function but are not true functions. (See “[^VAL functions](#)” on page 7-6 for information on true functions.) For example, this call:

```
retval = %myRoutine(arg1, arg2)
```

is processed by the compiler as if it were this call:

```
xcall myRoutine(retval, arg1, arg2)
```

To create a function, use the **FUNCTION** statement, followed by the function's data and procedure divisions. Use the **FRETURN** statement to return to the calling routine. If there isn't an explicit exit control statement, a **STOP** statement is implied at the end of a routine. See [FUNCTION-ENDFUNCTION on page 4-72](#) and [FRETURN on page 4-71](#) for more information.



Take care when naming your functions. A user-defined function with the same name as a system-supplied intrinsic function will replace the intrinsic function.

Function parameters are listed immediately after the **FUNCTION** statement using a format similar to that of field declarations. The function defines the data type of each parameter, but the calling program defines the argument size. You cannot define initial values in parameter declarations. See ["Parameter definitions" on page 2-33](#) for more information.

The example below shows the function **mult**.

```
function mult
    a_arg1    ,n
    a_arg2    ,n
record
    result    ,i4

proc
    result = a_arg1 * a_arg2
    freturn result
endfunction
```

To use a nonprototyped, nonlocal function in a program, you must declare it with the **EXTERNAL FUNCTION** statement and specify the data type of the return value. (See [EXTERNAL FUNCTION-ENDEXTERNAL on page 4-55](#) for more information.) In the example below, we call the function **mult**.

```
main
record
    retval    ,i4
external function          ;Declare function and specify return type
    mult      ,i

proc
    retval = %mult(3, 4)    ;Call mult function
endmain
```

Alternatively, you can run the Synergy Prototype utility (**dblproto**) with the **-out=namespace** option on your nonlocal functions, and then **IMPORT** the specified namespace into your program file. This enables you to use the functions without having to use the **EXTERNAL FUNCTION** statement.

A function can also be called as a subroutine by placing the return value as the first argument in the argument list. (This is possible because, as described above, Synergy Language functions pass the return value as an argument.)

In the following example, the function **mult** is called as a subroutine.

```
main
record
    retval    ,i4
proc
    xcall mult(retval, 3, 4)      ;Call mult function as a subroutine
endmain
```

When your routine needs to return only one value, it is more efficient to write it as a function than as a subroutine. As shown in the examples below, using a subroutine requires an additional line of code.

Subroutine:

```
xcall sroutine(retval, arg1)
if (retval) ...
```

Function:

```
if (%froutine(arg1)) ...
```

A function can be called from a C routine if that C routine was called from a Synergy Language routine. It can be called directly on OpenVMS; on Windows and UNIX, it can be called using the C interface. See [chapter 21, “Synergy Language C Interface,”](#) for more information.

^VAL functions

A ^VAL function is a “true” function. It returns an integer value that is the natural integer size of the machine. That is, for a 32-bit machine, it returns an **i4**; for a 64-bit machine, it returns an **i8**.

^VAL functions are similar to non-^VAL functions in that they start with a % and can be used in an expression anywhere a literal can occur. The primary difference between a ^VAL function and a non-^VAL function is in how the return value is handled. As described in the preceding section, in a non-^VAL function, the return value is really passed as an additional argument. In a ^VAL function, the return value is a “true” return value, not an argument. (The arguments passed in a ^VAL function correspond to the arguments as they are defined.)

Because the return value is not processed as an argument, a ^VAL function is more efficient than a regular function. For this reason, we recommend using ^VAL functions for all routines that return an integer. However, because the return value of a ^VAL function must be an integer that is the natural integer size of the machine, you have to use a regular function in cases where a different type of return value is required, such as alpha or even **i8** on a 32-bit machine. Note that if the return value is not an integer, it is converted to one.

To create a ^VAL function, use the FUNCTION statement with the ^VAL qualifier, followed by the function's data and procedure divisions. Use the FRETURN statement to return to the calling routine. If there isn't an explicit exit control statement, a STOP statement is implied at the end of a routine. See [FUNCTION-ENDFUNCTION on page 4-72](#) and [FRETURN on page 4-71](#) for more information.

The example below shows the function **mult** written as a ^VAL function.

```
function mult, ^val
    a_arg1    ,n
    a_arg2    ,n
record
    result    ,i4

proc
    result = a_arg1 * a_arg2
    freturn result
endfunction
```

To use your ^VAL function in a program, declare it with the EXTERNAL FUNCTION statement and specify ^VAL as the access type. For example:

```
main
record
    retval    ,D_NATINT
external function
    mult      ,^val                                ;Specify ^VAL as the access type

proc
    retval = %mult(3, 4)                            ;Call mult
endmain
```



If you use the -X compiler switch, you do not need to declare ^VAL functions with the EXTERNAL FUNCTION statement. See [“Compiling a Synergy Language Routine”](#) in the “Building and Running Synergy Language Programs” chapter of *Synergy Language Tools* for more information.

Understanding Routines

Writing Subroutines and Functions

You can also call a ^VAL function as a subroutine. To do this, make the call with XCALL, and then call XSTAT to retrieve the ^VAL return value. This method is not as efficient because you must make two XCALLs. For example:

```
main
record
    retval      ,D_NATINT

proc
    xcall mult(3, 4)          ;Call ^VAL function mult as a subroutine
    xcall xstat(retval)      ;Retrieve the ^VAL return value
endmain
```

The true return value can also be retrieved with a subroutine by returning the value with the XRETURN statement and then making a call to XSTAT to retrieve it. See [XSTAT on page 9-394](#) for more information and an example.

System-Supplied Subroutines and Functions

Synergy Language provides a set of prewritten external subroutines, functions, and data reference operations. For detailed descriptions of these routines, refer to the following chapters:

- ▶ Chapter 9, “System-Supplied Subroutines, Functions, and Classes”
- ▶ Chapter 10, “Synergy ActiveX API”
- ▶ Chapter 11, “Synergy DLL API”
- ▶ Chapter 12, “Synergy Floating Point API”
- ▶ Chapter 13, “Synergy HTTP Document Transport API”
- ▶ Chapter 14, “Synergy Symbol Table API”
- ▶ Chapter 15, “Synergy Routine Call Block API”
- ▶ Chapter 16, “Synergy Socket API”
- ▶ Chapter 17, “Synergy Windows Printing API”
- ▶ Chapter 18, “Synergy XML API”
- ▶ Chapter 19, “Synergy Windowing API”
- ▶ Chapter 20, “Synergy .NET Assembly API”

System-supplied subroutines

If you are using subroutines from one of the Synergy Language APIs, refer to the appropriate chapter to see if there are files you need to `.INCLUDE` or ELBs that you must link with.

System-supplied subroutines have the same characteristics and are called in the same way as subroutines you write yourself. See [“External subroutines” on page 7-2](#) for more information.

System-supplied functions

There are two categories of system-supplied functions: intrinsic and external. The *intrinsic functions* are defined internally and are part of the runtime. The *external functions* must be declared with the `EXTERNAL FUNCTION` statement before they can be used. Some of the Synergy Language API functions are external. When using functions from any of the Synergy Language APIs, refer to the appropriate chapter for any special requirements, such as files that must be included or the need to declare the functions.

Understanding Routines

System-Supplied Subroutines and Functions

Generally speaking, system-supplied functions have the same characteristics as functions you write yourself. (See “[Functions](#)” on page 7-4 and “[^VAL functions](#)” on page 7-6.) There are, however, some differences that you should be aware of.

- ▶ You cannot call system-supplied functions as subroutines, unless the subroutine syntax is explicitly specified in the documentation for that function. (For example, the syntax for [%DATE](#) on page 9-77 shows both formats.)
- ▶ Most system-supplied functions that return a numeric data type return an integer rather than a decimal. If you pass an intrinsic function’s return value to a subroutine that is expecting a decimal, as in the following example,

```
xcall sub(^size(fld))
```

you must either declare the corresponding argument an integer (**i**) or numeric (**n**) data type, or use the numeric argument compiler option to map all decimal-type arguments to numeric.

The following two points apply to intrinsic functions only:

- ▶ In addition to using intrinsic functions any place a literal can appear, you can also use an intrinsic function in a data division expression as long as it can be evaluated at compile time. For example, if an intrinsic function used in a data division expression includes a variable, it cannot be evaluated at compile time and will result in an error.
- ▶ Intrinsic functions that return integer results default to size **i4** for 32-bit machines and **i8** for 64-bit machines. If the value doesn’t fit into an **i4** value, **i8** is used.

Data reference operations

A data reference operation either allows data to be accessed as any valid data type or determines data characteristics. Each data reference operation name is preceded by a caret (^). Although all data reference operations except ^M can also be referenced as functions (for example, you can reference ^ARG as %ARG), we recommend that you always use a caret.

Data reference operations that return integer results default to size **i4** for 32-bit machines and **i8** for 64-bit machines. If the value doesn’t fit into an **i4** value, **i8** is used.

Data reference operations can be divided into four categories:

- ▶ Casting functions, which allow data to be accessed as any valid data type regardless of the actual data type with which it was declared:

^A	^ARGN	^I
^ARG	^D	^M
^ARGA	^F	^PASSTHRU

- ▶ Data conversion operations, which convert a piece of data and return a specified type:

^B ^X
^O

- ▶ Compile-time functions:

^ARGNUM ^VARIANT
^DEFINED

- ▶ Data characteristic operations, which return data characteristics such as the length of an expression or whether an argument has been passed:

^ADDR ^ARGTYPE ^PASSED
^ARGDIM ^DATATYPE ^SIZE

Argument Type Mapping

The data type of a parameter is declared in the routine (see “[Parameter definitions](#)” on page 2-33). However, a variable passed to a routine parameter may have been declared with a different data type in the calling routine. In this case, the data type actually used in the routine may not be what you would expect.

Keep the following rules in mind regarding what happens to argument data types when passing arguments to routines:

- ▶ If the parameter in the routine is defined as numeric (**n**), the data type of the passed-in variable will be used in the routine. If the parameter is passed to another routine, the caller’s data type is passed to the called routine.

For example, if a decimal variable is passed to a routine where the parameter is defined as numeric, the parameter will be treated as a decimal within the routine. If an implied-decimal (**d.**) variable is passed, it will be treated as implied-decimal. We recommend that you define parameters as **n** rather than **d**, **d.**, or **i** to ensure that the passed-in data retains its data type. You can use the numeric parameter compiler option to convert all decimal-type arguments to numeric. See “[Compiler options](#)” in the “Building and Running Synergy Language Programs” chapter of *Synergy Language Tools*.

- ▶ If the parameter in the routine is explicitly typed (**a**, **d**, **d.**, **i**, **p**, **p.**), that type will be used within the routine, regardless of the data type in the calling routine. If this parameter is passed as an argument to another subroutine, it is passed as the defined type, not the type passed by the caller.

For example, if a decimal variable of four decimal digits is passed to a routine where the parameter is defined as integer, the argument will be treated as an **i4** variable by interpreting the passed variable as binary data—with no conversion within the routine.

- ▶ If the parameter in the routine is defined as implied-numeric (**n.**), the compiler will ensure that the intermediate results of any expression that uses this variable will result in an implied result. This also applies to a **d** variable passed to a parameter defined as **d.** and a **p** variable passed to a parameter defined as **p.**



If the passed-in argument is implied-decimal and the parameter in the routine is defined as decimal or integer, the whole number part is used after the fractional part is rounded. You can change the program default to truncate by setting system option #11. Specifying the **ROUND** or **TRUNCATE** option on a **MAIN**, **FUNCTION**, or **SUBROUTINE** statement overrides the default rounding behavior for that routine (including if system option #11 is set). See [#11](#) in the “System Options” chapter of *Environment Variables and System Options* for information on system option #11; see [chapter 5, “Preprocessor and Compiler Directives,”](#) in this manual for information on compiler directives.

If a subroutine passes the data to a second subroutine, the rules defined above still apply. For example, say you have two subroutines, **sub1** and **sub2**. In **sub1**, **arg1** is declared as alpha; in **sub2**, **arg1** is declared as numeric. If **var** is defined as a decimal in the calling program, and then passed to **sub1** as **arg1**, it will be treated as an alpha within **sub1**. If **sub1** then passes **var** to **sub2** as its **arg1**, it will be treated as an alpha within **sub2**.



If your routine has multiple optional arguments and you want to make use of strong prototyping, you can specify the VARARGS modifier in your routine definition to enable you to pass more arguments than are declared without causing problems. (Note that if you use VARARGS, you will not be able to run your code on versions of Synergy prior to 9.1.1.)

How Arguments and Parameters Are Handled

In [“Writing Subroutines and Functions” on page 7-2](#), we discussed briefly how arguments are treated in subroutines, functions, and ^VAL functions. This section provides more detail about how arguments are passed and what happens when you make a routine call. For purposes of this discussion, we have divided routine arguments and parameters into three categories:

- ▶ Actual arguments
- ▶ Formal parameters
- ▶ Activation arguments

Actual arguments

Actual arguments are the arguments that you specify when you call a subroutine or function. For example, in the following calls, **1**, **“abcd”**, **var1**, and **4** are actual arguments:

```
xcall instr(1, "abcd", var1)
retval = %string(4)
```

Formal parameters

Formal parameters are the parameters declared in a subroutine or function. For example, in this declaration, **arg1** and **arg2** are formal parameters:

```
subroutine sub
    arg1      ,a
    arg2      ,d
```

Activation arguments

Activation arguments are the arguments referenced internally by a Synergy routine at runtime.

In subroutines and ^VAL functions, the actual arguments become the activation arguments. For example, in this call, the actual arguments **1**, **“abcd”**, and **var1** become the activation arguments:

```
xcall instr(1, "abcd", var1)
```

In non-^VAL functions, the return value is passed as an argument, becoming the first activation argument. For example, in this call, the return value **retval** and the actual argument **4** become the activation arguments:

```
retval = %string(4)
```

See [“Argument mapping during routine calls” on page 7-15](#) for more detail.

Argument mapping during routine calls

This section shows how calling subroutines and functions in the various supported ways affects how the actual and activation arguments and formal parameters are mapped. We also explain what goes wrong when you call a subroutine or function in an unsupported way.

Calling a subroutine as a subroutine

When calling a subroutine with the XCALL statement, the actual, formal, and activation mapping is straightforward. The first actual argument corresponds to the first activation argument, which corresponds to the first formal parameter. The second actual argument corresponds to the second activation argument, which corresponds to the second formal parameter, and so on.

For example, if this subroutine:

```
subroutine mysub
  arg1      ,n
  arg2      ,n
proc
  .
  .
  .
  xreturn
endsubroutine
```

is called with the XCALL statement like this:

```
xcall mysub(var1, var2)
```

the actual, activation, and formal mappings are like this:

Actual		Activation		Formal
var1	—————	1	—————	arg1
var2	—————	2	—————	arg2

Understanding Routines

How Arguments and Parameters Are Handled

Calling a function as a function

Because the return value of a Synergy function is not a “true” return value (see “[Functions](#)” on [page 7-4](#)), when you call a function by preceding it with a percent sign (%), the first actual argument corresponds to the second activation argument, which corresponds to the second formal parameter. The second actual argument corresponds to the third activation argument, which corresponds to the third formal parameter, and so on. The FRETURN statement corresponds to the first activation argument.

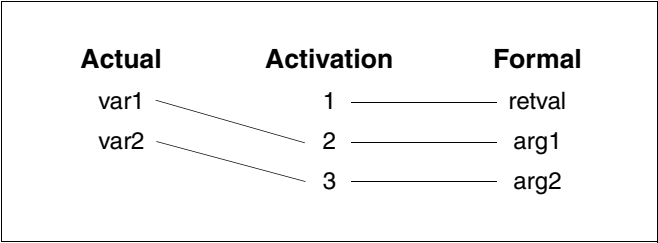
For example, if this function:

```
function myfunc
    arg1      ,n
    arg2      ,n
record
    retval    ,i4
proc
    .
    .
    .
    freturn retval
endfunction
```

is called like this:

```
result = %myfunc(var1, var2)
```

the actual, activation, and formal mappings are like this:



Calling a subroutine as a function

As discussed in “[External subroutines](#)” on [page 7-2](#), you can call a subroutine as a function provided that the first actual argument contains the results of the operation (and the subroutine is declared as a function in your code).

For example, if this subroutine:

```
subroutine mysub2
    retval    ,i4
    arg1      ,n
    arg2      ,n
proc
.
.
.
    xreturn
endsubroutine
```

is called as a function like this:

```
result = %mysub2(var1, var2)
```

the actual, activation, and formal mappings are like this:

Actual		Activation		Formal
result	—————	1	—————	retval
var1	—————	2	—————	arg1
var2	—————	3	—————	arg2

However, if the first actual argument is not the result of the operation, you should *not* call the subroutine as a function because the arguments will not map properly.

For example, if you were to call this subroutine:

```
subroutine mysub
    arg1      ,n
    arg2      ,n
proc
.
.
.
    xreturn
endsubroutine
```

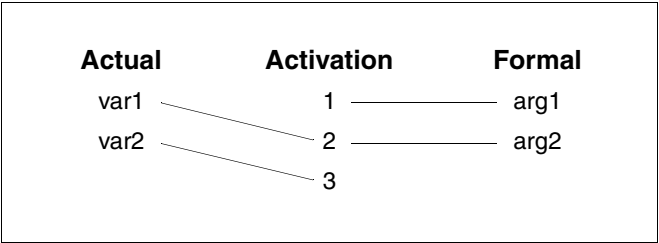
Understanding Routines

How Arguments and Parameters Are Handled

like this:

```
if (%mysub(var1, var2))
```

the actual, activation, and formal mappings are like this:



In a function call, the first activation argument is reserved for the return value of the function. As the mapping above shows, within the routine **arg1** would be null and **arg2** would use the value passed by **var1**. The value in **var2** would be “stranded” because there is no corresponding formal parameter. Depending on the routine, you may be able to retrieve this value with `^ARG` or `^ARGN`, or you may receive an error.

Calling a function as a subroutine

If you pass the function return value as the first argument of the subroutine, you can call a function as a subroutine. (See [“Functions” on page 7-4](#) for more detail.)

For example, if this function:

```
function myfunc
  arg1      ,n
  arg2      ,n
record
  retval    ,i4
proc
  .
  .
  .
  freturn retval
endfunction
```

is called like this:

```
xcall myfunc(result, var1, var2)
```

The arguments will map like this:

Actual		Activation		Formal
result	—————	1	—————	retval
var1	—————	2	—————	arg1
var2	—————	3	—————	arg2

Failure to pass the return value as the first argument will cause argument mapping problems. For example:

Actual		Activation		Formal
var1	—————	1		arg1
var2	—————	2	—————	arg2
		3	—————	

Within the routine, the value of **arg2** will be null. Upon returning from the routine, FRETURN returns the value in the first activation argument. Depending upon the routine and the type of data passed, you will more than likely receive some type of error.

Calling a ^VAL function as a function or a subroutine

Unlike regular functions, in ^VAL functions the return value is a “true” return value; it is *not* treated as an argument. (See “^VAL functions” on page 7-6 for details.)

For example, if this ^VAL function:

```
function myVfunc, ^val
    arg1      ,n
    arg2      ,n
record
    retval    ,i4
proc
    .
    .
    .
    freturn retval
endfunction
```

is called like this:

```
result = %myVfunc(var1, var2)
```

the arguments will map like this:

Actual		Activation		Formal
var1	————	1	————	arg1
var2	————	2	————	arg2

The argument mapping will look the same when calling a ^VAL function as a subroutine because the return value is retrieved with XSTAT rather than being treated as an argument.

8

Understanding Objects and Classes

Namespaces 8-2

Explains how to create and import a namespace.

Classes 8-4

Describes how to create a class.

Class Members 8-5

Discusses the members that can be part of a class, including methods, subroutines, functions, structures, records, groups, fields, properties, indexers, delegates, events, nested classes, operators, constructors, and destructors.

Instantiation and Destruction 8-12

Explains how to create an instance of a class and how an instance is destroyed.

Inheriting Class Functionality and Data 8-15

Describes how to extend a class.

Accessibility 8-16

Describes the difference between public, protected, private, internal, and protected internal access to a class or class member.

Casting 8-17

Discusses explicit type conversion.

Boxing 8-19

Defines boxing and unboxing and provides examples.

Sample Object Code 8-21

Describes three sample files in your Synergy/DE™ distribution.

Namespaces

What is a namespace?

A namespace is a unique name that represents a group of related classes, subroutines, functions, and other (nested) namespaces. You can think of it as an abstract container for these members. Its primary purpose is to prevent naming collisions between classes.

Two namespaces are supplied with Synergy/DE:

- ▶ The System namespace
- ▶ The Synergex.SynergyDE namespace

along with an empty namespace called synglobal. All classes that are distributed with Synergy/DE are in one of these two namespaces. (See [“System-Supplied Classes” on page 9-403](#) for more information about these classes.)

The SYNDEFNS environment variable specifies the name of the default namespace. If SYNDEFNS is not defined, synglobal is the default namespace. Subroutines and functions that have no declared namespace are added to the default namespace. If you use the default namespace, you do not need to follow the instructions in [“Importing a namespace”](#) below, because routines in the default namespace are imported automatically. (Note that the default namespace cannot contain classes, only subroutines and functions.)

Creating a namespace

1. Add a namespace declaration to your source file using the NAMESPACE statement. (See [NAMESPACE-ENDNAMESPACE on page 4-117](#).)
2. Add one or more class declarations within the namespace. (See [“Creating a class”](#) below.) You can also add structures, functions, and subroutines to the namespace.

A namespace can be partially defined in multiple files.

Importing a namespace

Importing allows you to use namespaces that are defined at least partially in other places. If you want to reference a class defined in a namespace that is not in the current source module, you must first import the namespace that contains the class into your application. The following instructions assume that the namespace has already been exported. See [“Prototyping” on page 1-37](#) for instructions on exporting a namespace.

1. Add an `IMPORT` statement to your `.dbl` file to import all the classes in a given namespace. For example, the following imports all members that belong to the **SomeNamespace** namespace:

```
import SomeNamespace
```
2. Compile your source code. When the compiler reaches an `IMPORT` statement, all the prototypes in the directory belonging to the namespace are included in the compile.

Classes

What is a class?

A class is a named set of related data and the functionality that works on that data. A class must be part of a namespace, even if the namespace is not specifically declared within the source code.

Creating a class

The following steps are general instructions for creating a class. Note that most of these steps are optional, since a class can consist solely of data or routines, or nothing at all.

1. Add a class declaration within a namespace declaration using the CLASS statement. (See [CLASS-ENDCLASS on page 4-22](#).) If you want the class to inherit functionality from an existing class, see [“Inheriting Class Functionality and Data” on page 8-15](#).
2. Add fields within the class declaration if desired.
3. Add one or more methods within the class if desired.
4. Add constructors and a destructor if desired.
5. Add properties to the class if desired.
6. Nest other class declarations inside the class if desired.
7. Save the source file containing the class declaration and compile it into a **.dbo** file.
8. If the class is not defined in the main program file, run the Synergy prototype utility against the source file to export the new class definition. (See [“The Synergy Prototype Utility”](#) in the “General Utilities” chapter of *Synergy Language Tools*.)

Class Members

A class member is any item that can be declared within a class declaration, such as a method, a field, a property, another nested class, and so on.

Methods, subroutines, and functions

Methods

A method is a routine within a class declaration that performs an operation using the fields of that class. When you declare a method, you can specify the following:

- ▶ The method's accessibility level (public, protected, or private)
- ▶ Whether the method can be executed even if the class that contains the method hasn't been instantiated
- ▶ Whether subclasses need to implement this method
- ▶ If the method overrides an inherited method or can be overridden
- ▶ A method parameter's direction (which indicates whether it receives data, returns data, or both) and enforcement (which indicates whether the parameter is optional or required)
- ▶ How a method argument is passed



If you give a class method the same name as a Synergy statement (for example, **close**), to get the method call, you may have to further qualify the call to such a method with either the **this** initializer (for example, **this.close**) or, if it's static, with the qualified name (for example, **myns.myclass.close**). We recommend that you don't redefine statements, to avoid having to fully qualify method calls.

Method signatures

A method signature is the fully qualified name of a method, which includes the namespace name(s), class name(s), method name, argument types, and return type, in the form

namespace[.namespace_n].class[class_n].method(type[, type, ...]) rettype

where *namespace* is the namespace or nested namespace names and *class* is the class or nested class names containing the method *method*. *Type* can be one of the following:

- ▶ A Synergy type **a**, **i**, or **d**
- ▶ A structure name in the form *\$mystructure*
- ▶ An object handle in the form *@myclass*
- ▶ A real array in the form *[ntype*, where *n* is the number of strides (from 2 through 9, omitted if stride is 1) and *type* is one of the Synergy types

- ▶ A dynamic array in the form $\{n\text{type}$, where $\{n$ is supplied for each level (n is the number of strides from 2 through 9, omitted if stride is 1) and type is any of the above types
- ▶ **^VAL** or **^REF** to include arguments declared as **^VAL** or **^REF**, respectively
- ▶ The **VARARGS** modifier to indicate a variable number of arguments

Rettype is the method's return type, which in addition to the above types, includes **VOID**.

Overriding a method

Overriding a method means providing a method declaration in an inherited class that supersedes the method declaration with the same signature in the parent class. For every parent class method that you want to override, you must add a method declaration and implementation to the derived class and specify the **OVERRIDE** modifier in the **METHOD** statement. The signature for the derived class method must match the signature for the parent class method that's being overridden. In addition, the parent class method must be marked as **VIRTUAL**, **ABSTRACT**, or **OVERRIDE**.



On OpenVMS, if the class is in a shared image, you must do an **OPENELB** of the shared image so that the virtual method can be found. (See [OPENELB](#) on page 9-228 for syntax.)

Overloading a method

A method is overloaded when another method in the same class has the same name but different arguments. Overloading enables you to perform analogous operations on different types of data. Calls to methods are resolved based on the types of the arguments passed in the call, and the best match is selected from overloaded methods.

Subroutines and functions

You can add a subroutine or function to a class by enclosing the existing subroutine or function inside a class declaration.

The following rules apply to subroutines and functions within a class:

- ▶ A subroutine or function inside a class declaration is public by default. (Most other class members are private by default.)
- ▶ A subroutine or function inside a class declaration cannot be overloaded, overridden, or redeclared.
- ▶ A subroutine or function inside a class declaration is a nonstatic, re-entrant instance method.
- ▶ A function determines its return type using the first **FRETURN** value coded within the function if not specified in the function declaration.
- ▶ For functions and subroutines, parameter modifier defaults are **BYREF** and **OPTIONAL**. These modifiers can be changed as described in [“Parameter definitions”](#) on page 2-33. Note that direction is not specified by default.

Class fields, records, groups, and structures

A class can optionally contain one or more field declarations, which may or may not be encapsulated in a record declaration. For example,

```
public class myclass
  private myfield ,i4
  public record myrecord
    myfield2 ,i4
    myfield3 ,i4
  endrecord
endclass
```

A class record can either be named or unnamed. If a record identifier is declared, it represents an instance variable of that record type within the class and can be used to access fields in that record. Field identifiers are optional in a named record. However, if the record is unnamed, fields in that record must have identifiers.

You can also declare a named group within a class record. A named group and its members inherit the accessibility of the record, if specified; otherwise, they default to private.

You can optionally declare any field modifier or access modifier on a record within a class. The specified modifiers apply to all groups and fields in that record unless overridden at the field level. (See [“Field definitions” on page 2-20](#) for the syntax of a field definition and for more information about field modifiers and access modifiers.)

Neither a common record nor a global data section may be declared within a class.

The ENDRECORD delimiter within a class or routine is optional. If you don’t specify it, the next nonfield member of the class delimits the record.

Upon instantiation of a class that contains fields, all of the fields in that class are initialized. You can get or set an accessible field value from an instantiated class as follows:

```
value = var1.myfield
var1.myfield = value
```

The lifetime of a class field, class record, or class group is determined by the lifetime of the class instance. This is called instance data. There is one copy of instance data in each object instance of a given class.

When a data field is designated as STATIC, its value can be accessed (both read and write) even if the class that contains the field hasn’t been instantiated. Only one copy of static data exists, while many copies of instance data may exist. To access a static field for a class, append the class name to the beginning of the field name. For example,

```
MyClass.myfield = value
```

Classes can also contain structures, which can have accessibility modifiers to determine when they can be used. The ENDSTRUCTURE delimiter within a class or routine is optional.

Properties and indexers

A property is usually used to provide controlled access to data within a class. In referencing routines, properties can be treated as if they were class fields, with certain restrictions. These restrictions include the following:

- ▶ The CLEAR and INIT statements cannot be used on a property. Set the property to a value instead.
- ▶ IF or WHILE or USING *property=expression* must be split into two statements, because you can't call the get and set method on a property in a single statement.
- ▶ A property can't be passed to a subroutine argument other than type IN or unspecified.

To add a property to a class and then use it in a program,

1. Create a new class or edit an existing class.
2. Use the PROPERTY statement to add a property to the class defining a get accessor method and/or a set modifier method that accesses a field value.

For example,

```
public class myclass
  private myfield, i4
  public property myproperty, i4
    method get
      proc
        mreturn myfield
      endmethod
    method set
      proc
        myfield = value
      endmethod
    endproperty
  endclass
```

3. Save the file containing the class declaration and compile it.
4. Add code to use the property. For example,

```
c1, @myclass
var1, i4
c1 = new myclass
c1.myproperty = 5
var1 = c1.myproperty
```

An indexer allows methods on an object to be called as though the object is an array. An indexer has the same requirements as a property, with a few additional requirements:

- ▶ You must provide at least one argument to the indexer.
- ▶ You can execute get and set accessors for an indexer property of a class using array syntax. For example, based on the following code,

```
m, @myclass
i, i4
m = new myclass()
```

`m[2] = 23` would run the set accessor, and `i = m[2]` would run the get accessor.

- ▶ An indexer cannot be declared as `STATIC`.

See [“Indexer properties” on page 4-138](#) for an example.

Delegates

A delegate is a type-safe function pointer used by the .NET Framework that enables you to pass a method as a parameter value. See [DELEGATE-ENDDELEGATE on page 4-36](#) for more information.

Enumerations

An enumeration is a data type that consists of a set of named values. See [ENUM on page 4-48](#) for more information.

Nested classes

A nested class is a class declared within another class declaration. There is no limit to the number of nesting levels you can have.

You must use the name of the outer class as well as the nested class when referencing a nested class. In other words, the outer class name acts as if it were a namespace. For example, to create an instance of a nested class named **class3**, you would do the following:

```
p1, @class2.class3
p1 = new class2.class3( )
```

A nested class has access to any of the other members of an outer class, regardless of accessibility.

Instance data would be accessed via an instance of the containing class, while static data would be available using the class itself. For example,

```
class c1
    static    x    , i4
    y        , i4
```

```
class c2
  method      m1      ,void
  record
    h1          ,@c1
  proc
    h1 = new c1()
    c1.x = 5
    h1.y = 10
  end
endclass
endclass
```

Also see [“Nested classes” on page 4-24](#), along with the rest of the syntax specification for [CLASS-ENDCLASS on page 4-22](#).

Operators

For a list and explanation of the operators that can be declared within a class, see [“Object operators” on page 3-14](#).

Constructors and destructors

A constructor is a class member method that is called whenever an instance of the class is created. It has the same name as the class. It is common to put initialization code in a constructor.

Constructors can have parameters, and a class can have multiple constructors as long as the method signatures are different.

For example,

```
public class class2
  public method class2      ;Default constructor
  endmethod
  public method class2      ;Constructor that takes an i4
    p1,      i4
  endmethod
```

A constructor cannot have a return type. Only access modifiers (PUBLIC, PROTECTED, and PRIVATE) are allowed on a constructor. If you don't declare a constructor, the compiler automatically generates a default constructor that has public accessibility.

You can also declare a constructor initializer to run in conjunction with a constructor. The initializer can be specified as **parent**, to run a parent class constructor, or **this**, to run a current class constructor, followed by zero or more argument values. If you don't specify an initializer, the default constructor, **parent()**, is assumed.

For example, if the **mychild** class extended another class and you wanted to run its constructor that took an **i4** value, your code might look like this:

```
method mychild
    p1, i4
    parent(p1)
proc
endmethod
```

If this constructor called another constructor within **mychild**, it might look like this:

```
method mychild
    p1, i4
    this()
proc
endmethod
```

The initializer method signature must resolve to a constructor method in the parent class (if **parent** is used) or in the current class (if **this** is used). On Synergy standard, you must not create a circular reference; otherwise, you will create an object that is never destroyed.

A destructor is a class member function that is called before an instance of the class is destroyed. You can use it to perform any class-specific cleanup operations before the last reference to the class instance is cleared.

You can optionally declare a single destructor for a class by giving the method the same name as the class with a tilde (~) at the beginning. For example,

```
public class class2
    method ~class2
```

By default, a destructor is a virtual method that has protected accessibility.

Only one destructor is allowed for each class. A destructor declares no parameters and cannot return any values.

See [“Instantiation and Destruction” on page 8-12](#) for more information.

Instantiation and Destruction

Instantiating and using a class

Instantiating a class means to create an object in memory that corresponds to the form of a previously defined class. Before you instantiate a class, you must already have declared or imported it (see [“Creating a class” on page 8-4](#) or [IMPORT on page 4-94](#)). You might also have exported it to a prototype file (see [“Prototyping” on page 1-37](#)).

1. Declare a local variable that is an object handle to an instance of the class for the class type whose functionality you want to access. For example,

```
record
    x, @SomeClass
```

2. Instantiate the class, assigning the variable to the new object. This causes the constructor that matches the method signature of the call to be run. For example,

```
x = new SomeClass(someparams)
```

See [“The NEW keyword” on page 8-13](#) for more information about this syntax.

3. Using the local object handle to reference the object, call an accessible method, or access an accessible field value. The following example accesses the method **someMethod** in the object referenced by **x**:

```
val = x.someMethod(someparams)
```

The example below accesses the value for the **myfield** field in the object referenced by **x**:

```
val = x.myfield
```

4. Add code to assign an object reference to a variable. The object instance must be the same or be a derived class of the declared destination variable type. (Note that this does not create a copy of the object but adds an additional reference to the same object.)

```
y, @SomeClass
y = x
```

5. Add code to clear the object reference held by the variable when finished with the object. If this is the last reference for the object, the object is released and the destructor is run before the object is finally deleted. The following example clears the object reference assigned to the local variable **x**:

```
clear x
```

Alternatively, the object reference is implicitly cleared when the variable goes out of scope (see [“Destroying a class instance” on page 8-13](#) for more information about data going “out of scope”). Or if a variable holds an object reference (for example, A), an assignment of another object to that variable releases the reference to object A.

6. Compile the application and link to the object file or ELB that contains the class being used.

The instantiations of nested and outer classes are not connected. Therefore, if you instantiate an nested class, its outer class is not automatically instantiated, and vice versa.

The NEW keyword

When you instantiate a class, you will use the NEW keyword, as shown below:

variable = NEW *class_name* ([*arguments*])

variable

The variable name to assign to the new object.

class_name

The full name of the class to instantiate.

arguments

(optional) The argument values that you want to pass to parameterized constructors. Note that the parentheses are required even if no arguments are specified.

The passed arguments determine which class constructor will be executed, because the arguments are compared against the method signatures of the declared constructors for the class. The constructors for inheriting classes are called in hierarchy order when the class is instantiated. For example, if classC extends classB which extends classA, and classC is instantiated, Synergy/DE will run the constructors in the following order: classA, classB, and then classC.

Destroying a class instance

After the last reference to the instantiated class has been released, the destructor method for the class and any of its parents are called in reverse hierarchy order. For example, if classC extends classB which extends classA, and an instance of classC is getting destroyed, Synergy/DE will run the destructors in the following order: classC, classB, and then classA.

Some of the events that cause references to be released, which could result in the destruction of the object, are the object variable going out of scope, being cleared, or being reassigned. The conditions under which data goes out of scope are as follows:

Data	Goes out of scope when...
A local stack variable defined by the DATA statement	The end of the block is reached
A caught exception	The end of the block is reached

Understanding Objects and Classes

Instantiation and Destruction

Data	Goes out of scope when...
Stack record data	The end of a local record or routine is reached
Class data	The class is destroyed
Common and static record data	The field is cleared



Circular references can cause objects not to be released, and the majority of circular references are not found by the compiler. As an application developer, it is your responsibility to remove circular references from your code. If you do not, your program will experience memory leaks. (You can locate circular references in your code using the debugger SHOW CLASSES/WARNINGS command.)

In Synergy standard, object handles declared via the DATA statement reduce the object reference count when the block is exited, and the destructor is automatically called if the reference count goes to zero. In .NET, the object lifetime is nondeterministic and depends only on the garbage collector. Reference counting does not occur. Destructors can be called in any order and may be called after the STOP statement.

Inheriting Class Functionality and Data

An extending class inherits the following members from the immediate parent class, as well as any members that the parent inherits from its parent, in recursive fashion:

- ▶ All nonstatic methods (not constructors)
- ▶ All nonstatic fields and properties
- ▶ All nested classes

After you extend the class, you can override inherited methods and provide additional methods, fields, or properties.

To extend a class,

1. Create a class as instructed in [“Creating a class” on page 8-4](#).
2. Multiple namespaces can exist in a single source file. If the parent class is not in the same namespace as the class added in step 1, add an `IMPORT` statement to the class file to import the namespace of the parent class definition, specifying the namespace of the parent class. (If the parent class *is* in the same namespace as the class added in step 1, an explicit `IMPORT` statement is not required, because the parent class is imported implicitly.)
3. Using the `EXTENDS` modifier, add the parent class name to the class declaration.

For example,

```
public class class1 extends myparentclass
    ; put class members here
endclass
```

4. For all parent class methods or properties that you want to override, add a method or property declaration and implementation to the new (derived) class and specify the `OVERRIDE` modifier. The signature for the method must match the signature for the parent class method that’s being overridden. The parent class method must be marked as `VIRTUAL`, `ABSTRACT`, or `OVERRIDE`.
5. (optional) Add additional methods, fields, and properties to the child class that are not found in the parent class.
6. Save the source file and compile it into a `.dbo` file.
7. (optional) Run the Synergy prototype utility against the source file to export the new class definition.

If you don’t use the `EXTENDS` modifier, the class extends the system object (`System.Object`).

Constructors and destructors are not inherited.

Accessibility

When access is allowed to a class or its members, that class or member is said to be *accessible*. Accessibility is typically controlled through the use of the **PUBLIC**, **PROTECTED**, **PRIVATE**, **INTERNAL**, and **PROTECTED INTERNAL** modifiers in a class or member declaration.

- ▶ Public access is most accessible. Access to the item is not restricted. The item can be accessed anywhere, inside or outside of the class.
- ▶ With protected access, only the containing class or its derived classes can access the item.
- ▶ Private access is least accessible. Access is restricted to the containing type, which means the item can only be accessed by other members within the same class. You can make changes without worrying about affecting other things.
- ▶ Internal access is limited to the current assembly. (Synergy .NET only)
- ▶ Protected internal access is limited to the current assembly and types derived from the containing class. (Synergy .NET only)

We recommend that you use private or protected access for greater maintainability. Classes, methods, and properties are private by default, unless designated otherwise.

The types used in the signature of any member must be as accessible as the member itself. For example, you cannot have a parameter declaration or return type with private accessibility in a publicly accessible method.

You can change the accessibility of nonvirtual members in an inherited class. This means that you can specify any valid accessibility for a nonvirtual member within the inheriting class, regardless of the accessibility of the corresponding member in the parent class. For example, field **classA.fred** might be private, but if **classB** extends **classA**, you can define **classB.fred** as public.

When you **.INCLUDE** from a repository and specify the **PUBLIC** modifier, all fields are marked public. The default states for namespace members, class members, and structure members are as follows:

Member	Default access state
Namespace member	PUBLIC
Class member	PRIVATE
Structure member (in a structure that is not declared in a routine)	PUBLIC

Casting

Casting is a process in which the data type of a variable or expression is explicitly converted to another type. Synergy Language enables you to cast an object variable to a specific class type (including @class). You can also cast an integer value (for example, a Boolean) to or from an **i4** variable (for example, an enumeration).

Casting supports upcasting (converting a derived class to a more general parent class) and downcasting (converting from a parent class to a more specific class). For example, in [figure 8-1](#), you can cast **c2** to **c1** or **object** and vice versa, but you can't cast **c3** to **c2** or **c1**.

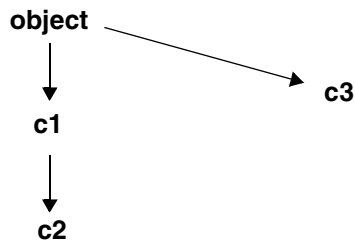


Figure 8-1. Class hierarchy.

The example below assumes ChildClass extends ParentClass and OtherClass doesn't extend either class:

```

var1, @ParentClass
var2, @ChildClass
var3, @ChildClass
var4, @ParentClass
var5, @OtherClass
var1 = new ParentClass()
var2 = new ChildClass()
var4 = (ParentClass)var2      ;Cast allowed but not necessary
var3 = (ChildClass)var2      ;Cast allowed but not necessary
var3 = (ChildClass)var1      ;Cast allowed (but if var1 contains
                             ; an instance of a ParentClass object
                             ; and not a ChildClass object, this will
                             ; fail at runtime)
var5 = (OtherClass)var1      ;Invalid cast
  
```

To reference a member of the object that has been cast, you need to wrap the cast part of the path in parenthesis. Two examples of the syntax are shown below:

```
((parent_class) object_var).field  
((parent_class) object_var).method()
```

The first example demonstrates how to access a field from an object variable that has been cast. The second shows how to call a method on a cast object variable.



If you get an “Invalid statement at or near {%s}” error (INVSTMT) or an “Invalid expression at or near {%s}” error (INVEXPR) for an expression in a complex statement that contains the cast of a path, try enclosing the full path in parentheses. For example, change

```
if (boolean)a1[1] then  
to  
if ((boolean)a1[1]) then
```

Without the parentheses, the parser interprets **(boolean)** as the condition in the IF statement and **a1[1]** as the statement to be executed after an implied THEN statement. The parentheses ensure that the full path **(boolean)a1[1]** is interpreted as the condition.

To allow casting of objects to unrelated types, you must implement an `op_Explicit` operator in your class that has a return type to which you want to support conversion. `op_Explicit` cannot be invoked directly, only through casting. The following example uses `op_Explicit` to convert from **i4** to **myclass**:

```
public static method op_Explicit, @myclass  
    p1, i4  
proc  
    mreturn new myclass(p1)  
end
```

Each conversion method must have one parameter that designates the source of the conversion and a return type to denote the destination for the conversion.

Also see [“Conversion operators” on page 3-18](#) for user-defined explicit conversions for classes.

Boxing

Boxing and unboxing are the built-in capabilities of the compiler and runtime to convert value types and descriptor types to or from objects. Note that **id** is used to define a boxed implied-decimal data type (**d.**), and **ip** is used to define a boxed implied-packed data type (**p.**).

The variable used to hold a boxed value can be declared as either explicitly typed (**@a**, **@d**, **@i4**, etc.) or untyped (**@***). Keep in mind that a typed boxed variable is processed and validated at compile-time. To box a value, you must cast it using either the boxing cast syntax (**@type**) or (**object**). When you box using the (**@type**) cast, the value is explicitly boxed as the specified type.

```
obj = (@d) dvar
```



When boxing a numeric literal, the expression type defaults to **d**. Therefore, in order to box an integer or implied-decimal, you need to explicitly cast the literal as (**@i4**) or (**@id**).

When you box using the (**object**) cast, the type inside the box is determined from the type of the expression. The runtime creates an object and then puts a copy of the value type into the object. For example,

```
obj = (object) mystructvar
```

Before you can access members of a boxed type, the boxed type must first be unboxed. Unboxing allows access to the value type inside the boxed object.

In most cases, you will need to explicitly cast the result to unbox it. However, the boxed type is automatically unboxed under the following circumstances:

- ▶ An assignment is made in which the right side's type within the box is the same as the left side's type.
- ▶ A boxed structure's field is accessed.
- ▶ A boxed type argument is passed to an unboxed type parameter. (The types must match or must both be integral or numeric.)
- ▶ A boxed variable is used in certain Synergy statements, such as I/O statements.
- ▶ An expression containing boxed operands matching the type of other operands.

Automatic unboxing requires the boxed variable to be explicitly typed.

The following example does an automatic unbox on a field access of a boxed structure:

```
record
    shnd      ,@mystruct
    svar      ,mystruct
proc
    shnd = (@mystruct) svar
    x = shnd.fld           ;Automatic unbox
```

However, for an unknown boxed type, such as

```
obj    ,@*
```

you cannot use automatic unboxing. Instead, you must explicitly cast the variable to unbox it:

```
obj = shnd  
x = ((mystruct)obj).fld    ;Unbox
```

In the example below, the automatic box of the variable **field** creates a new object of type **@i4** (the type of variable **field**) and copies in the value of the variable **field**:

```
record  
    field    ,i4  
    sfield    ,structure  
    ooi4    ,@i4  
    oosfield ,@structure  
    .  
    .  
    .  
    ooi4 = (object)field  
    oosfield = (object)sfield  
    field = (i4)ooi4  
    sfield = (structure)oosfield
```

To unbox, the runtime extracts the **i4** from the object in the example above and stores it in the variable **field**. For example,

```
field = (i4)ooi4  
sfield = (structure)oosfield
```

When an item is boxed, it is of type **@System.Object**, so only **System.Object**'s members are accessible (for example **ToString()**). Boxing is very important when using the **System.Collections.ArrayList** class, because the **ArrayList** can only deal with entities derived from **System.Object**.

Sample Object Code

Your Synergy/DE distribution includes three test files in the `dbl/examples` directory:

- ▶ **myclass.dbl** (class definitions)
- ▶ **myooprogram.dbl** (program that demonstrates many object-oriented features)
- ▶ **mystructprogram.dbl** (example of using a structfield with a global structure)

To use these example programs,

1. Run **dblproto** on **myclass** to generate the import file for the **MyNameSpace** namespace. (See “[The Synergy Prototype Utility](#)” in the “General Utilities” chapter of *Synergy Language Tools* for instructions.)
2. Compile **myooprogram** and **myclass** into **myooprogram.dbr**.

9

System-Supplied Subroutines, Functions, and Classes

System-Supplied Routines 9-9

Provides specification pages for the external subroutines that are part of Synergy Language:

^A – Access data as alpha.....	9-10
%ABS – Return the absolute value of an expression	9-11
ACCHR – Define additional activation characters.....	9-12
ACESC – Define additional activation characters.....	9-16
^ADDR – Obtain the address of a data element.....	9-17
^ARG – Access a parameter as alpha	9-19
^ARGA – Access a parameter as alpha	9-20
^ARGDIM – Return the dimension number of an argument	9-21
^ARGN – Access a parameter as numeric.....	9-23
^ARGNUM – Return the parameter position	9-24
^ARGTYPE – Return the data type of a parameter.....	9-25
ASCII – Convert a decimal value into an ASCII character.....	9-26
ASCR5 – Convert ASCII characters into RAD50 characters	9-28
ASTRST – Restore the contents of work areas used as the result of an AST	9-29
ASTSAV – Save the contents of work areas used as the result of an AST	9-30
%ATRIM – Return a string stripped of trailing blanks	9-31
^B – Interpret a string as a binary number.....	9-32
BEGFL – Position to first record.....	9-34
%BIN – Return a binary string representation of a numeric value.....	9-35
%BIT_IS_CLR – Evaluate if a bit is clear	9-37
%BIT_IS_SET – Evaluate if a bit is set	9-39
%BKSTR – Return the position of the last occurrence of a substring	9-41
BREAK – Issue a break to the channel	9-42
BTOD – Convert binary values to decimal data.....	9-43
%CHAR – Return an ASCII character for a numeric expression	9-45

CHNOPN – Check whether a channel is in use	9-46
%CHOPEN – Determine if an I/O channel is open	9-48
CMDLN – Return the command line that started your program	9-49
%CNV_IP – Convert native integer data to portable form	9-53
%CNV_PI – Convert portable integer data to native form	9-54
COMPRESS – Compress a data string	9-55
COPY – Copy file contents to another file	9-57
%CPUTIME – Return the accumulated CPU time	9-59
CPUTM – Return the accumulated CPU time	9-60
CREMBX – Create a mailbox	9-61
CVTIIV – Pass an integer value to a field	9-64
CVTIZV – Pass an integer value to a zoned numeric value	9-65
^D – Access data as decimal or implied-decimal	9-66
DACHR – Disable activation characters	9-67
DAESC – Disable activation characters	9-68
DATA_DECRYPT – Decrypt ciphertext data	9-69
DATA_ENCRYPT – Encrypt plaintext data	9-71
DATA_SALTIV – Generate a salt and/or initialization vector	9-73
^DATATYPE – Return the data type	9-75
%DATE – Return the date	9-77
%DATECOMPILED – Return the compilation date and time	9-79
%DATETIME – Return the date and time	9-80
DBL\$ subroutines	9-82
DBL\$DEVCLT – Return the class and type of a device	9-86
%DBL\$EXECUTE_IMAGE_ROUTINE – Execute a routine in a shared image	9-88
DBL\$SETKRF – Set the key of reference for the next operation on an ISAM file	9-90
DBL\$SNDOPR – Send a message to the system operator	9-91
%DECML – Convert an ASCII character to its decimal value	9-92
^DEFINED – Determine if a field or parameter has been defined	9-94
DELET – Delete the specified file	9-95
DELMBX – Mark a permanent mailbox for deletion	9-96
^DESCR – Pass an argument by descriptor	9-97
DFLAG – Set runtime option flags	9-98

DTOB – Convert decimal data to binary values.....	9-101
EMPBUFF – Write out modified I/O buffers	9-103
ENDFL – Position the file pointer after the last record of a file	9-104
ENVRN – Identify the current operating system and machine	9-105
%ERLIN – Return the line number of the statement that caused an error	9-107
%ERNUM – Return the number of the most recently trapped error	9-108
%ERR_TRACEBACK – Generate error traceback	9-109
ERRMOD – Return the module in which the most recent error occurred	9-110
%ERROR – Return the number of the most recently trapped error	9-111
ERTXT – Return an error message	9-113
EXEC – Execute a non-Synergy Language program.....	9-115
EXITE – Exit the routine with a specific error.....	9-116
EXITERROR – Exit the routine with a specific error	9-118
^F – Access variable as implied-decimal data type	9-119
%FALSE – Return a value of zero	9-120
FATAL – Chain to another program if a fatal error occurs.....	9-121
FILL – Fill a variable with a pattern.....	9-123
FILNM – Return the file specification of the open channel	9-124
FLAGS – Set runtime option flags	9-126
FORK – Split the current process	9-131
FREE – Release record locks on a channel	9-133
%FSTAT – Return the value of the last floating point call	9-134
FSTAT – Return the value of the last floating point call	9-137
FXSUBR – Dispatch to a floating-point function	9-138
GETCM – Load the contents of a common area into a variable	9-139
GETDFN – Return the default file specification	9-140
GETFA – Return file attributes	9-141
GETFNM – Return the file specification of the open channel.....	9-147
GETLOG – Translate an environment variable.....	9-148
GETRFA – Retrieve the current position in a file.....	9-150
GLINE – Get a line of input	9-153
GTPPN – Return information about the current process	9-155
%HEX – Return a hexadecimal string representation of a numeric value	9-157

^I – Access data as integer	9-158
%IMPLIED – Convert an expression to zoned numeric implied-decimal	9-159
INITPORT – Initialize a serial port	9-160
%INSTR – Search for a substring within a string.....	9-165
%INT – Return the whole number in a numeric expression.....	9-168
%INTEGER – Convert an expression to an integer value.....	9-169
ISAMC – Create an ISAM file	9-170
ISCLR – Clear an ISAM file.....	9-179
%ISINFO – Return numeric ISAM file status and key information.....	9-181
%ISINFOA – Return alpha ISAM file status and key information	9-185
ISKEY – Retrieve ISAM key information.....	9-189
ISSTS – Return status information about an ISAM file	9-192
JBNAM – Provide the image name of the calling program.....	9-195
%JBNO – Return the job number for the current process	9-196
%JPERIOD – Return the Julian period value for a date	9-198
%KEYVAL – Extract the value of an index file’s key from a record	9-201
KILL – Terminate the calling program.....	9-203
%LEN – Return the length of the data area	9-204
%LINE – Return the current line number	9-205
LOCALIZE – Localize currency	9-206
^M – Map a structure to a data area	9-208
%MEM_PROC – Implement memory management functions	9-210
MODNAME – Get the name of a routine.....	9-218
MONEY – Specify the currency symbol	9-220
%NDATE – Return a numeric date	9-221
%NUMARGS – Return the number of arguments passed to a routine	9-223
^O – Interpret a string as an octal number	9-224
%OCT – Return an octal string representation of a numeric value	9-226
OPENELB – Make an ELB’s subroutines available to executing program	9-228
%OPTION – Determine if a system option is set	9-230
%PACKED – Convert a numeric expression to packed decimal data.....	9-232
%PAINT – Return the code for the current paint character.....	9-233
PAINT – Redefine the paint character	9-234

PAK – Compress numeric data.....	9-236
PARSE – Extract components of a file specification	9-239
^PASSED – Determine whether an argument was passed	9-242
^PASSTHRU – Access a parameter as its calling type	9-243
POSRFA – Restore the saved position in an ISAM file.....	9-244
PURGE – Delete previous versions of a file	9-247
PUTCM – Put the contents of a variable into a common area	9-248
QSORT – Perform an in-memory sort.....	9-249
R5ASC – Convert a RAD50 word into ASCII.....	9-251
RANDM – Generate a random number.....	9-253
RCFLG – Control the receive option flags.....	9-255
RCVID – Change the default name referenced by RECV	9-258
%RDLEN – Return the length of the last record read	9-260
%RDTRM – Return the terminating character of the last record read	9-261
%RECNUM – Return the number of the last record processed	9-262
^REF – Pass an argument by reference	9-263
RENAM – Rename a file.....	9-264
%RFA – Return a record’s address	9-267
%RND – Round a value to the nearest whole number	9-268
%ROUND – Round a value to the nearest whole number	9-269
%RSIZE – Return the size of the last record read.....	9-270
RSTAT – Return information about the last record read	9-272
RSTATD – Return numeric information about the last record read	9-274
%RTERM – Return the terminating character of the last record read	9-275
RUNJB – Issue a system command or start a concurrent process.....	9-277
%RVSTR – Search for the last occurrence of a substring within a string.....	9-281
S_BLD – Construct a formatted string	9-283
S_DETAB – Expand tabs	9-286
S_MATCH – Match a string.....	9-288
S_PARSE – Parse a string and return information about it.....	9-290
S_WRAP – Wrap a string.....	9-294
SDMS_SEL – Set file type access behavior.....	9-296
SERIAL – Return the registration string of the current runtime	9-298

%SET_PRIORITY_CLASS – Set the priority class for a process	9-299
SETCTL – Modify the operation of control characters	9-301
SETDFN – Set the default filename extension	9-303
SETLOG – Set the value of an environment variable	9-304
SHELL – Execute the system command processor or create a subprocess	9-306
^SIZE – Return the size of an expression	9-310
SORT – Provide a callable interface to DBLSORT	9-312
SPAWN – Execute a command or program	9-314
%STRING – Convert a value to a string.....	9-318
STTY – Control terminal settings.....	9-319
%SUCCESS – Determine if low-order bit is on or off	9-320
%SYN_ATEXIT – Register a routine to be called when a service shuts down	9-321
%SYN_CHARTOSTR – Convert C-style character pointer to alpha	9-322
%SYN_ESCAPE_HANDLE – Escape characters in a string in a handle	9-324
%SYN_FREECHN – Get the next available channel.....	9-327
%SYN_GETDIR – Get the current directory path	9-328
%SYN_GETSTATE – Retrieve a Synergy state option.....	9-329
SYN_REPORTEVENT – Write entries to the event log.....	9-330
%SYN_SETDIR – Set the current directory path.....	9-331
%SYN_SETSTATE – Set a Synergy state option	9-332
%SYN_SYSERRTXT – Get the formatted system error message.....	9-334
%SYN_UNAME – Get the current username	9-335
%SYN_UNESCAPE_HANDLE – Unescape escape codes in a string in a handle	9-336
%SYN_XNAME – Get a function name from its segment address	9-337
%SYSERR – Return the status code from the last system call.....	9-338
%SYSID – Return the ID of the system where the source is being compiled.....	9-339
TFLSH – Flush the terminal buffer.....	9-341
TIME – Return the current system time.....	9-343
%TIMEZONE – Return the time zone of the current machine	9-344
%TNMBR – Return the terminal number.....	9-346
TRACEBACK – Print the current traceback	9-349
%TRIM – Return the length of a variable without trailing blanks	9-350
%TRIMZ – Return the length of a variable without trailing blanks	9-352

%TRUE – Return a value of one	9-354
%TRUNC – Truncate a numeric expression	9-355
%TRUNCATE – Truncate a numeric expression	9-356
TT_NAME_TO_NUMBER – Convert a terminal name to its number	9-357
TT_NUMBER_TO_NAME – Convert a terminal number to its name	9-358
TTBRDCST – Enable a program to trap broadcast messages.....	9-359
TTCHAR – Return type, lines, and width of a file.....	9-361
TTFLGS – Set terminal-oriented flags.....	9-364
TTMBX – Associate a mailbox with a channel opened to a terminal device	9-370
%TTNAME – Return the terminal name.....	9-371
%TTSTS – Determine whether input is pending.....	9-372
UNPAK – Decompress numeric data	9-374
%UNSIGNED – Return the unsigned binary value of a numeric expression	9-375
^VAL – Pass an argument by value or declare a value type function	9-376
^VARIANT – Return the value of an internal compiler variable.....	9-378
^VARIANT – Return the value of an internal compiler variable.....	9-378
%VERSN – Return the Synergy Language version number and operating system	9-379
VMCMD – Execute a DCL command	9-380
VMMSG – Get the text of an OpenVMS system message	9-382
%VMS – Indicate if a program is being compiled on OpenVMS.....	9-384
%WAIT – Suspend program execution.....	9-385
%WKDAY – Return the day of the week	9-387
^X – Interpret a string as a hexadecimal number	9-389
%XADDR – Return the address of a routine.....	9-391
XARGS – Return information about arguments passed to external routine	9-393
XSTAT – Get return value from last routine called	9-394
XSUBR – Invoke an external routine with the specified arguments.....	9-397
^XTRNL – Provides the value of a link-time symbol	9-400
%ZONED – Convert expression to decimal in zoned numeric representation	9-402

System-Supplied Classes 9-403

Describes the classes that are supplied as part of Synergy Language:

System.String.....	9-404
System.Object.....	9-406

System.Exception..... 9-408

System.Array..... 9-410

Synergex.SynergyDE.SynException 9-412

System.Collections.ArrayList 9-414

Synergex.SynergyDE.Collections.ArrayList 9-417

Synergex.SynergyDE.Select.Select 9-420

Synergex.SynergyDE.Select.From 9-425

Synergex.SynergyDE.Select.Where 9-427

Synergex.SynergyDE.Select.OrderBy 9-437

Synergex.SynergyDE.Select.Event..... 9-438

Synergex.SynergyDE.Select.AlphaEnumerator and RestrictedAlphaEnumerator..... 9-439

System-Supplied Routines

This chapter gives the syntax and usage for the Synergy Language system-supplied external subroutines, intrinsic functions, and data reference operations. For information on calling subroutines, functions, and data reference operations, along with details about argument typing, see [chapter 7, “Understanding Routines.”](#) Routines that are part of the Synergy Language APIs are not documented in this chapter; see [chapter 10](#) through [chapter 20](#) for information about those routines. For information on the conventions used in documenting the syntax, see [page xxix](#).

^A – Access data as alpha

`^A (expression)`

Arguments

expression

An expression to access as alpha. (**a** or **n**)

Discussion

^A accesses data as an alpha data type.

^A is useful when using decimal data in file I/O statements. ^A doesn't convert *expression* to an alpha value; the data remains unchanged. Only the data type is changed for the reference.

For example, the following fields are each four bytes long:

```
dec      ,d4,    5
int      ,i4,    1819242310
```

If **dec** is used with ^A in the following DISPLAY statement:

```
display(TTCHN, ^a(dec))
```

the following characters are displayed:

```
0005
```

Accessing an integer field as alpha is not very useful; however, the following example illustrates the result when each integer byte happens to represent an ASCII character. If **int** is used with ^A in the following DISPLAY statement (on a little-endian machine):

```
display(TTCHN, ^a(int))
```

the following characters, which represent the ASCII values of the data, are displayed:

```
Fool
```

Use %STRING to display integer values, or decimal values that can be negative.

Examples

```
reads(TTCHN, ^a(age))
writes(TTCHN, "Total Price: " + ^a(price + tax))
```

%ABS – Return the absolute value of an expression

absolute = %ABS (*number*)

Return value

absolute

The absolute value of the specified numeric expression. The result has the same data type as *number*. (**n**)

Arguments

number

A numeric expression whose absolute value is returned. (**n**)

Examples

If your algorithm uses the equation

$$y = |x - 3|$$

you would code it as follows:

`y = %abs(x - 3)`

ACCHR – Define additional activation characters

```
xcall ACCHR(character[, ...])
```

Arguments

character

One or more numeric codes, separated by commas, each representing an ASCII character. Valid codes are 0 through 255. (**n**)

Discussion

The subroutine ACCHR defines additional characters to be processed as activation characters during terminal input. (You can later deactivate the specified characters using the DACHR subroutine.)

Activation characters are only meaningful with terminal READS, the W_FIELDS(WF_INPUT) subroutine, and the W_DISP(WD_READS) subroutine. In all other situations, activation characters are ignored. If no activation characters are defined, terminal input ends when a standard record terminator is encountered. The standard terminators for terminal input are the carriage return, line feed, form feed, TAB, and ESC keys. Standard terminators cannot be undefined.

When you define activation characters using ACCHR, Synergy Language considers the specified characters to be terminal input terminators. The input operation ends when either a standard terminator or an activation character is encountered. The receiving data area does not contain the terminating character. The terminating character is always accessible from either the %RDTRM intrinsic function or the RSTAT external subroutine.

Activation characters are not echoed, which means that your program can stop input from a CRT screen without corrupting information displayed on the screen. Input from the last line of a CRT screen does not scroll the screen if the input terminator is an activation character.

When the user presses RETURN in response to a terminal input request, the system automatically generates a record terminator (a carriage return/line feed pair on Windows and OpenVMS and a line feed on UNIX). Both characters are echoed to the terminal, and the line feed, as a standard record terminator, ends the input action. The %RDTRM intrinsic function or the RSTAT external subroutine indicates that the line feed was the terminating character.

However, if the program defines the carriage return as an activation character, a different set of actions occurs. The system doesn't echo or internally generate a line feed in response to a carriage return. The carriage return ends the input request, and the %RDTRM intrinsic function or the RSTAT external subroutine indicates that the carriage return was the terminating character.

If the delete character is set as an activation character, it is ignored as an activation character in W_FIELDS(WF_INPUT).

If you try to define an activation character that is already defined, your attempt is ignored, and the character remains defined. Activation characters are reset by STOP chains.



If you use UI Toolkit, all details of managing function key processing are handled for you.

See also

[DACHR on page 9-67.](#)

Examples

```
.define TTCHN,1
record
    io          ,a30
    i           ,d3

proc
    open(TTCHN, i, "tt:")
    xcall acchr(7, 9, 13, 127)    ;bell, horizontal tab, carriage return,
                                ; rubout

loop,
    display(TTCHN, "Type the input: ")
    reads(TTCHN, io) [EOF=DONE]
    display(TTCHN, '... Input was "')
    if (%rdlen)
        display(TTCHN, io(1, %rdlen))
    io(1,3) = %rdlen [left:i]
    io(11,13) = %rdtrm, "XXX"
    write(TTCHN, '" (' + io(1,i) + ") long, terminated" +
    & " by code ", io(11,13))
    goto loop

done,
    stop
end
```

Special notes

Many CRT terminals have function keys that transmit character sequences to the computer. Because they often begin with the ASCII escape character (ESC), we refer to them as escape sequences. Your terminal user guide describes the special functions and sequences available.

In the past, programs that used these special function keys assumed the entire burden of terminal input management. They had to turn echoing on and off, receive each character using the ACCEPT statement (thus incurring the overhead of single-character input), and perform all activity on the characters received (character echoing, character/line deletions, field overflow control, deleted character repainting, and so forth). This extra coding often limited a program to a specific environment, which restricted portability.

Synergy Language simplifies the writing of compact routines to handle intricate terminal input requirements, because the runtime performs most of the required functions internally. The runtime provides functionally identical features regardless of the operating environment, which enables you to move your programs easily between operating systems.

The following example illustrates the use of the ACCHR subroutine for terminal input. Several utility functions are enabled upon input of either control characters, or VTn00 function keys. This example also illustrates the use of TTFLGS 4, which causes the Synergy runtime to parse escape sequences and return a code based upon the key.

```
main test_fkeys

record
    function          ,i4
    inp_string        ,a80

.define FUNCTION_1   ,1
.define FUNCTION_2   ,2
.define FUNCTION_3   ,3
.define FUNCTION_4   ,4
.define FUNCTION_5   ,5
.define FUNCTION_6   ,6
.define FUNCTION_7   ,7

proc
    open(1, o, "tt:")
    xcall ttflgs(4000, 1)                ;Enable parsing escape sequences
                                         ; (UNIX and OpenVMS) so we DON'T
                                         ; have to activate 27 below

    xcall acchr(1,2,4,5,6,8,10)          ;Activate CTRL+A,B,D,E,F,H,J

    reads(1, inp_string)
```



```
using %rdtrm select
(1, 294),                                ;CTRL+A or F14
    function = FUNCTION_1
(2, 274),                                ;CTRL+B or Up arrow
    function = FUNCTION_2
(4, 276),                                ;CTRL+D or Left arrow
    function = FUNCTION_3
(5),                                     ;CTRL+E
    function = FUNCTION_4
(6, 277),                                ;CTRL+F or Right arrow
    function = FUNCTION_5
(8, 292),                                ;CTRL+H or F12
    function = FUNCTION_6
(10, 293),                               ;CTRL+J or F13
    function = FUNCTION_7
(),                                       ;Undefined
    clear function
endusing

writes(1, "Function " + %string(function) + " activated")
endmain
```

ACESC – Define additional activation characters

The ACESC subroutine is functionally identical to the ACCHR subroutine. It is included for compatibility with earlier versions of Synergy Language. See [ACCHR on page 9-12](#) for a thorough discussion.

^ADDR – Obtain the address of a data element

address = ^ADDR(*variable*)

Return value

address

The address of the referenced data. (**i**)

Arguments

variable

A variable whose address will be obtained. (**a** or **n**)

Discussion

Specify the data storage for *address* with D_ADDR to ensure the variable is large enough to contain the entire address value.

Examples

```
.include "$DVIDEF" library "DBLSTARLET"

external function
    sys$getdviw      , ^val

record
    stat              , D_NATINT
    group itmlst      , [2]i
        group lw1     , i4
            len        , i2
            code       , i2
        endgroup
    bufaddr           , D_ADDR
    retlen            , D_ADDR
endgroup
tt_name              , a16

proc
    open(1, o, "tt:")
    itmlst[1].len = ^size(tt_name)
    itmlst[1].code = DVI$_DEVNAM
    itmlst[1].bufaddr = ^addr(tt_name)
    clear itmlst[1].retlen, itmlst[2].lw1
    stat = %sys$getdviw(, , "TT:", ^ref(itmlst), , , )
```

System-Supplied Subroutines, Functions, and Classes

^ADDR

```
        if (.not. (stat .band. 1))
            xcall lib$stop(^val(stat))
; Get the device name of the terminal
        writes(1, tt_name)
end
```

^ARG – Access a parameter as alpha

^ARG (position)

Arguments

position

The ordinal position of the parameter. (**n**)

Discussion

^ARG accesses the specified formal parameter as alpha data type.

^ARG is a positional parameter reference and cannot be present in a main routine. It accesses a parameter as if the parameter were declared as alpha (**a**).

When ^ARG is used, the parameters do not have to be defined in the subroutine parameter block.

Examples

```
subroutine sub
  arg1      ,a           ;Optional parameter
  arg2      ,a           ;Optional parameter
  arg3      ,a           ;Optional parameter
record
  i          ,d2

proc
  for i from 1 thru %numargs
    if (^passed(^arg(i)))
      .
      .  process parameter(i)
      .
```

^ARGA – Access a parameter as alpha

^ARGA (position)

Arguments

position

The ordinal position of the parameter. (**n**)

Discussion

^ARGA accesses the specified formal parameter as alpha data type.

^ARGA is equivalent to ^ARG. ^ARGA is a positional parameter reference and cannot be present in a main routine. It accesses a parameter as if the parameter were declared as alpha (**a**).

^ARGDIM – Return the dimension number of an argument

dim_num = ^ARGDIM(*position*, *dim_index*)

Return value

dim_num

The number of the specified dimension. (**n**)

Arguments

position

The ordinal position of the argument. (**n**)

dim_index

The index of the dimension to access. (**n**)

Discussion

If *position* specifies a nonexistent argument, an “Argument missing” error (\$ERR_ARGMIS) is generated.

If *position* specifies an argument outside of the passed range, an “Incorrect number of subroutine arguments” error (\$ERR_WROARG) is generated.

If *dim_index* specifies an invalid dimension index, an “Invalid number of dimensions” error (\$ERR_INVDIM) is generated.

If the specified argument is not dimensioned, the only valid *dim_index* value is 1, and the return value will also be 1.

Examples

```

record
    val      , [3,4,5]a2
    val2     , a5

proc
    xcall sub(val, , val2)
end

subroutine sub
    arg      , [* , * , *]a
    arg2     , a
    arg3     , a

```

System-Supplied Subroutines, Functions, and Classes

^ARGDIM

```
record
    num,      i4

proc
    num = ^argdim(%argnum(arg), 1)      ;Returns 3
    num = ^argdim(%argnum(arg), 2)      ;Returns 4
    num = ^argdim(%argnum(arg), 3)      ;Returns 5
    num = ^argdim(3, 1)                  ;Returns 1
    num = ^argdim(2, 4)                  ;Error ARGMIS (#2 not passed)
    num = ^argdim(4, 4)                  ;Error WROARG (#4 beyond end)
    num = ^argdim(%argnum(arg), 7)      ;Error INVDIM (No dimension #7)
    xreturn
endsubroutine
```


^ARGN – Access a parameter as numeric

^ARGN (position)

Arguments

position

The ordinal position of the parameter. (**n**)

Discussion

^ARGN accesses the specified formal parameter as if the parameter were declared as numeric.

^ARGN is a positional parameter reference and cannot be present in a main routine.

Examples

```
subroutine sub
  a_arg1      ,a
  a_arg2      ,a
  a_result    ,d

proc
  if (a_arg1) then
    a_result = 2 * ^argn(2)
  else
    a_result = a_arg2
  xreturn
endsubroutine
```

^ARGNUM – Return the parameter position

position = ^ARGNUM(*parameter*)

Return value

position

The ordinal position of the specified parameter. The result is integer data type. **(i)**

Arguments

parameter

The name of an parameter whose position will be returned. **(a)**

Discussion

^ARGNUM is evaluated at compile time.

^ARGTYPE – Return the data type of a parameter

arg_type = ^ARGTYPE(*parameter*)

Arguments

arg_type

One of the following return values: (i)

D_TYPE_A	Synergy data type is a .
D_TYPE_D	Synergy data type is d .
D_TYPE_ID	Synergy data type is d..
D_TYPE_I	Synergy data type is i .
D_TYPE_P	Synergy data type is p .
D_TYPE_IP	Synergy data type is p..
D_TYPE_OBJ	Synergy data type is @* or @class .

parameter

The parameter whose data type value will be returned.

Discussion

^ARGTYPE returns a numeric value for the specified parameter's data type.

Use the .IS. operator to determine a specific type of class. (See [“Object operators” on page 3-14](#) for more information about .IS..)

ASCII – Convert a decimal value into an ASCII character

```
xcall ASCII(decimal, ascii)
```

Arguments

decimal

An arithmetic expression whose value will be converted into a single ASCII character. (**n**)

ascii

A variable to be loaded with the ASCII character derived from the decimal value. (**a**)

Discussion

The ASCII subroutine converts a decimal value into a single ASCII character and stores it left-justified over blanks in a destination variable.

If the specified arithmetic expression yields a value greater than 256, Synergy Language divides the value by 256 and uses the remainder to determine the ASCII character. For example, a value of 572 is interpreted as 60.

The ASCII character that is stored corresponds to the 8-bit binary pattern represented by the decimal value. For example, a value of 67 causes an uppercase letter “C” to be stored in the destination variable.

The character is stored in the leftmost position of the destination variable. The remaining character positions of the destination variable are cleared to blanks.



%CHAR is more efficient than the ASCII subroutine for literal values, because %CHAR is evaluated at compile time.

Examples

The following program scans a stream file for a line feed embedded in a record and replaces it with a blank.

```
record
    data      ,a30
    i         ,i4
    ascii     ,a1
    loc       ,d2

proc
    open(2, i, "badfil.txt")
    xcall ascii(10, ascii)
    repeat
```

```
begin
  incr i
  gets(2, data, i) [eof=eof]
  if (loc = %instr(1, data, ascii))
    begin
      clear data(loc:1)
      put(2, data, i)
    end
  end
close 2
stop
end
```

ASCR5 – Convert ASCII characters into RAD50 characters

```
xcall ASCR5 (ascii, rad50)
```

Arguments

ascii

A literal or variable that contains three ASCII characters to be converted. **(a)**

rad50

The variable that will be loaded with two RAD50 characters, left-justified over blanks. **(a)**

Discussion

The ASCR5 subroutine converts three ASCII characters into two RAD50 characters. (RAD50 was a compression technique used on PDP-11 systems.) The ASCR5 subroutine complements the R5ASC subroutine.

ASCII-to-RAD50 conversions are valid for the following ASCII characters:

- ▶ Uppercase letters A through Z
- ▶ Digits 0 through 9
- ▶ The following special characters: dollar sign (\$), blank (), and period (.)

If the ASCII string contains any other characters, it is still converted, but it yields a RAD50 value that cannot be converted back correctly with the R5ASC external subroutine.

If the size of the ASCII value is less than three characters, an “Argument specified with wrong size” error (\$ERR_ARGSIZ) is generated. If the size is more than three characters, only the leftmost three characters are converted. If the size of the *rad50* variable is less than two characters, an “Argument specified with wrong size” error is also generated.

See also

[R5ASC on page 9-251.](#)

Examples

```
record
  asc      ,a3
  rad      ,a2

proc
  xcall ascr5("ABCDE", rad)      ;Converts "ABC"
  asc = "1A2"
  xcall ascr5(asc, rad)          ;Converts "1A2"
  stop
end
```

ASTRST – Restore the contents of work areas used as the result of an AST

VMS

```
xcall ASTRST
```

Discussion

The ASTRST subroutine restores the contents of work areas used by a Synergy Language subroutine executed as the result of an AST (asynchronous system trap).

The ASTRST subroutine is called only during the execution of an AST subroutine, and it must be the last statement executed in the subroutine before the RETURN statement.

The counterpart to the ASTRST subroutine is the ASTSAV subroutine, which must be executed at the beginning of the AST subroutine. (See [ASTSAV on page 9-30](#).)

See also

[“AST Support in Synergy Language”](#) in the “Statements, Subroutines, and Functions on OpenVMS” chapter of the *Professional Series Portability Guide*.

ASTSAV – Save the contents of work areas used as the result of an AST

VMS

```
xcall ASTSAV
```

Discussion

The ASTSAV subroutine saves the contents of work areas used by a Synergy Language subroutine executed as the result of an AST (asynchronous system trap).

The ASTSAV subroutine is called only during the execution of an AST subroutine, and it must be the first statement executed in the subroutine.

The counterpart to the ASTSAV subroutine is the ASTRST subroutine, which must be executed at the end of the AST subroutine. (See [ASTRST on page 9-29](#).)

See also

[“AST Support in Synergy Language”](#) in the “Statements, Subroutines, and Functions on OpenVMS” chapter of the *Professional Series Portability Guide*.

%ATRIM – Return a string stripped of trailing blanks

string = %ATRIM(*expression*)

Return value

string

A string stripped of its trailing blanks. If *expression* is null or only contains blanks, the return value is a null string. (a)

Arguments

expression

The alpha expression whose blanks will be stripped. (a)

Examples

The example below displays the data in the fields **name** and **comp** without any trailing blanks.

```
subroutine show_rec
record custmast
    id          ,d5
    name         ,a50
    comp         ,a50

proc
    writes(TTCHN, "Name: " + %atrim(name))
    writes(TTCHN, "Company: " + %atrim(comp))
    xreturn
endsubroutine
```

^B – Interpret a string as a binary number

binary = ^B(*expression*[, *size*])

Return value

binary

The integer value of an alpha expression, evaluated as a binary number. If *expression* is null, the result is 0. (i)

Arguments

expression

The alpha expression to interpret as binary. (a)

size

(optional) A value that specifies the size of the resulting integer. Valid values are

- 1** The result is returned in an **i1** value.
- 2** The result is returned in an **i2** value.
- 4** The result is returned in an **i4** value.
- 8** The result is returned in an **i8** value.

Discussion

If *expression* does not contain only valid binary digits (0 and 1), a “Binary digits expected in argument (string)” error (\$ERR_BDIGXP) is generated.

By default, ^B produces an **i4** value if the return value fits in 32 bits or an **i8** value if the return value is larger than 32 bits. If *size* is specified, the value is returned in an integer of that size. If the value exceeds the specified size, it is truncated to fit.

A ^B value that fits in 32 bits but has the high bit set will be negative. If you want a positive value when a negative **i4** would normally be returned, you can specify a value of 8 in the optional *size* argument.



The integer return value is of the same endian type as the executing machine. If data is to be packed and stored, then retrieved in this way, and you want to function in a multiplatform environment, your code must take this into account. Data stored on disk files should be converted to portable integer format. See [%CNV_IP on page 9-53](#) and [%CNV_PI on page 9-54](#) for more information.

Examples

The example below gets binary data for a record and stores it as bit flags.

```

subroutine get_flags
    a_bit_flags          ,i
external function
    yesno                ,d
record flags
    public               ,d1
    domestic              ,d1
    less50                ,d1
    vmail                 ,d1

proc
    public = %yesno("Public company?")
    domestic = %yesno("Domestic sales only?")
    less50 = %yesno("Less than 50 employees?")
    vmail = %yesno("Voice mail system?")
    a_bit_flags = ^b(flags)
    xreturn
endsubroutine

function yesno
    a_txt                ,a
record
    yesno                ,a1

proc
    display(TTCHN, a_txt, " (Y/N) ")
    reads(TTCHN, yesno)
    freturn (yesno .eq. 'Y')
endfunction

```

BEGFL – Position to first record

```
xcall BEGFL(channel)
```

Arguments

channel

An expression that evaluates to a channel number as specified in a previous OPEN statement. (**n**)

Discussion

The BEGFL subroutine positions the file pointer at the first record in the file. BEGFL is used with sequential, relative, stream, and indexed files.

After a BEGFL call, a READS returns the first record of the file to the DIBOL program. A WRITES writes the designated record as the first record of the sequential or relative file. Any records that are locked due to a previous operation are unlocked.

On Windows and UNIX, the BEGFL subroutine resets the key of reference for an ISAM file to zero. If system option #35 is set, the current key of reference is retained for subsequent sequential I/O. On OpenVMS, however, the key of reference is never cleared, regardless of whether or not system option #35 is set. To work around this discrepancy, you can call the SETKRF routine after the BEGFL call. (See [DBL\\$SETKRF on page 9-90](#) for more information about SETKRF.)



On Windows and UNIX, if you want to retain the key of reference and don't want to set system option #35 (which may change the behavior of other things), use GETFA(CH, "KRF", krf) followed by FIND(CH, REC, KEYNUM:krf, POSITION:Q_BOF).

An “Invalid operation for file type” error (\$ERR_FILOPT) is generated if *channel* is not associated with a file.

See also

The POSITION:Q_BOF option of [FIND on page 4-57](#) and [4-233](#).

%BIN – Return a binary string representation of a numeric value

string = %BIN(*value*[, *magnitude*])

Return value

string

The specified number as a string representation of binary characters, “0” or “1”. (**a**)

Arguments

value

The value to convert to a binary character string. (**n**)

magnitude

(optional) A value that determines the number of characters in the returned string, as follows: (**n**)

1	8-character string
2	16-character string
4	32-character string
8	64-character string

Discussion

%BIN converts a decimal data value to a 32-bit longword value.

If *magnitude* is not present or if it contains a value other than 1, 2, 4, or 8, the resulting string is 32 characters long (if *value* can be represented in 32 bits) or 64 characters (if *value* cannot be represented in 32 bits).

Examples

The following example takes binary data from a record, stores it as byte flags, and displays the results.

```
subroutine show_cflags
  a_bit_flags      ,a

record
  group flags      ,a
    unused_flg1    ,d1                ;Represents bit 0
    unused_flg2    ,d1
    unused_flg3    ,d1
    unused_flg4    ,d1
  public           ,d1
```

System-Supplied Subroutines, Functions, and Classes

%BIN

```
        domestic      ,d1
        less50        ,d1
        vmail         ,d1                ;Represents bit 7
    endgroup

proc
    flags = %bin(a_bit_flags, 1)
    if (public)
        writes(1, "Public company")
    if (domestic)
        writes(1, "Domestic sales")
    if (less50)
        writes(1, "Less than 50 employees")
    if (vmail)
        writes(1, "Voice mail system")

    xreturn
endsubroutine
```

%BIT_IS_CLR – Evaluate if a bit is clear

status = %BIT_IS_CLR(*bit*, *value*)

Return value

status

One of the following values: (**n**)

1 The bit is clear.

0 The bit is set.

Arguments

bit

The number of the bit to be evaluated. (**n**)

value

The value whose binary representation contains the bit to be evaluated. (**n**)

Discussion

%BIT_IS_CLR evaluates a bit value in the binary representation of a value.

Bit numbering is zero-based. *Bit* can be 0 through 63 (or 31 on 32-bit systems), corresponding to values 1 through 8000000000000000 (hexadecimal).



We recommend using bitwise Boolean operators rather than the %BIT_IS_CLR and %BIT_IS_SET functions.

See also

[“Bitwise operations” on page 3-13.](#)

Examples

The following example evaluates bit number 2 in the binary representation of 20:

```
%bit_is_clr(2,20)
```

Since the binary representation of 20 is 10100, and bit number 2 is actually the third bit from the right, this call to %BIT_IS_CLR returns a value of zero.

System-Supplied Subroutines, Functions, and Classes

%BIT_IS_CLR

The example below translates bit flags and displays their values to the terminal.

```
subroutine show_bflags
    a_bit_flags      ,n           ;Bit positions
    .define PUBLIC_BIT  ,4
    .define DOMESTIC_BIT ,5
    .define LESS50_BIT  ,6
    .define VMAIL_BIT   ,7

proc
    xcall showit("Public company?",
    &             %bit_is_clr(PUBLIC_BIT, a_bit_flags))
    xcall showit("Domestic sales only?",
    &             %bit_is_clr(DOMESTIC_BIT, a_bit_flags))
    xcall showit("Less than 50 employees?",
    &             %bit_is_clr(LESS50_BIT, a_bit_flags))
    xcall showit("Voice mail system?",
    &             %bit_is_clr(VMAIL_BIT, a_bit_flags))
    return
endsubroutine

subroutine showit
    a_txt      ,a           ;Text to display
    a_set      ,n           ;1 = set, 0 = not set
record
    yn         ,a1          ;Y or N

proc
    if (a_set) then
        yn = 'N'
    else
        yn = 'Y'
    display(TTCHN, a_txt, " (Y/N) ", yn)
    return
endsubroutine
```


%BIT_IS_SET – Evaluate if a bit is set

status = %BIT_IS_SET(*bit*, *value*)

Return value

status

One of the following values: (**n**)

1 The bit is set.

0 The bit is clear.

Arguments

bit

The number of the bit to be evaluated. (**n**)

value

The value whose binary representation contains the bit to be evaluated. (**n**)

Discussion

%BIT_IS_SET evaluates a bit value in the binary representation of a value.

Bit numbering is zero-based. *Bit* can be 0 through 63 (or 31 on 32-bit systems), corresponding to values 1 through 8000000000000000 (hexadecimal).



We recommend using bitwise Boolean operators rather than the %BIT_IS_CLR and %BIT_IS_SET functions.

See also

[“Bitwise operations” on page 3-13.](#)

Examples

The following example evaluates bit number 2 in the binary representation of 20:

```
%bit_is_set(2,20)
```

Since the binary representation of 20 is 10100, and bit number 2 is actually the third bit from the right, this call to %BIT_IS_SET returns a value of **1**.

System-Supplied Subroutines, Functions, and Classes

%BIT_IS_SET

The example below translates bit flags and displays their values to the terminal.

```
subroutine show_bflags
    a_bit_flags      ,n      ;Bit positions
    .define PUBLIC_BIT    ,4
    .define DOMESTIC_BIT  ,5
    .define LESS50_BIT    ,6
    .define VMAIL_BIT     ,7

proc
    xcall showit("Public company?",
    &             %bit_is_set(PUBLIC_BIT, a_bit_flags))
    xcall showit("Domestic sales only?",
    &             %bit_is_set(DOMESTIC_BIT, a_bit_flags))
    xcall showit("Less than 50 employees?",
    &             %bit_is_set(LESS50_BIT, a_bit_flags))
    xcall showit("Voice mail system?",
    &             %bit_is_set(VMAIL_BIT, a_bit_flags))
    return
endsubroutine

subroutine showit
    a_txt      ,a      ;Text to display
    a_set      ,n      ;1 = set, 0 = not set
record
    yn         ,a1      ;Y or N

proc
    if (a_set) then
        yn = 'Y'
    else
        yn = 'N'
    display(TTCHN, a_txt, " (Y/N) ", yn)
    return
endsubroutine
```

%BKSTR – Return the position of the last occurrence of a substring

%BKSTR returns the position of the rightmost occurrence of a substring within a string.

%BKSTR is superseded by %RVSTR. See [%RVSTR on page 9-281](#) for more information.

BREAK – Issue a break to the channel

UNIX

```
xcall BREAK(channel)
```

Arguments

channel

The expression that contains the Synergy Language channel opened to a terminal or character device. **(n)**

Discussion

The BREAK subroutine causes a “break” to be issued on the device specified by *channel*. A break is typically a zero state on the transmission line for 0.233 seconds.

If *channel* is not opened to a terminal or character device, an “Invalid operation for file type” error (\$ERR_FILOPT) is generated.

BTOD – Convert binary values to decimal data

```
xcall BTOD(binary, decimal)
```

Arguments

binary

A field that contains the binary value to be converted. (**a** or **n**)

decimal

The variable that will be loaded with the destination of the converted binary value in decimal form. (**d**)

Discussion

The BTOD subroutine, like its counterpart the DTOB subroutine, supports manipulation of binary bytes (8 bits), words (16 bits), and longwords (32 bits).

BTOD converts a binary value into the appropriate ASCII string in a decimal field, while DTOB does the converse. If the field overflows, no error occurs. If the binary source is larger than four bytes, the rightmost four bytes are treated as a four-byte source. A one-byte source field yields the same result as the DECML subroutine.



With the addition of the integer data type, the BTOD and DTOB routines are no longer required in dealing with native representation of binary data. We recommend you use the integer data type or the %ZONED function for greater efficiency.

See also

[DTOB on page 9-101.](#)

Examples

If you define the following fields:

```
fld1    ,a1
fld2    ,a2
fld4    ,a4
fld5    ,a5
fld6    ,d6
```

the statements below perform these results:

```
xcall dtob(-1, fld5)           ;Yields %XXXXXXXXXXXX in fld5
                                ; (equiv. to ^i(fld5)=-1)
xcall dtob(255, fld4)          ;Yields %X000000FF in fld4
                                ; (equiv. to ^i(fld4)=255)
xcall btod(fld5, fld6)         ;Yields fld6 = -1
                                ; (equiv. to ^d(fld6)=^i(fld5))
xcall btod(fld4, fld2)         ;Yields fld2 = "255"
                                ; (equiv. to ^d(fld2)=^i(fld4))
```

%CHAR – Return an ASCII character for a numeric expression

character = %CHAR (*expression*)

Return value

character

The character that corresponds to the ASCII character value of expression. (**a**)

Arguments

expression

The expression to convert. (**n**)

Discussion

Expression is evaluated modulo 256. (If it is greater than 255, it is divided by 256 and the remainder is used.) For example, %CHAR(299) is equivalent to %CHAR(43).

%CHAR(0) returns a null character.

Examples

The following code loads the delete character as the initial value in **del**.

```
record
    del          , a1

proc
    del = %char(127)
```

CHNOPN – Check whether a channel is in use

```
xcall CHNOPN(channel, status)
```

Arguments

channel

The expression that contains the channel you want to check. (**n**)

status

Returned with one of the following values to tell you if the channel is open or closed: (**n**)

1 The channel is in use.

0 The channel is closed.

Discussion

The CHNOPN subroutine checks whether a channel is in use by an open file. CHNOPN is functionally identical to DBL\$CHOPEN.

Examples

The following program opens a specified file on the first available channel.

```
.define TTCHN          ,1
record
    filename            ,a20          ;Filename to open
    chn                 ,d4           ;Channel to open

proc
    open(TTCHN, o, "tt:")
    xcall flags(4020)
    do forever
        begin
            display(TTCHN, "Enter a filename to open: ")
            reads(TTCHN, filename)
            xcall openit(filename, chn)
            if (chn)
                writes(TTCHN, "Opened " + filename +
                    & "on channel #" + %a(chn))
        end
    end
end
;End of do forever loop
;End of main routine
```



```

subroutine openit                                ;Find an available channel
  a_file          ,a                            ;Contains name of file to open
  a_rchn          ,d                            ;Returned with next available channel

record
  stat            ,d1                          ;Used with XCALL CHNOPN
proc
  for a_rchn from 2 thru 1024
    begin
      xcall chnopn(a_rchn, stat)
      if (.not.stat)
        begin
          open(a_rchn, i, a_file) [err=nofile]
          xreturn
        end
      end
    end

  operr,
    writes(TTCHN, "Exceeds maximum channels " + a_file)
    goto oprtn

  nofile,
    writes(TTCHN, "Cannot open " + a_file)

  oprtn,
    clear a_rchn
    xreturn
endsubroutine

```

%CHOPEN – Determine if an I/O channel is open

status = %CHOPEN(*channel*)

Return value

status

One of the following values: (**n**)

1 The specified channel is currently open.

0 The specified channel is closed.

Arguments

channel

The number of the channel in question. (**n**)

Discussion

%CHOPEN determines whether or not an I/O channel has been opened.

Examples

The following function builds a channel-in-use table for the first 10 channels.

```
function chn_tbl
literal
    bits                ,10i4, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512
record flags
    chn                  ,i4
    chn_tbl              ,i4

proc
    clear chn_tbl
    for chn from 1 thru 10
        if (%chopen(chn))
            chn_tbl = (chn_tbl .bor. bits(chn))
    freturn (chn_tbl)
endfunction
```

CMDLN – Return the command line that started your program

```
xcall CMDLN(command, [num_args], [arg_pos][, arg_len])
```

Arguments

command

The variable that will be loaded with the command line. (**a**)

num_args

(optional) A variable that will be loaded with the number of arguments in *command*. (**n**)

arg_pos

(optional) An array that will be loaded with the beginning position of each argument. (**n**)

arg_len

(optional) An array that will be loaded with the length of each argument. (**n**)

Discussion

The CMDLN subroutine returns the command line that started your Synergy Language program. *Command* is loaded left-justified over blanks.

On Windows and UNIX, *command* includes the system command that starts the runtime (**dbr**). The command includes the full path name of the Synergy runtime. CMDLN does not return the quotation marks in the command line.

WIN

To avoid problems, do not specify a trailing slash character (\) as part of the directory path. For example, use

```
dbr genxml -f test.xml -s "c:\My Documents\smc"
```

rather than

```
dbr genxml -f test.xml -s "c:\My Documents\smc\"
```

The Windows command line parser treats the trailing slash as an escape character, which may cause it to interpret the character that follows the slash incorrectly.

VMS

You need to either set up a symbol as a foreign command to invoke the program or, on OpenVMS 6.1 and above, include the location of the program in the DCL\$PATH search list. For example you must enter

```
prog:==$DIR:prog
```

or

```
define DCL$PATH DBLDIR,UTILS,DIR
```

to make the following command line work with the CMDLN subroutine:

```
prog arg1 arg2
```

If *num_args* is passed, it is returned with the number of arguments contained in *command*. On Windows and UNIX, this includes the path of **dbf** as the first argument and the program filename as the second argument. Arguments are divided according to the rules for each specific platform. For instance, on Windows the first argument is the entire path of **dbf.exe**, even if it includes spaces. On OpenVMS, the arguments are divided at each occurrence of any number of spaces, commas, or tabs that are not enclosed in matching quotation marks.

If *arg_pos* is passed, the beginning position of each argument in *command* is returned in each successive element of *arg_pos*. To retrieve more than the first position, *arg_pos* must be a real array (that is, declared using [] for the dimensions). If the number of elements passed is less than the number of arguments encountered, only the number of elements passed are returned. If more than *num_args* elements are passed, the remaining elements are filled with zeros. Positions are returned base-1 (in other words, 1 indicates the first character of *command*).

If *arg_len* is passed, the length of each argument (not including any delimiters) is returned in each successive element of *arg_len*. To retrieve more than the first length, *arg_len* must be a real array (that is, declared using [] for the dimensions). If the number of elements passed is less than the number of arguments encountered, only the number of elements passed is returned. If more than *num_args* elements are passed, the remaining elements are filled with zeros.

Note that the number of elements in *arg_pos* and *arg_len* do not have to match, and the rules stated above are applied independently to each. Note also that *num_args* is always returned with the true number of arguments, even if neither array contains enough elements to receive the information about all of them.

Examples

The following example checks the number of arguments in the command line that starts your program and displays an error if that number is incorrect. If the number of arguments is correct, the main routine calls the **do_it** routine with each argument that was passed on the command line. In the **process_cmdlin** function, the **argpos** and **arglen** arrays are filled with the starting position and length of each argument by the CMDLN routine. The argument array, **a_argary**, is then filled with each of the arguments that were found on the command line. You might use this routine to identify the start and length of a filename specification on the command line (for example, on Windows, where a filename specification can contain a space), which enables your program to easily recognize the entire specification as a single argument.

```
main
.define MAX_ARGS      3
.align
record
    argcnt              ,i4
    cnt                 ,i4
    args                ,[MAX_ARGS]a255

external function
    process_cmdlin    ,^val

proc
    xcall flags(1001010, 1)
    open(1, o, "TT:")

    argcnt = %process_cmdlin(args, MAX_ARGS)
    if ((argcnt .lt. 1) .or. (argcnt .gt. MAX_ARGS))
        goto usage
    for cnt from 1 thru argcnt
        xcall do_it(args[cnt])

done,
    close 1
    stop

usage,
    writes(1, "Wrong number of arguments")
    writes(1, "Number of arguments must be between 1 and " +
        & %string(MAX_ARGS))
    goto done
endmain
```

```
function process_cmdlin      ,^val
a_argary                    ,[*]a
a_maxargs                   ,n
record
    argcnt                  ,i4
    cnt                     ,i4
    start                   ,i4
    argpos                   ,[10]i4
    arglen                   ,[10]i4
    cmdlin                   ,a255

proc
    xcall cmdln(cmdlin, argcnt, argpos, arglen)
.ifdef OS_VMS
    cnt = 1
.else
.ifdef DEBUG
    cnt = 4
.else
    cnt = 3
.endif
.endif
    start = cnt - 1
    do
        begin
            a_argary[cnt - start] = cmdlin(argpos[cnt]:arglen[cnt])
            incr cnt
        end
    until((cnt .gt. argcnt) .or. ((cnt - start) .gt. a_maxargs))
    freturn (argcnt - start)
endfunction
```

%CNV_IP – Convert native integer data to portable form

portable = %CNV_IP(*native_int*)

Return value

portable

The converted data in portable form. It is the same size as *native_int*. (a)

Arguments

native_int

The native-form integer field to convert. (i)

Discussion

%CNV_IP converts native-form integer data to portable form.

When you use the integer data type, your data files are not portable between big-endian and little-endian machines. You can use the %CNV_IP intrinsic function to convert your integer data to portable form, so it can be interpreted by Synergy Language systems running on different machine architectures. For example, you might want to use %CNV_IP to maintain a file that must be accessed by machines with differing architectures over a network like NFS.

To convert your data back to native form, use %CNV_PI. (See [%CNV_PI on page 9-54](#).) Usually you will call %CNV_IP and %CNV_PI to convert integers directly within a data record before it is written out. By doing this, however, you must keep track of the data form (native or portable). Synergy Language will not know the difference between a native integer and a portable integer.



For integer data in ISAM files on Windows and UNIX, we recommend using the portable integer specification, *l=pos:len* (or PORT_INT *pos:len* in XDL form).

Examples

The following routine writes an integer to a file in portable data format.

```
function wt_portable
  ch      ,n      ;Open channel
  int     ,i      ;Integer to convert and write to file
  size    ,n      ;Size of integer
record
  p_dat    ,a8      ;Portable integer data
proc
  p_dat(1:size) = %cnv_ip(int)      ;Convert integer to portable data
  puts(ch, p_dat(1:size))           ; and write to file
  freturn (0)
endfunction
```

%CNV_PI – Convert portable integer data to native form

native = %CNV_PI (*portable_int*)

Return value

native

The converted data in native form. (i)

Arguments

portable_int

The portable integer field to convert. (a)

Discussion

%CNV_PI converts a portable integer created by a previous %CNV_IP intrinsic function back to native form. (See [%CNV_IP on page 9-53](#).)

When you use the integer data type, your data files are not portable between big-endian and little-endian machines. Once you've used the %CNV_IP intrinsic function to convert your data to portable form and moved it to another machine, you can use %CNV_PI to convert your data back to native form.



For integer data in ISAM files on Windows and UNIX, we recommend using the portable integer specification, *I=pos:len* (or PORT_INT *pos:len* in XDL form).

Examples

The following routine reads a portable integer from a file and returns it as a native integer.

```
function rd_portable
    ch          ,n                ;Open channel
    size        ,n                ;Size of integer
record
    p_dat       ,a8                ;Portable integer data
    int         ,i4
proc
    gets(ch, p_dat(1:size))        ;Read portable data from file
    int = %cnv_pi(p_dat(1:size))    ; convert back to integer
    freturn (int)                  ; and return it
endfunction
```


COMPRESS – Compress a data string

`xcall COMPRESS (cntrl, dest, src, len)`

Arguments

cntrl

The compress control flag. (**n**)

D_COMP	1	Compress
D_UNCOMP	2	Uncompress
DR_COMP	3	Raw compress
DR_UNCOMP	4	Raw uncompress

dest

The data area where the resulting data string is stored. (**a**)

src

The data area containing the data string to compress or uncompress. (**a**)

len

Returned with the resulting length of the operation or -1 on compression failure. (**n**)

Discussion

D_COMP and DR_COMP compress the data record specified by *src* for the length of *src* and store the result in the data record specified by *dest*. The D_COMP form encodes the length of the uncompressed record and stores it at the beginning of the compressed record as a two-byte portable integer. The raw form, DR_COMP, does not store the length. The size of *dest* must be at least three bytes larger than *src* using D_COMP and at least one byte larger using DR_COMP.

D_UNCOMP and DR_UNCOMP uncompress the compressed data record specified by *src*, provided the same form was used to compress the data. If the raw form is used, the length of *dest* is used as the exact length of the uncompressed record.

Dest is not blank-filled if its length is larger than the resulting data string. The size of *dest* must be greater than or equal to the size of the uncompressed record on D_UNCOMP and equal to the size of the uncompressed record on DR_UNCOMP.

The returned *len* is the length of the compressed data record on D_COMP/DR_COMP and the length of the uncompressed data record on D_UNCOMP/DR_UNCOMP.

If the uncompress fails due to invalid compression codes or an invalid *dest* length in the raw form, the returned *len* is set to -1 and *dest* is loaded with the incomplete uncompressed record up to the failure.

Examples

The following routines write and read compressed records to a file.

```
subroutine write_compressed
    a_chn    ,d
    a_rec    ,a

record
    cmprec   ,a MAXRECSZ+3
    clen     ,i2

proc
    xcall compress(D_COMP, cmprec, a_rec, clen)
    puts(a_chn, ^a(clen))          ;See %CNV_IP to make clen portable
    puts(a_chn, cmprec(1:clen))
    xreturn
endsubroutine

subroutine read_compressed
    a_chn    ,d
    a_rec    ,a
    a_len    ,d                    ;Returned record length

record
    cmprec   ,a MAXRECSZ
    clen     ,i2

proc
    gets(a_chn, cmprec(1:clen))
    xcall compress(D_UNCOMP, a_rec, cmprec, a_len)
    if (a_len .eq. -1)
        writes(1, "Compression failure - D_UNCOMP")
    xreturn
endsubroutine
```

COPY – Copy file contents to another file

```
xcall COPY (filename, new_filename[, mode])
```

Arguments

filename

The name of an existing file. **(a)**

new_filename

The name of a new file to be created with the contents of *filename*. **(a)**

mode

(optional) The mode in which to do the copy: **(n)**

0 Binary (default)

1 Text

Discussion

The COPY subroutine copies the contents of a single existing file to a new file.

If *mode* is 0, the source file is assumed to be binary and will be copied exactly as is. This is the default and must be used when copying Synergy ISAM files.

If *mode* is 1, the source file is assumed to be a text file containing text terminated by one or more line terminators. If the file is being copied across the network using *x/Server*, the line terminator(s) are translated to the system where *new_filename* is created. If the file is being copied from a UNIX or Windows system to an OpenVMS system, the file being created is a sequential stream file.

If you specify a new filename that already exists, one of two things occurs, depending on whether the FLAGS subroutine runtime option flag 3 is set:

- ▶ If flag 3 is set, the runtime checks for an existing file with the same name, and a “Cannot supersede existing file” error (\$ERR_REPLAC) is generated.
- ▶ If flag 3 is not set, the runtime does not check for duplicate filenames, and the existing file is overwritten as part of the renaming process. For example, if file **a.b** already exists and the following statement is executed:

```
xcall copy("old.ddf", "a.b")
```

the current **a.b** file is deleted, and the current **old.ddf** is copied to the new **a.b** file.

If the old file is already open by any user in update, output, or append mode, a “File in use by another user” error (\$ERR_FINUSE) is generated. The same error is generated if the new file is already open by any user in any mode.

Filename and *new_filename* can also include a search list logical, which specifies one or more directory paths in which to search for the file. If a search list logical is used on the *new_filename* specification, the copied file is placed in the first directory specified in the search list.

If the COPY subroutine can't find the specified old filename, you'll get a "File not found" error (\$ERR_FNF).

WIN, UNIX

To copy an ISAM file, you must make two calls to the COPY subroutine, specifying the name and extension for each file (*file.ism* and *file.is1*) in addition to mode 0.

If the file is being copied across the network in binary mode to OpenVMS, a sequential stream file is created that contains the binary information from the source file.

VMS

If the COPY subroutine is called locally on OpenVMS, the attributes of the file being copied are preserved. The third argument (*mode*) is ignored.

%CPU TIME – Return the accumulated CPU time

cpu_time = %CPU TIME

Return value

cpu_time

The CPU time accumulated by the current process. The returned value represents the number of 10 millisecond intervals (tics) accumulated. The result is integer data type. (i)

Discussion

%CPU TIME is evaluated at runtime and is an alternative to the CPU TIME subroutine.

WIN

The time is elapsed time, not accumulated system time.

UNIX

The CPU time accumulated includes CPU time for all child processes that the runtime waits for (for example, XCALL SPAWN and XCALL SHELL but not XCALL RUNJB).

The CPU time is not zeroed at the beginning of a Synergy Language program. To measure the CPU time consumed by a program you must sample the CPU time at program startup, and at the checkpoint, sample it again and subtract the first value from the second.

%CPU TIME is functionally equivalent to the CPU TIME subroutine.

Examples

```
record
    last      ,i4
    now       ,i4

proc
    xcall flags(1001010, 1)
    open(1, o, "TT:")
    last = %cputime
    xcall doit
    now = %cputime
    writes(1, "Elapsed time = " + %string(now - last))
    close 1
    stop
end
```

CPUTM – Return the accumulated CPU time

```
xcall CPUTM(time)
```

Arguments

time

Returned with the accumulated CPU time. (**n**)

Discussion

The CPUTM subroutine returns the CPU time accumulated by the current process.

UNIX

The CPU time accumulated includes CPU time for all child processes that the runtime waits for (for example, XCALL SPAWN and XCALL SHELL but not XCALL RUNJB).

CPU time is accumulated for an entire process, not for a program or programs within the process. The returned numeric value represents the number of 10 millisecond intervals (tics) accumulated.

The CPU time is not zeroed at the beginning of a Synergy Language program. To measure the CPU time consumed by a program, you must sample the CPU time at program startup, and at the checkpoint, sample it again and subtract the first value from the second.



We recommend you use %CPUTIME instead of the CPUTM subroutine. (See [%CPUTIME on page 9-59](#).)

Examples

```
record
    last      ,i4
    now       ,i4

proc
    xcall flags(1001010, 1)
    open(1, o, "TT:")
    xcall cputm(last)
    xcall doit
    xcall cptime(now)
    writes(1, "Elapsed time = " + %string(now - last))
    close 1
    stop
end
```

CREMBX – Create a mailbox

VMS

```
xcall CREMBX (channel, [name], [temp_flag], [message_size], [memory], [mask],
&                [access][, wait])
```

Arguments

channel

The expression that contains the Synergy Language channel on which the mailbox is to be opened. **(n)**

name

(optional) An expression that contains the logical name to be assigned to the mailbox. **(a)**

temp_flag

(optional) An expression that indicates whether the mailbox being created is temporary or permanent: **(n)**

0 Temporary mailbox (default)

1 Permanent mailbox

message_size

(optional) An expression that contains the maximum size of messages that can be sent to the mailbox. If this value isn't specified, the system defaults to the DEFMBXMXMSG value in SYSGEN. **(n)**

memory

(optional) An expression that contains the number of bytes of memory used to buffer messages sent to the mailbox. If *memory* isn't specified, the system defaults to the DEFMBXBUFQUO value in SYSGEN. **(n)**

mask

(optional) An expression that specifies the protection mask. The protection mask is a value that consists of four groups of four-bit fields. See the Discussion section below for additional information about protection masks. The default is no protection. **(n)**

access

(optional) An expression that indicates the access mode to be associated with the channel to which the mailbox is assigned: **(n)**

- PSL\$C_KERNEL** Kernel
- PSL\$C_EXEC** Executive
- PSL\$C_SUPER** Supervisor
- PSL\$C_USER** User (default)

wait

(optional) An expression that determines whether the program will wait for another program to write to or read from the mailbox before processing continues. A nonzero value forces the program to wait. **(n)**

Discussion

The CREMBX subroutine enables you to create a mailbox using the OpenVMS \$CREMBX service.

The user must have TMPMBX privileges to create a temporary mailbox, or PRMMBX privileges to create a permanent mailbox. The specified logical name is entered into the group or system logical-name table for a temporary or permanent mailbox. If the logical name is already associated with a mailbox, that mailbox is opened for the user. CREMBX arguments parallel those of the OpenVMS \$CREMBX system service. Refer to the \$CREMBX system service section in your *OpenVMS System Services Reference Manual* for details.

Mask is a decimal value that consists of four groups of four-bit fields. The four groups are World, Group, Owner, and System. The four-bit fields are L, P, W, and R, for logical access, physical access, write access, and read access, respectively. The bits in the *mask* value are assigned as follows:

Groups	WORLD				GROUP				OWNER				SYSTEM			
Fields	L	P	W	R	L	P	W	R	L	P	W	R	L	P	W	R
Bits	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00

Thus, a *mask* value of 10 sets fields L and W in the System group. A value of 12288 sets fields W and R in the World group.

Examples

Let's assume program A opens a mailbox using the following statement:

```
xcall crembx(11, "REDMBX:", 1, , , , 0)
```

This statement tells the system to create a permanent mailbox and associate the logical name **REDMBX** with it. The mailbox is opened for I/O on Synergy Language channel 11. If a mailbox is already associated with the name **REDMBX**, that mailbox is opened. The last argument (**0**) tells the program to continue execution rather than wait for a READS or WRITES on the mailbox. If program B later executes the following line:

```
open(9, i, "REDMBX")
```

the open processor will determine that the filename is a logical name associated with an existing mailbox and open it. This action is equivalent to executing the following statement:

```
xcall crembx(9, "REDMBX", 1, , , , 0)
```

Note that if you want to open the mailbox without waiting for a READS or WRITES, you must either open it with the CREMBX subroutine, or, if you open it with the OPEN statement, specify WAIT:0 on the READS statement.

CVTIIV – Pass an integer value to a field

```
xcall CVTIIV(value, field)
```

Arguments

value

Passed directly or using one of the following functions: (**n**)

^VAL Pass the integer by value.

^REF Pass the integer by reference.

VMS

^XTRNL Pass the integer by external reference.

field

The variable that will be loaded with the binary value. (**n**)

Discussion

The CVTIIV subroutine converts an integer value to a specified field with no data type conversion. The integer must be passed by value, reference, or external reference.

The CVTIIV subroutine supports only three integer sizes:

- 1 Byte size
- 2 Word size
- 4 Longword size

Designating a size other than 1, 2, or 4 generates an “Argument specified with wrong size” error (\$ERR_ARGSIZ).

You can use this routine to store the address of a field in a variable by using the ^REF function as the first argument. You can also use CVTIIV to store an externally defined constant or address by using the ^XTRNL function.



You can use the integer assignment instead of this subroutine. See [“Moving integer data to a decimal/packed destination” on page 3-37](#) for more information about assigning integer values.

Examples

```
xcall cvtiiv(^ref(variable), byte_val)
```

CVTIZV – Pass an integer value to a zoned numeric value

```
xcall CVTIZV(value, field)
```

Arguments

value

Passed directly or using one of the following functions: (**n**)

^VAL Pass the integer by value.

^REF Pass the integer by reference.

VMS

^XTRNL Pass the integer by external reference.

field

The variable that will be loaded with the converted integer. (**n**)

Discussion

The CVTIZV subroutine converts an integer value to a zoned numeric value. The integer must be passed by value, reference, or external reference.

A field can be converted either into itself or into another named field.



You can use the integer assignment instead of this subroutine. See [“Moving integer data to a decimal/packed destination” on page 3-37](#) for more information about assigning integer values.

Examples

The following example converts the externally defined constant `SS$_ENDOFFILE` and places the result in the variable `sseof`. `SS$_ENDOFFILE` is a system service condition value.

```
xcall cvtizv(^xtrnl(SS$_ENDOFFILE), sseof)
```

^D – Access data as decimal or implied-decimal

`^D (expression[, precision])`

Arguments

expression

An expression to be accessed as decimal or implied-decimal. (**a** or **n**)

precision

(optional) The fractional precision. (**n**)

Discussion

`^D` accesses data as a decimal or implied-decimal data type.

`^D` does not convert *expression* to a decimal or implied-decimal value; the data itself remains unchanged. Only the data type is changed for the reference. (If you need to convert an expression to an implied-decimal value, use the `%IMPLIED` function as described on [page 9-159](#). There is no conversion function that creates a decimal from another type.)



Before using `^D(expression)` in another expression, be sure to validate that the data is numeric. Using `^D` on non-numeric data yields undefined numeric results.

If you specify *precision*, the expression is accessed as implied-decimal with the specified fractional precision. If you omit *precision*, the expression is accessed as decimal. *Precision* cannot be less than zero, greater than 28, or greater than the size of *expression*.

Examples

```

record drec
    len      ,d4
    siz      ,d4
    val      ,d4

record
    amount   ,d10.4

proc
    clear ^d(drec)                ;Clear all decimals with zeros
    amount = 234.5678

    ; ^d(amount, 2) changes amount from a d10.4 to a d10.2
    ; This causes the contents of amount to be viewed as 123456.78

    ^d(amount, 2) = %round(^d(amount, 2))    ;Round off last two digits
                                              ;Amount will contain 234.57

    xreturn
end

```

DACHR – Disable activation characters

```
xcall DACHR(character[, ...])
```

Arguments

character

One or more decimal codes, each representing an ASCII character. Valid codes are 0 through 255. (**n**)

Discussion

The DACHR subroutine deactivates activation characters (as established by the ACCHR subroutine) so they will not be processed during terminal input.

If you try to disable a character that is not currently in the list of activation characters, your attempt will be ignored. The default activation characters (line feed, form feed, TAB, and ESC) cannot be deactivated.

See also

[ACCHR on page 9-12](#) for more information about activation characters and sample code that contains the DACHR subroutine.

DAESC – Disable activation characters

The DAESC subroutine is functionally identical to the DACHR subroutine. It is included for compatibility with earlier versions of Synergy Language. See [DACHR on page 9-67](#) for a thorough discussion.

DATA_DECRYPT – Decrypt ciphertext data

```
xcall DATA_DECRYPT(type, password, source, [destination], [len], [salt, iv][, pad])
```

Arguments

type

One of the following encryption types: **(n)**

DC_3DES Triple DES CBC three-key mode

DC_AES128 AES 128-bit CBC mode

DC_AES192 AES 192-bit CBC mode

DC_AES256 AES 256-bit CBC mode

password

The password to be used on the source data buffer. **(a)**

source

The cipher text data to be decrypted.

destination

(optional) A variable to be loaded with plaintext data as a result of this operation.

len

(optional) A variable to be set to the length of the data written to *destination* (or to the required destination size if *destination* is not specified). **(n)**

salt

(optional) A salt value for the decryption algorithm. **(a)**

iv

(optional) An initialization vector value for the decryption algorithm. **(a)**

pad

(optional) When set to 0, turns off PKCS padding. (See [“Data encryption” on page 1-35](#) for more information about padding.) **(n)**

Discussion

The DATA_DECRYPT routine decrypts data passed in *source* and returns it in *destination*.

The *source* specification must be exactly the data that was encrypted. If the data was encrypted into a field larger than the actual encrypted ciphertext data, the *source* specification must be ranged.

If *len* is specified and *destination* is not, the exact size of the decrypted plain text data will be returned.

If *salt* and *iv* are not specified, the values supplied to the decryption routine are nulls. The size of *salt* and *iv* are the size as returned from the DATA_SALTIV routine, or else an \$ERR_ARGSIZ error occurs.

The size of the *salt* and *iv* arguments are as follows:

Encryption type	Salt size	Initialization vector size
DC_3DES	8	8
DC_AESnnn	8	16

When using the dbl8 compiler, you must `.INCLUDE` the file **DBLDIR:atacrypt.def** before using DATA_DECRYPT.

See also

- ▶ [“Data encryption” on page 1-35.](#)
- ▶ [DATA_ENCRYPT on page 9-71.](#)
- ▶ [DATA_SALTIV on page 9-73.](#)

DATA_ENCRYPT – Encrypt plaintext data

```
xcall DATA_ENCRYPT(type, password, source, [destination], [len], [salt, iv][, pad])
```

Arguments

type

One of the following encryption types: **(n)**

DC_3DES Triple DES CBC three-key mode

DC_AES128 AES 128-bit CBC mode

DC_AES192 AES 192-bit CBC mode

DC_AES256 AES 256-bit CBC mode

password

The password to be used on the source data buffer. **(a)**

source

The plain text data to be encrypted.

destination

(optional) A variable to be loaded with ciphertext data as a result of this operation.

len

(optional) A variable to be set to the length of the data written to *destination*. **(n)**

salt

(optional) A salt value for the encryption algorithm. **(a)**

iv

(optional) An initialization vector value for the encryption algorithm. **(a)**

pad

(optional) When set to 0, turns off PKCS padding. (See “Data encryption” on page 1-35 for more information about padding.) **(n)**

Discussion

The DATA_ENCRYPT routine encrypts data passed in *source* and returns it in *destination*.

The following rules apply to *len* and *destination*:

- ▶ If *len* is not specified, the *destination* specification must be the exact size of the encrypted ciphertext data.
- ▶ If *len* is specified, the size of *destination* must be equal to or greater than the size of the encrypted ciphertext data.
- ▶ If *len* is specified and *destination* is not, the exact size of the ciphertext data will be returned; no encryption is performed.

If *salt* and *iv* are not specified, the values supplied to the encryption routine are nulls.

When using the dbl8 compiler, you must `.INCLUDE` the file **DBLDIR:atacrypt.def** before using DATA_ENCRYPT.

See also

- ▶ [“Data encryption” on page 1-35.](#)
- ▶ [DATA_DECRYPT on page 9-69.](#)
- ▶ [DATA_SALTIV on page 9-73.](#)

DATA_SALTIV – Generate a salt and/or initialization vector

```
xcall DATA_SALTIV (type, [salt], saltlen, [iv], ivlen)
```

Arguments

type

One of the following encryption types: **(n)**

DC_3DES Triple DES CBC three-key mode

DC_AES128 AES 128-bit CBC mode

DC_AES192 AES 192-bit CBC mode

DC_AES256 AES 256-bit CBC mode

salt

(optional) A variable to be loaded with a salt value for key generation. **(a)**

saltlen

A variable to be set to the length of the salt value for the specified encryption type. **(n)**

iv

(optional) A variable to be loaded with an initialization vector (IV) for the specified encryption type. **(a)**

ivlen

A variable to be set to the length of the initialization vector for the specified encryption type. **(n)**

Discussion

DATA_SALTIV generates a random salt and/or an initialization vector for encrypting and decrypting data. You can then pass this information, along with the user password, as arguments to the DATA_ENCRYPT routine to maximize the effectiveness of the cipher you are using. Because neither the salt nor the initialization vector is considered to be sensitive information, you can write these values to a file and use them again to decrypt the data.

When you specify *type* and only *saltlen* and/or *ivlen* (i.e., not *salt* and/or *iv*), the appropriate length is returned but no salt or initialization vector value is generated.

When using the dbl8 compiler, you must `.INCLUDE` the file **DBLDIR:atacrypt.def** before using DATA_SALTIV.

See also

- ▶ [“Data encryption” on page 1-35.](#)
- ▶ [DATA_DECRYPT on page 9-69.](#)
- ▶ [DATA_ENCRYPT on page 9-71.](#)

^DATATYPE – Return the data type

`^DATATYPE(position)`

Arguments

position

The ordinal position of the parameter whose data type will be returned. (**n**)

Discussion

^DATATYPE returns the data type of a variable or parameter as one of the following values, which are defined by the compiler:

D_TYPE_A	1	Alpha
D_TYPE_D	2	Decimal
D_TYPE_ID	4	Implied-decimal
D_TYPE_I	8	Integer
D_TYPE_P	32	Packed
D_TYPE_IP	36	Implied-packed
D_TYPE_OBJ	24	Object handle

Use the .IS. operator to determine a specific type of class. (See [“Object operators” on page 3-14](#) for more information about .IS..)

Examples

```
subroutine stringval
  a_results      ,a
  ;Implied second parameter

proc
  if (^passed(^arg(2))) then
  begin
    using (^datatype(2)) select
      (D_TYPE_A),
      a_result = "String is " + ^arg(2)
      (D_TYPE_D, D_TYPE_I),
      a_result = "Value is " + %string(^argn(2))
      (),
      a_result = "Value is invalid"
    endusing
  end
```

System-Supplied Subroutines, Functions, and Classes

^DATATYPE

```
    else
        a_result = "String or value not passed"
    xreturn
endsubroutine
```

%DATE – Return the date

date = %DATE([*jperiod*])

or

xcall DATE(*date*[, *jperiod*])

Return value

date

A formatted date string, stored left-justified over blanks. (**a**)

Arguments

jperiod

(optional) The Julian period for the date to be returned. The default is the current date. (**n**)

Discussion

Depending on whether the destination is **a9** or **a11**, the DATE subroutine returns either the current date or the date for the Julian period passed in the format DD-MMM-YY or DD-MMM-CCYY, where DD is the day of the month, MMM is the abbreviation for the month, YY is the last two digits of the year (for example, 11-NOV-91), and CCYY is the year with the century included.

The %DATE function, on the other hand, always returns the date in the format DD-MMM-CCYY. If the length of *date* is not large enough to receive the entire result, the date is truncated, with the truncated value stored according to the alpha to alpha storage rules. (See [“Moving alpha data to an alpha destination” on page 3-26](#) for more information.) For example, if *date* is an **a9** field, the returned date is truncated to DD-MMM-CC.

The abbreviations for the months are as follows:

JAN
FEB
MAR
APR
MAY
JUN
JUL
AUG
SEP
OCT
NOV
DEC

Examples

The following example checks to see if the current date is the same as the date passed to the function. If it is the same, the function returns a value of true; otherwise, it returns a value of false.

```
function chk_exp_date                                ;Called daily to generate
    a_expdate                                     ; expired account list

proc
    freturn (%date .eq. a_expdate)                ;Return TRUE if expired today,
                                                ; else FALSE
endfunction
```


%DATECOMPILED – Return the compilation date and time

date = %DATECOMPILED

Return value

date

The compilation date and time. (**a**)

Discussion

%DATECOMPILED is evaluated at compile time.

The date and time are returned in the format DD-MMM-YYYY hh:mm:ss.

DD	The day (01 through 31)
MMM	The month (Jan through Dec)
YYYY	The four-digit year (for example, 1995)
hh	The hour (00 through 23)
mm	The minutes (00 through 59)
ss	The seconds (00 through 59)

%DATETIME – Return the date and time

date = %DATETIME[(*utc_offset*)]

Return value

date

The current date and time. (**a20**)

Arguments

utc_offset

(optional) The optional offset in minutes east of the Greenwich Meridian. (**n**)

Discussion

The date and time are returned in the format YYYYMMDDhhmmssuuuuuu:

YYYY The four-digit year (for example, 1994)

MM The month (01 through 12)

DD The day of the month (01 through 31)

hh The hour (00 through 23)

mm The minute of the hour (00 through 59)

ss The second of the minute (00 through 59)

uuuuuu The microseconds (000000 through 999999)

If your operating system doesn't provide all six microsecond digits, any unknown digits are returned as zeros.

If *utc_offset* is specified, %DATETIME returns the Coordinated Universal Time (UTC) date and time. If *utc_offset* is 0 or not specified, the date and time are the local date and time.

See also

[%TIMEZONE](#) on page 9-344.

Examples

```

subroutine sh_date_time

record
  group datetime  ,a20
  group date      ,a8
    year          ,a4
    month         ,a2
    day           ,a2
  endgroup
  group time      ,a12
    hour          ,a2
    minute        ,a2
    second        ,a2
    msecond       ,a6
  endgroup
endgroup
string           ,a60
slen             ,d3

proc
  datetime = %datetime
  xcall s_bld(string, slen,
    &          "Date and Time: %A = %A/%A/%A %A:%A:%A",
    &          datetime, day, month, year, hour,
    &          minute, second)
  writes(TTCHN, string)
  return
endsubroutine

```

In the example below, passing %TIMEZONE as the optional argument to %DATETIME causes %DATETIME to return the UTC date and time string.

```

utc_date = %datetime(%timezone)

```

DBL\$ subroutines

The majority of the DBL\$ subroutines are syntactically and functionally identical to subroutines with the same name but without the “DBL\$” prefix, with a few exceptions noted below. Refer to the specified pages for the correct syntax for each.

DBL\$ routine	Description	Identical routine	Page
DBL\$ASCII	Convert a decimal value into an ASCII character	ASCII	9-26
DBL\$ASTRST	Restore the contents of work areas used as the result of an AST	ASTRST	9-29
DBL\$ASTSAV	Save the contents of work areas used as the result of an AST	ASTSAV	9-30
DBL\$BEGFL	Position to first record	BEGFL	9-34
DBL\$BKSTR	Reverse search a string for a target value	BKSTR RVSTR	9-281
DBL\$CHOPEN	Check whether a channel is in use	CHOPN	9-46
DBL\$CPUTM	Return accumulated CPU time	CPUTM	9-60
DBL\$CREMBX	Create a mailbox	CREMBX	9-61
DBL\$CVTIIV	Pass an integer value to a field	CVTIIV	9-64
DBL\$CVTINT	Convert decimal data to binary values	DTOB	9-101
DBL\$CVTIZ	Convert binary values to decimal data	BTOD	9-43
DBL\$CVTIZV	Pass an integer value to a zoned numeric value	CVTIZV	9-65
DBL\$CVTZI	Convert decimal data to binary values	DTOB	9-101
DBL\$DATE	Return a date string	DATE	9-77
DBL\$DECML	Convert an ASCII character to its decimal value	DECML	9-92
DBL\$DELET	Delete the specified file	DELET	9-95
DBL\$DELMBX	Mark a permanent mailbox for deletion	DELMBX	9-96

DBL\$ routine	Description	Identical routine	Page
DBL\$DEVCLT	Return the class and type of a device	none	9-86
DBL\$ENDFL	Position the file pointer after the last record of a file	ENDFL	9-104
DBL\$ERRMOD	Return the module in which the most recent error occurred	ERRMOD	9-110
DBL\$ERROR	Return an error code number	ERROR	9-111
DBL\$ERRTXT	Return an error message	ERTXT	9-113
DBL\$EXECUTE_IMAGE_ROUTINE	Execute a routine in a shared image	none	9-88
DBL\$EXITERROR	Exit the routine with a specific error	EXITE	9-116
DBL\$FATAL	Chain to another program if a fatal error occurs	FATAL	9-121
DBL\$FLAGS	Set runtime option flags	FLAGS	9-126
DBL\$GETCM	Load the contents of a common area into a variable	GETCM	9-139
DBL\$GETDFN	Return the default file specification	GETDFN	9-140
DBL\$GETFA	Return file attributes	GETFA	9-141
DBL\$GETFNM	Return the file specification of the open channel	GETFNM	9-147
DBL\$GLINE	Get a line of input	GLINE	9-153
DBL\$INSTR	Search a string for a target value	INSTR	9-165
DBL\$INTCVT	Convert binary values to decimal data	BTOD	9-43
DBL\$JBNO	Provide the job number for the current process	JBNO	9-196
DBL\$KILL	Terminate the calling program	KILL	9-203
DBL\$MONEY	Specify the currency symbol	MONEY	9-220
DBL\$PAK	Compress numeric data	PAK	9-236

System-Supplied Subroutines, Functions, and Classes

DBL\$ subroutines

DBL\$ routine	Description	Identical routine	Page
DBL\$PARSE	Extract components of a file specification	PARSE	9-239
DBL\$PURGE	Delete previous versions of a file	PURGE	9-247
DBL\$PUTCM	Put the contents of a variable into a common area	PUTCM	9-248
DBL\$RENAM	Rename a file	RENAM	9-264
DBL\$RSTAT	Return information about the last record read	RSTAT	9-272
DBL\$RSTATD	Return numeric information about the last record read	RSTATD	9-274
DBL\$SETCTL	Modify the operation of control characters	SETCTL	9-301
DBL\$SETDFN	Set the default file specification	SETDFN	9-303
DBL\$SETKRF	Set the key of reference for the next operation of an ISAM file	none	9-90
DBL\$SIZE	Return the length of the data area	SIZE	9-310
DBL\$SNDOPR	Send a message to the system operator	none	9-91
DBL\$TIME	Return the current system time	TIME	9-343
DBL\$TNMBR	Return the terminal number	TNMBR	9-346
DBL\$TRACEBACK	Print the current traceback	TRACEBACK	9-349
DBL\$TT_NAME_TO_NUMBER	Convert a terminal name to its number	TT_NAME_TO_NUMBER	9-357
DBL\$TT_NUMBER_TO_NAME	Convert a terminal number to its name	TT_NUMBER_TO_NAME	9-358
DBL\$TTBRDCST	Enable a program to trap broadcast messages	TTBRDCST	9-359
DBL\$TTCHAR	Return type, lines, and width of a file	TTCHAR	9-361
DBL\$TTFLGS	Set terminal-oriented flags	TTFLGS	9-364

DBL\$ routine	Description	Identical routine	Page
DBL\$TTNAME	Return name of the terminal running the program	TTNAME	9-371
DBL\$TTSTS	Determine whether input is pending	TTSTS	9-372
DBL\$UNPAK	Decompress numeric data	UNPAK	9-374
DBL\$VERSN	Identify the current operating system	VERSN	9-379
DBL\$VMCMD	Execute a DCL command	VMCMD	9-380
DBL\$VMMSG	Get the text of an OpenVMS system message	VMMSG	9-382
DBL\$WKDAY	Return the day of the week	WKDAY	9-387
DBL\$XARGS	Return information about arguments passed to external routine	XARGS	9-393
DBL\$XSTAT	Get return value from last routine called	XSTAT	9-394

DBL\$DEVCLT – Return the class and type of a device

VMS

```
xcall DBL$DEVCLT ([channel], [device], [class], [type], [sec_class][, sec_type])
```

Arguments

channel

(optional) An expression that evaluates to the channel on which the device (or a file that resides on that device) was opened in a previous OPEN statement. **(n)**

device

(optional) A field or literal that contains the device specification (used if the device is not currently open). **(a)**

class

(optional) Returned with a string that identifies the device class. The returned strings are shown in parentheses in the Device class column of the table in the Discussion section below. *Class* must be at least four bytes. **(a)**

type

(optional) Returned with a string that identifies the device type. The returned strings are shown in the Device types returned column of the table in the Discussion section below. *Type* must be at least eight bytes. **(a)**

sec_class

(optional) Returned with a string that identifies the secondary device class. The returned strings are shown in parentheses in the Device class column of the table in the Discussion section below. *Sec_class* must be at least four bytes. **(a)**

sec_type

(optional) Returned with a string that identifies the secondary device type. The returned strings are shown in the Device types returned column of the table in the Discussion section below. *Sec_type* must be at least eight bytes. **(a)**

Discussion

The DBL\$DEVCLT subroutine returns the class and type of a device.

Channel should not be specified in the same calling sequence as *device* and vice versa. If the device is open, use a call of this format:

```
xcall dbl$devclt(channel,, class, type)
```

If the device is not open, use a call of this format:

```
xcall dbl$devclt(, device, class, type)
```

If both *channel* and *device* are specified, *channel* is used and *device* is ignored.

The following device classes and types are returned:

Device class (value returned)	Device types returned
Disk (DISK)	The particular type (such as RA80, RX50) is returned.
Magtype (TAPE)	The particular type (such as TU78, TK50) is returned.
Terminal (TERM)	The particular type (VT100, LA36) is returned. An unknown type is returned as TTYUNKNWN; any VT200 series is returned as VT200SRS; any PRO series is returned as PRO.
Line printer (LP)	The particular type (such as LP11, LA11) is returned.
Card reader (CARD)	The particular type (such as CR11) is returned.
Mailbox (MLBX)	The particular type (such as MBX, NULL) is returned.
Workstation (WKST)	The particular type (such as VS100, VS125, VS300) is returned as VS300.

The strings moved to *class*, *type*, *sec_class*, and *sec_type* are moved according to the rules for moving alpha data. (See [“Moving alpha data to an alpha destination”](#) on page 3-26 for more information on assigning alpha values.)

The arguments *sec_class* and *sec_type* are the same as *class* and *type*, except in cases where a secondary device is associated with the primary device. Two examples of this condition follow:

- ▶ For a spooled line printer, the primary class and type reflect the intermediate spooling device, usually a disk, and the secondary class and type reflect the line printer.
- ▶ For a terminal that has an associated mailbox, the primary class and type reflect the terminal, and the secondary class and type reflect the mailbox.

%DBL\$EXECUTE_IMAGE_ROUTINE – Execute a routine in a shared image

VMS

value = %DBL\$EXECUTE_IMAGE_ROUTINE (*file*, *routine*, ^VAL (*nargs*) [, *arguments*, ...])

or

xcall DBL\$EXECUTE_IMAGE_ROUTINE (*file*, *routine*, ^VAL (*nargs*) [, *arguments*, ...])

Return value

value

The return value for *routine*. (**n**)

Arguments

file

The name of the shared image file. The default file specification **SYSS\$SHARE:.EXE** is used. (**a**)

routine

The name of the routine to be executed. (**a**)

nargs

The number of arguments to be passed to *routine*. If there are no arguments, *nargs* must be 0. (**n**)

arguments

(optional) A list of arguments to be passed to *routine*.

Discussion

The DBL\$EXECUTE_IMAGE_ROUTINE subroutine executes a routine contained in a shared image. The shared image does not have to be linked with the user program.

Only the filename component of *file* is significant. The device, directory, and file type field are ignored. By default, OpenVMS attempts to locate the file in SYSS\$SHARE: unless the name is a logical. The only way to use a shared image file that is not in SYSS\$SHARE is by assigning a logical to refer to it, and passing that logical name as *file*.

DBL\$EXECUTE_IMAGE_ROUTINE uses LIB\$FIND_IMAGE_SYMBOL to find the address of the routine specified by *routine*. All restrictions and rules that apply to LIB\$FIND_IMAGE_SYMBOL also apply to DBL\$EXECUTE_IMAGE_ROUTINE.

Arguments must follow the same rules as if *routine* was being called using the XCALL statement.
LIB\$FIND_IMAGE_SYMBOL signals all errors if unsuccessful.

See also

[^VAL on page 9-376.](#)

Examples

If subroutine SUB is in the shared image MY_SHR_LIB and is normally called as

```
xcall sub(a, b, c)
```

then using DBL\$EXECUTE_IMAGE_ROUTINE, you would call it as follows:

```
xcall dbl$execute_image_routine("MY_SHR_LIB", "SUB", ^val(3), a, b, c)
```

DBL\$SETKRF – Set the key of reference for the next operation on an ISAM file

VMS

`xcall DBL$SETKRF (channel, key_number)`

Arguments

channel

The channel on which the files are open. (**n**)

key_number

The key of reference for the next operation: (**n**)

0 Primary key

1 – 7 Alternate keys

Discussion

The DBL\$SETKRF subroutine sets the key of reference that the next operation will use. It resets the current position in the file to the first record.



We recommend you use the KEYNUM qualifier on I/O statements instead of the DBL\$SETKRF subroutine. (See [KEYNUM](#) on page 4-222.)



You can call this routine as SETKRF as long as you don't compile with the **/NOGBLDEFS** option.

DBL\$SENDOPR – Send a message to the system operator

VMS

xcall DBL\$SENDOPR (*message*)

Arguments

message

A field, literal, or record that contains the message. The maximum message size is 120 characters. **(a)**

Discussion

The DBL\$SENDOPR subroutine sends a message to the system operator.

Message is sent using the \$SENDOPR system service with the OPC\$_RQ_RQST message type. It is targeted for the central operator.



You can call this routine as SENDOPR as long as you don't compile with the **/NOGBLDEFS** option.

%DECML – Convert an ASCII character to its decimal value

```
result = %DECML(expression)
```

or

```
xcall DECML(expression, result)
```

Return value

result

The decimal equivalent of the converted character. If *expression* is null, *result* will be zero. (n)

Arguments

expression

The alpha expression whose leftmost character is to be converted to a decimal ASCII value. (a)

Discussion

The DECML subroutine converts a single ASCII character into its corresponding decimal value and returns or stores the value in the specified *result* variable. (See [Appendix B](#) for a complete set of ASCII characters and their decimal equivalents.)

The maximum decimal value is 255, so the size of *result* must be large enough or truncation will result.



If sign is not important (characters greater than 127 are not used), it's more efficient to cast a single-character variable as an integer or overlay the first character with an **i1** to access its ASCII value. For example:

```
record
    achar          ,a1
    asc_value      ,i1 @achar
```

Examples

```
record
    achar          ,a3
    dchar          ,d3

proc
    xcall decml("A1$", dchar)                ;dchar = 065 ("A")
    achar = "t-R"
    xcall decml(achar, dchar)                 ;dchar = 116 ("t")
    xcall decml(achar(2,2), dchar)            ;dchar = 045 ("-")
    stop
end
```

The following subroutine builds a script filename from a partial filename and uppercases it if it is lowercase.

```

subroutine bldnam
    a_name      ,a
record
    ix          ,d3

proc
    if (.not. (%instr(2, a_name, ".")))          ;If no extension, add one
    begin
        a_name((%trim(a_name) + 1):4) = ".wsc"
        if (%instr(1, a_name, ":", ix)) then    ;If filename is uppercase,
            ix = %decml(a_name(ix+1:1))        ; uppercase extension too.
        else
            ix = %decml(a_name)
        case ix of
            begincase
                65 - 90: upcase a_name
            endcase
        clear ix
    end
    return
endsubroutine

```

^DEFINED – Determine if a field or parameter has been defined

`^DEFINED ( variable_spec )`

Return value

`^DEFINED` returns a value of 1 if the entity has been defined, or a value of 0 if the entity has not been defined. **(i)**

Arguments

variable_spec

A field or path specification. **(a or n)**

Discussion

`^DEFINED` is evaluated at compile time. It determines if a field or parameter has been defined.

You don't need to specify a full path specification for *variable_spec* as long as you specify enough information to determine that the field exists.



Macros are expanded when `^DEFINED` is processed. Therefore, it is not possible to use `^DEFINED` to determine whether a macro variable is defined. Instead, use `.IFDEF` and `.IFNDEF` for this purpose.

Examples

This example determines whether code is compiled based on whether or not **rec.dvar** has been defined.

```
.if .not. ^defined(rec.dvar)
.           ;This code will be compiled if rec.dvar
.           ; is not defined
.
.endc
```


DELET – Delete the specified file

```
xcall DELET(filename)
```

Arguments

filename

An expression that contains a file specification. See [“File specifications” on page 1-43](#) for details on the format of a file specification. (a)

Discussion

The DELET subroutine locates and deletes the specified file. If the specified file does not exist, no error is generated. However, if the filename is null or blank, a “Bad filename” error (\$ERR_FILSPC) is generated.

To delete an ISAM file, you must include the filename extension. Both the index and the related data file are deleted.

In previous versions of Synergy Language, you could also specify a channel argument. This syntax is still valid, but the channel is ignored.

VMS

Filename can contain wildcard characters. The default filename extension is **.ddf**. In addition, a given file can have more than one version. If *filename* does not explicitly specify the version number, all versions of that file are deleted.

Before deleting a file, make sure that you first close any channels that your program has open to that file.

If the file is open by your program or anyone else on the system when the program calls DELET, a “File in use by another user” error (\$ERR_FINUSE) is generated.

Examples

```
literal
    file ,a20,"DP3:test01.xyz"

proc
    xcall delet(file)
    xcall delet("DM7:demox.tv")
    stop
end
```

DELMBX – Mark a permanent mailbox for deletion

VMS

```
xcall DELMBX(channel)
```

Arguments

channel

The Synergy Language channel on which the mailbox was opened, using either CREMBX or an OPEN statement. (**n**)

Discussion

The DELMBX subroutine enables the process running the program to mark a permanent mailbox for deletion. The actual deletion occurs when no more channels are assigned to the mailbox.

The DELMBX subroutine marks the mailbox to be deleted when it is no longer used by any process. (You can also specify a temporary mailbox to be deleted, but temporary mailboxes are automatically deleted when they are no longer in use.) The mailbox continues to exist as long as at least one process has a channel assigned to it.

If you specify a channel that is not currently opened, a “Channel has not been opened” error (\$ERR_NOOPEN) is generated.

See also

- ▶ [CREMBX on page 9-61.](#)
- ▶ [OPEN on page 4-124.](#)

Examples

```
subroutine cleanup
proc
    xcall close_chns
    xcall delmbx(MBXCHN)
    return
stop
```

^DESCR – Pass an argument by descriptor

`^DESCR (expression)`

Arguments

expression

An argument to pass by descriptor. (**a** or **n**)

Discussion

^DESCR passes an argument by descriptor.

The result of *expression* is passed by descriptor. A descriptor is a memory block that contains information about the associated variable or literal.

Because Synergy Language passes arguments by descriptor by default, ^DESCR is only available for consistency with the other methods of passing arguments (^VAL, ^REF, and ^XTRNL).

DFLAG – Set runtime option flags

```
xcall DFLAG (flags [, control])
```

Arguments

flags

The runtime option flags. Each flag is a single digit, and the digit positions are assigned ordinal numbers from right to left. Thus, the low-order digit is digit number 1. Synergy Language examines the flags value digit by digit to determine which flags are set, and hence, which functions are to be performed. The Discussion section below contains a detailed description of the flags and their meanings. (**n**)

control

(optional) A value that controls how the individual digit positions in the flag specification value are interpreted. If *control* is not specified, each digit in *flags* is interpreted as follows: (**n**)

- 0** Reset the associated flag.
- n** Set the associated flag, where *n* is any nonzero decimal number.

We recommend that you use the *control* value when setting and resetting your *flags* values. If *control* is present, its value is interpreted as follows:

- 0** Reset all flags whose corresponding digit position contains a nonzero value. Leave all other flags unmodified.
- 1** Set all flags whose corresponding digit position contains a nonzero value. Leave all other flags unmodified.
- 2** Set the *flags* variable to a value that reflects the current settings of the flags and leave all flags unmodified. Each digit of the *flags* variable is set to zero if the associated flag is currently reset or to one if the flag is set. If the length of the variable is less than 10, the settings are returned as in a decimal-to-decimal assignment. (See [“Moving decimal/packed data to a decimal/packed destination” on page 3-33](#) for more information about decimal-to-decimal assignments.)

Discussion

The DFLAG subroutine, like the FLAGS subroutine, controls the settings of various runtime option flags. These flags control the way the runtime acts in response to the execution of Synergy Language statements.

When program execution begins, all program-controlled flags are normally reset, and the program can alter them as needed. The flag settings are not preserved across program chaining. In effect, the STOP statement resets all of the flags as its last action, even if another program is being started. The flag settings in one program do not affect the settings in a program that is running concurrently from another terminal.

Remember that the rightmost digit in the flag specification string corresponds to flag number 1, as shown in [figure 9-1](#). Because Synergy Language interprets the *flags* argument as if it were storing it to a **d10** variable, you don't have to specify a flag if that flag and all flags to its left have a value of zero. All rules that apply when storing to a decimal variable apply here. For example, if you specify

```
xcall dflag(00100)
```

Synergy Language interprets your argument as 0000000100. (All flags corresponding to these zeros are cleared.)

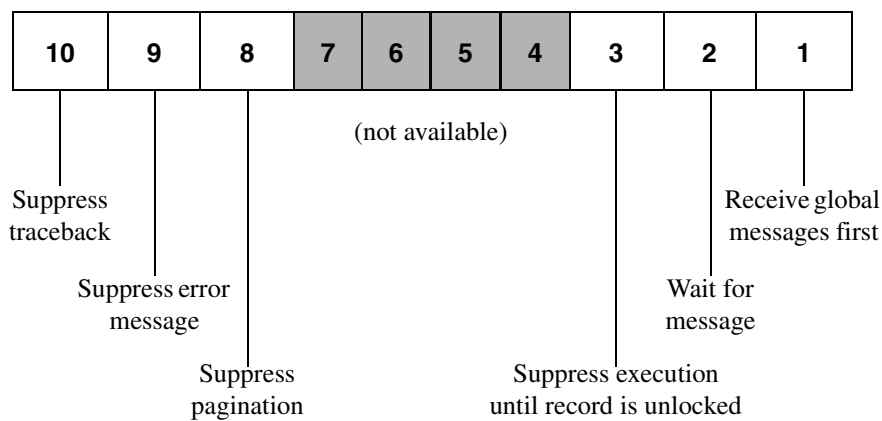


Figure 9-1. DFLAG runtime option flags.

A more detailed description of the flags is provided below.

Flag	Description	Windows	UNIX	OpenVMS
1	When flag 1 is set, the program receives any available global messages before any local messages. When flag 1 is reset, the program receives local messages before global messages.		X	X
2	Flag 2 causes the program to wait for a message if no messages exist. When flag 2 is reset, program control transfers to the “no message pending” label specified in the RECV statement.		X	X

System-Supplied Subroutines, Functions, and Classes

DFLAG

Flag	Description	Windows	UNIX	OpenVMS
3	When an I/O statement tries to access a locked record and flag 3 is set, execution is suspended until the record is unlocked. At that time, the I/O statement is completed. When flag 3 is reset, an attempt to access a locked record generates a “Record is locked” error (\$ERR_LOCKED). This error is trapped if an ONERROR statement or an I/O error list that contains a LOCK entry is included in the I/O statement that attempted access. (This flag is overridden by the the WAIT qualifier.)	X	X	X
4 – 7	No longer available.			
8	When flag 8 is set, SJC\$_NOPAGINATE is set in LPQUE statements. When flag 8 is reset, SJC\$_NOPAGINATE is turned off.			X
9	When flag 9 is set, the “Unexpected VMS system error” message (VMSError) is suppressed on any signaled error condition. When flag 9 is reset, the message is issued on any signaled error condition. This flag does not affect trapped OpenVMS error conditions, only unexpected signaled conditions such as those created by LIB\$SIGNAL and LIB\$STOP.			X
10	When flag 10 is set, Synergy Language traceback is suppressed on any signaled error condition. When flag 10 is reset, traceback is not suppressed. This flag does not affect trapped OpenVMS error conditions, only unexpected signaled conditions such as those created by LIB\$SIGNAL and LIB\$STOP.			X



To make your code easier to maintain, use the flag number as the nonzero digit that sets the flag. For example, a *flags* value of “301” tells you at a glance that flags 3 and 1 are set.

See also

[FLAGS on page 9-126.](#)

Examples

The example below sets DFLAG flags 1 and 3 and leaves the others unmodified.

```
xcall dflag(301, 1)
```

DTOB – Convert decimal data to binary values

```
xcall DTOB(decimal, binary)
```

Arguments

decimal

A field that contains the decimal value to be converted. (**n**)

binary

The variable that will be loaded with the converted decimal value in binary form. (**a** or **n**)

Discussion

The DTOB subroutine, like its counterpart the BTOD subroutine, simplifies manipulation of binary bytes (8 bits), words (16 bits), longwords (32 bits), and quadwords (64 bits). DTOB converts Synergy Language decimal data to binary values, while BTOD does the converse.

Upon entry, many non-Synergy Language subroutines and library routines require binary bytes, words, or longwords to be passed by reference. Because all data in Synergy Language is in ASCII format and is manipulated by descriptor, you can either use integer data fields or the DTOB and BTOD routines to convert to and from binary and decimal data.



Quadwords (64 bits) are only supported on 64-bit machines such as HP OpenVMS Alpha or I64, HP Tru64 UNIX, HP-UX 64-bit, and Sun Solaris 64-bit.



We recommend you use the integer data type or the %INTEGER function (see [%INTEGER on page 9-169](#)) for greater efficiency.

The DTOB routine converts a signed Synergy Language decimal field to a binary value. The decimal value is obtained from a Synergy Language expression (variable, literal, computation, and so forth), and the binary value is placed in an alpha or decimal, Synergy Language-named field. Note how the following conditions affect DTOB:

- ▶ If the destination field (*binary*) is **a1** or **d1**, DTOB performs a byte conversion and yields the same result as the ASCII subroutine.
- ▶ If *binary* is **a2** or **d2**, DTOB performs a word conversion.
- ▶ If *binary* is **a4** or **d4**, DTOB performs a longword conversion.
- ▶ If *binary* is **a8** or **d8**, DTOB performs a quadword conversion.
- ▶ If *binary* is anything other than one, two, or four bytes, it is treated as though it were only one byte, and no error occurs.

See also

[BTOD on page 9-43.](#)

Examples

If you enter the following data:

```
fld1      ,a1
fld2      ,a2
fld4      ,a4
fld5      ,a5
fld6      ,d6
```

the statements below perform these results:

```
xcall dtob(-1, fld5)      ;Yields %XXXXXXXXFFFF in fld5
                           ; (equiv. to ^i(fld5)=-1)
xcall dtob(255, fld4)     ;Yields %X000000FF in fld4
                           ; (equiv. to ^i(fld4)=255)
xcall btod(fld5, fld6)    ;Yields fld6 = -1
                           ; (equiv. to ^d(fld6)=^i(fld5))
xcall btod(fld4, fld2)    ;Yields fld2 = "255"
                           ; (equiv. to ^d(fld2)=^i(fld4))
```


EMPBUF – Write out modified I/O buffers

VMS

```
xcall EMPBUF (channel)
```

Arguments

channel

An expression that evaluates the channel number specified in a previous OPEN statement. (n)

Discussion

The EMPBUF subroutine writes out all modified I/O buffers associated with the file or device opened on the specified channel.



EMPBUF is only available for consistency with previous versions of Synergy Language; we recommend that you use the FLUSH statement instead.

The EMPBUF subroutine uses the RMS service call \$FLUSH to instruct the file system to flush its buffers. The file system and the device, however, determine whether EMPBUF physically writes out the buffers. If records were locked on a previous operation, they remain locked.

See also

- ▶ Your OpenVMS operating system documentation for more information about \$FLUSH.
- ▶ [FLUSH on page 4-61](#).

Examples

```
store(5, rec1)           ;Write out important stuff to file
store(5, rec2)
xcall empbuf(5)          ;Make sure records have been flushed to disk
```

ENDFL – Position the file pointer after the last record of a file

VMS

```
xcall ENDFL(channel)
```

Arguments

channel

The expression that contains the channel number opened to the file. (**n**)

Discussion

The ENDFL subroutine is used with sequential and relative files to position the file pointer after the last record of the file.

After calling ENDFL, a WRITES writes the designated record after the last record of the file. Any records that are locked due to a previous operation are unlocked.

Examples

```
xcall endfl(5)  
writes(5, rec)
```

ENVRN – Identify the current operating system and machine

```
xcall ENVRN(system [, machine])
```

Arguments

system

The variable to be loaded with the code of the current operating system: (**a** or **n**)

-1	Other systems
6	OpenVMS
8	UNIX
18	Windows XP
19	Windows Server 2003
20	Windows Vista
21	Windows Server 2008
22	Windows 7
23	Windows Server 2008 R2

machine

(optional) The variable to be loaded with the system code of the current machine: (**a** or **n**)

003	SCO OpenServer
004	IBM AIX 5L 32-bit
009	HP-UX 32-bit
020	Sun Solaris 32-bit
021	HP Tru64 UNIX
028	Linux 32-bit (Red Hat or SUSE)
101	Windows 32-bit
104	Windows 64-bit (x64)
200	HP OpenVMS Alpha
202	HP OpenVMS I64
304	IBM AIX 5L 64-bit
309	HP-UX 64-bit (PA-RISC)
320	Sun Solaris 64-bit (SPARC)

420	Sun Solaris 64-bit (x64/x86)
428	Linux 64-bit (x64)
509	HP-UX 64-bit (Itanium)

Discussion

The ENVRN subroutine returns decimal codes that identify the operating system and, optionally, the machine on which the program is running and the CPU code. You can use these values to control program actions that depend on the operating system or machine.



Prior to Synergy Language 7.5.1, the Windows XP operating system returned a value of 15.

Examples

```
.define TTCHN          ,1
record
    opsys              ,d2

proc
    open(TTCHN, i, "tt:")
    xcall envrn(opsys)
    display(TTCHN, "This program is running on ")
    case opsys of
        begincase
            6:
                writes(TTCHN, "OpenVMS")
            8:
                writes(TTCHN, "UNIX")
            18:
                writes(TTCHN, "Windows XP")
            19:
                writes(TTCHN, "Windows Server 2003")
        endcase
    else
        writes(TTCHN, "something new")
    stop
end
```

%ERLIN – Return the line number of the statement that caused an error

line_num = %ERLIN

Return value

line_num

The line number of the statement that caused the most recent trappable error. (**n**)

Discussion

%ERLIN is identical to DIBOL's %ERLINE, and you can substitute one for the other.

Examples

The following subroutine displays the most recent Synergy Language error number, text, and line number.

```
subroutine proc_error
.define TTCHN          ,1
record
    msg                ,a45

proc
    xcall ertxt(%ernum, msg)
    writes(TTCHN, "Error#" + %string(%ernum) + ": " msg +
    &          " at line " + %string(%erlin))
    return
endsubroutine
```

%ERNUM – Return the number of the most recently trapped error

error = %ERNUM

Return value

error

The code number of the error that was most recently trapped. (**n**)

Discussion

%ERNUM is the same as %ERROR without any of the optional arguments.

Examples

See the example for [%ERLIN](#) on page 9-107.

%ERR_TRACEBACK – Generate error traceback

status = %ERR_TRACEBACK(*err_text*[, *clearall*])

Return value

status

One of the following values: (**n**)

1 (True) Traceback information is available.

0 (False) The *clearall* option was used, or no further traceback information is available.

Arguments

err_text

The associated error text. (**a**)

clearall

(optional) Any nonzero value, which clears all traceback information. (**n**)

Discussion

Err_text has a maximum of 256 characters.

%ERR_TRACEBACK returns error information in first in, first out order, so the location of the error is the first piece of information returned. If %ERR_TRACEBACK returns false, no additional information is returned.

When the optional *clearall* argument is used, the *err_text* argument may be omitted as long as a placeholder comma is used.

%ERR_TRACEBACK is for use with \$ERR_CATCH and is mandatory to clean up memory. It should always be used in a loop, as shown below, to return all lines of traceback information:

```
while (%err_traceback(error_description))
  begin
    writes(logfile_chan, error_description)
  end
```

See [“Error catching” on page 4-122](#) for more information about \$ERR_CATCH and for a more detailed example.

ERRMOD – Return the module in which the most recent error occurred

```
xcall ERRMOD([module][, line])
```

Arguments

module

(optional) A field or record that is returned with the name of the module in which the most recent error occurred. (**a**)

line

(optional) A field that is returned with the line number of the module in which the most recent error occurred. (**n**)

Discussion

The module name is moved to *module* according to the rules for moving alpha data. The line number is moved to *line* according to the rules for moving numeric data.

With objects, the method signature is returned, and it can be quite large.

See also

[“Assignment Statements” on page 3-26](#) for the rules for moving alpha and numeric data.

%ERROR – Return the number of the most recently trapped error

```
error = %ERROR ([line], [status], [stv], [source_file], {reserved1}{, reserved2})
```

or

```
xcall ERROR (error, [line], [status], [stv], [source_file], {reserved1}{, reserved2})
```

Return value

error

The code number of the most recently trapped error. If no error has occurred yet, %ERROR returns a value of 0. **(n)**

Arguments

line

(optional) The variable that will be loaded with the line number of the statement that generated the error. (Program listings display the line number assigned to each line in a program. Line number information is only maintained for source lines in files that contain the PROC statement.) **(n)**

status

(optional) Returns the error number associated with the last system call. **(n)**

stv

(optional) Returns the Synergy DBMS error code associated with the last I/O statement on Windows and UNIX or the RMS STV value associated with the last I/O statement on OpenVMS. **(n)**

source_file

(optional) The variable that will be loaded with the number of the source file in which the error was generated. **(n)**

reserved1

reserved2

(optional) Reserved for internal use. **(a)**

Discussion

You will normally use %ERROR in the part of your program code that handles trappable errors. You can pass the result of %ERROR to the ERTXT subroutine to display the text of the error.

The most recent error and line numbers are global and are accessible to all routines in a program. You can easily access the same information with the %ERNUM and %ERLIN functions.

If the length of *line* is too short to hold all significant digits of the value being stored, an “Argument specified with wrong size” error (\$ERR_ARGSIZ) occurs.

On .NET, *source_file* is always returned 0.

On Windows and UNIX, only statements that perform file I/O can generate a Synergy DBMS error. On all operating systems, subsequent I/O can change the value, so retrieving the *stn* should be done as close to the I/O statement causing the error as possible. This value should be added to any application error logs in the event a failure occurs and Synergy/DE Developer Support is contacted.

Examples

The following subroutine displays the most recent Synergy Language error number, text, and line number.

```
subroutine proc_error
.define TTCHN,1
record
    msg          ,a45

proc
    xcall ertxt(%error, msg)
    writes(TTCHN, "Error#" + %string(%error) + ": " + msg +
    &      " at line " + %string(%erlin))
    return
endsubroutine
```

ERTXT – Return an error message

```
xcall ERTXT(error, message, [len][, position])
```

Arguments

error

The number or mnemonic of the error for which you're requesting an error message. **(n)**

message

The variable that will be loaded with the error message, stored left-justified over blanks. **(a)**

len

(optional) The length of the message text. **(n)**

position

(optional) Returned with the position of the message text, after the facility, severity, and mnemonic. **(n)**

Discussion

The ERTXT subroutine enables your program to report trappable errors accurately to the person running the program. It returns a descriptive error message that corresponds to the specified error code.

The message can have up to 45 characters. If the variable is too short to hold the message, only the leftmost portion of the message is stored.

If the specified error number is undefined, the ERTXT subroutine returns the error text from \$ERR_UNDFERR ("Undefined error").

ERTXT is functionally identical to DBL\$ERRTXT.

VMS

If system option #35 is set, the returned message includes the facility, severity, and mnemonic, and the *position* argument is loaded with the starting position of the message text. Otherwise, if option #35 is not set, the text does not include the facility, severity, or mnemonic, and the *position* argument has a value of 1.

Examples

The following subroutine displays the most recent Synergy Language error number, text, and line number.

```
subroutine proc_error
  .define TTCHN,1
  record
    msg          ,a45

  proc
    xcall ertxt(%error, msg)
    writes(TTCHN, "Error#" + %string(%error) + ": " + msg +
      &      " at line " + %string(%erlin))
    return
  endsubroutine
```

EXEC – Execute a non–Synergy Language program

UNIX, VMS

```
xcall EXEC (program[, argument, ...])
```

Arguments

program

An expression that contains the name of the program to execute. (**a**)

argument

(optional) One or more variables or literals that contain any arguments the program requires. A maximum of 31 arguments is allowed. (**a** or **n**)

Discussion

The EXEC subroutine terminates the calling Synergy Language program and executes a non–Synergy Language program using the **execvp** function call. On OpenVMS, you must set the logical VAXC\$PATH to a search list where the program image will be found.

The EXEC subroutine performs the same cleanup as the STOP command. In effect, EXEC acts as a chain to a non–Synergy Language program, including shell scripts. You cannot return from a successful EXEC.



Do not use this routine in an ELB that is being used with x/ServerPlus.

Examples

The example below executes the program **/usr/games/rogue** with **-s** as an argument. Note that the WRITES statement is only executed if the EXEC subroutine fails.

```
.define TTCHN      ,1
record
    program        ,a*,    "/usr/games/rogue"
    arg1           ,a*,    "-s"
proc
    open(TTCHN, o, "tt:")
    onerror nope
    xcall exec(program, arg1)
nope,
    writes(TTCHN, "Cannot execute: " + program)
stop
end
```

EXITE – Exit the routine with a specific error

```
xcall EXITE(error[, trap_flag])
```

Arguments

error

The mnemonic or literal of the Synergy Language error to generate. (**n**)

trap_flag

(optional) Specifies whether or not the error is trappable, as follows: (**n**)

- 0** Error is not trappable.
- 1** Error is trappable. (default)

Discussion

The EXITE subroutine enables you to exit the current external subroutine or function with the specified Synergy Language error. It also enables you to specify whether that error can be trapped.

If the value of *trap_flag* is one, the error can be trapped. However, a call must be made to ONERROR prior to the function or subroutine call that contains the EXITE, or the error will not be trapped. If the value is zero, the error cannot be trapped. See the “[Error Messages](#)” chapter of *Synergy Language Tools* for a complete list of errors.

To generate a nontrappable error, set *trap_flag* to 0.

Examples

The following hypothetical program checks whether a system is valid. If the system is not valid, it generates a false Synergy Language error (without disclosing the actual location of the check) so the user will call his or her developer to have “problem” fixed.

Program #1:

```
main
proc
    open(15, o, 'tt:')
    xcall chk_sys
    stop
endmain

subroutine chk_sys
record
    serial, a80
external function
    security, ^val
```

```

proc
  xcall serial(serial)
  if (%security(serial) .eq. %FALSE) then
    xcall exite($ERR_NOOPEN, 0)
  return
endsubroutine

```

Output:

%DBR-E-NOOPEN, Channel has not been opened

%DBR-I-ATLINE, At line 4 in routine check_system (check_system.dbl)

Program #2:

```

main
proc
  open(15, o, 'tt:')
  onerror errlbl
  xcall chk_sys
  offerror
  stop

errlbl,
  offerror
  writes(15, "Security error")
  stop

endmain

subroutine chk_sys
record
  serial, a80
external function
  security,      ^val

proc
  xcall serial(serial)
  if (%security(serial) .eq. %FALSE) then
    xcall exite($ERR_NOOPEN, 1)
  return
endsubroutine

```

Output:

Security error

EXITERROR – Exit the routine with a specific error

The EXITERROR subroutine is functionally identical to the EXITE subroutine. See [EXITE on page 9-116](#) for a thorough discussion.

^F – Access variable as implied-decimal data type

^F accesses a variable as implied-decimal data type.

^F is available only for consistency with previous versions of Synergy Language. It is a casting function and is equivalent to the ^D data reference operation. See [^D on page 9-66](#) for more information.

%FALSE – Return a value of zero

%FALSE

Return value

%FALSE returns the value 0. (**n**)

Discussion

%FALSE is a compile-time function that returns a value of 0.

The %FALSE function, like the %TRUE function, helps you clarify the intentions of certain statements. It's much easier to understand what a statement is doing if you make a reference to the value %FALSE instead of the decimal literal 0.



You cannot use %FALSE if you're including the file **tools.def** (which contains the flag definitions and value-control arguments required by the UI Toolkit utilities) in your Synergy Language programs. Instead, we recommend that you use the FALSE identifier. If you're using UI Toolkit, you don't have to define FALSE, because it's already defined in **tools.def**.

Examples

The statement

```
varxy = %false
```

is logically equivalent to the following:

```
varxy = 0
```

Here's another example:

```
record
    trans          ,a10
    len            ,d3
    tstmode        ,d1,    %false

proc
    xcall getlog("TST", trans, len)
    if (len) then
        tstmode = %true
    xcall build_it(tstmode)
    xreturn
end
```

FATAL – Chain to another program if a fatal error occurs

```
xcall FATAL (action[, filename])
```

Arguments

action

One of the following codes that tells the runtime what to do if an error occurs: (**n**)

- 0** Return to system level. The default program is not invoked, and any second argument is ignored.
- 1** Chain to the default, user-designated program. If this program doesn't exist, control returns to the terminal.
- 2** Chain to the specified program (*filename*). If *filename* is not specified, control returns to the terminal.
- 3** Return the name of the default, user-designated program in the *filename* variable. If this program doesn't exist, *filename* is filled with blanks.

filename

(optional) An expression that contains the file specification for the program to which to chain if a fatal or untrapped error occurs. It may also be a variable that will be returned with the name of the default user-designated program. (**a**)

Discussion

The FATAL subroutine provides a graceful way to catch fatal and trappable errors. It specifies a program to which the current program should chain if an error occurs and sends an informational message to that program.

When your program chains to the program specified by the FATAL subroutine, a message containing error information is also sent to that program. To receive the message, you must use the RECV statement in the specified program.

The message doesn't include field names. It has the following format:

Character position	Data type	Information
1 – 3	d3	Synergy Language fatal error number
4 – 13	d10	Windows Error code from GetLastError() UNIX Last errno value OpenVMS System service error code
14 – 23	d10	The line number of the statement that caused the error

Character position	Data type	Information
24 – 53	a30	The name of the routine that caused the error
54 – 83	a30 ^a	The name of the program that contains the error

- a. A maximum of 25 characters is returned for the program name. (The remaining 5 bytes are required to store "MAIN\$", which always precedes the program name but which is not returned as part of the name. See [MAIN-ENDMAIN on page 4-105](#) for more information.)

For example, you can define the following record in the program to which you want to chain:

```
record error
  err1      ,d3
  err2      ,d10
  erln      ,d10
  modul     ,a30
  prgnm     ,a30
```

To receive the message, issue the following RECV statement in the receiving program:

```
recv(error, none)
```

If no message is available, control transfers to the statement labeled **none**.

The FATAL subroutine left-justifies the routine and program names it sends to the program to which it chains on a fatal error.

WIN, UNIX

The FATAL subroutine uses the environment variable DBL_FATAL_IMAGE as *filename* (when action 3 is specified) and as the default program to which to chain.

WIN, VMS

The FATAL subroutine uses the logical name DBL\$FATAL_IMAGE as *filename* (when action 3 is specified) and as the default program to which to chain.

Examples

The example below tells the program to chain to the **ferr** file if a fatal error occurs.

```
xcall fatal(2, "ferr")
```

In the example below, control is returned to the terminal if a fatal error is detected. Any previous filenames specified in the FATAL routine are disabled, and the program terminates as usual.

```
xcall fatal(0)
```

FILL – Fill a variable with a pattern

```
xcall FILL(pattern, destination)
```

Arguments

pattern

An expression that contains a value to be loaded into the destination variable. **(a)**

destination

The variable that will be loaded with the specified pattern. **(a)**

Discussion

The FILL subroutine fills an alpha variable with the specified pattern.

All character positions of the destination variable are loaded left-justified. If the destination is longer than the pattern, the pattern is repeated until the destination variable is filled. For example, loading a 15-byte variable with the pattern “ABCD” gives the variable the following contents:

ABCDABCDABCDABC

Examples

```
subroutine sub
record
    dest      ,a10
    pat       ,a16, "0123456789ABCDEF"

proc
    xcall fill("]", dest)           ;dest = "]]]]]]]]]]]"
    xcall fill(pat, dest)           ;dest = "0123456789"
    xcall fill(pat(4,6), dest)      ;dest = "3453453453"
    xcall fill("A.", dest)          ;dest = "A.A.A.A.A."
    return
endsubroutine
```

FILNM – Return the file specification of the open channel

```
xcall FILNM(channel, file_spec[, length])
```

Arguments

channel

The open channel. If the channel is not open, a “Channel has not been opened” error (\$ERR_NOOPEN) is generated. **(n)**

file_spec

The variable that will be loaded with the file specification, left-justified over blanks. (See “[File specifications](#)” on page 1-43 for a description of a file specification layout.) **(a)**

length

(optional) The length of the filename string. **(n)**

Discussion

The FILNM subroutine returns the file specification of the file that is currently open on a specified channel.

The information returned by the FILNM subroutine can clarify an I/O-oriented error when it is trapped by a program. The program can report the error number, use the ERTXT subroutine to provide a description of the error, and then call FILNM to report the specification of the file involved.

If *file_spec* is too short to hold the entire file specification, only the leftmost portion of the specification is returned.

The FILNM subroutine of a channel opened to TT: returns TT:.

Examples

```
subroutine proc_error
    a_chn          ,d          ;Optional
.define TTCHN      ,1
record
    msg            ,a45
    file           ,a80

proc
    xcall ertxt(%ernum, msg)
    writes(TTCHN, "Error# + %string(%ernum) + ": " + msg + "at line "
    &         + %string(%erlin))
```

```
if (^passed(a_chn))  
  begin  
    xcall film(a_chn, file)  
    writes(TTCHN, "File: + file(1:%trim(file))")  
  end  
  return  
endsubroutine
```

FLAGS – Set runtime option flags

```
xcall FLAGS (flags [, control])
```

Arguments

flags

The runtime option flags. Each flag is a single digit, and the digit positions are assigned ordinal numbers from right to left. Thus, the low-order digit is digit number one. Synergy Language examines the flags value digit by digit to determine which flags are set, and hence, which functions are to be performed. The Discussion section below contains a detailed description of the flags and their meanings. (**n**)

control

(optional) A value that controls how the individual digit positions in the flag specification value are interpreted. If *control* is not specified, each digit in *flags* is interpreted as follows: (**n**)

- 0** Reset the associated flag.
- n** Set the associated flag, where *n* is any nonzero decimal number.

We recommend that you use the *control* value when setting and resetting your *flags* values. If *control* is present, its value is interpreted as follows:

- 0** Reset all flags whose corresponding digit position contains a nonzero number. Leave all other flags unmodified.
- 1** Set all flags whose corresponding digit position contains a nonzero number. Leave all other flags unmodified.
- 2** Set the *flags* variable to a value that reflects the current settings of the flags and leave all flags unmodified. Each digit of the *flags* variable is set to zero if the associated flag is currently reset or to one if the flag is set. If the length of the variable is less than 10, the settings are returned as in a decimal-to-decimal assignment. (See [“Moving decimal/packed data to a decimal/packed destination” on page 3-33](#) for more information about decimal-to-decimal assignments.)

Discussion

The FLAGS subroutine, like the DFLAG subroutine, controls the settings of various runtime option flags. These flags control the way the runtime acts in response to the execution of Synergy Language statements.

When program execution begins, all program-controlled flags are normally reset, and the program can alter them as needed. The flag settings are not preserved across program chaining. In effect, the STOP statement resets all of the flags as its last action, even if another program is being started. The flag settings in one program do not affect the settings in a program that is running concurrently from another terminal.

Remember that the rightmost digit in the flag specification string corresponds to flag number one, as shown in [figure 9-2](#). Because Synergy Language interprets the *flags* argument as if it were storing it to a **d10** variable, you don’t have to specify a flag if that flag and all flags to its left have a value of zero.

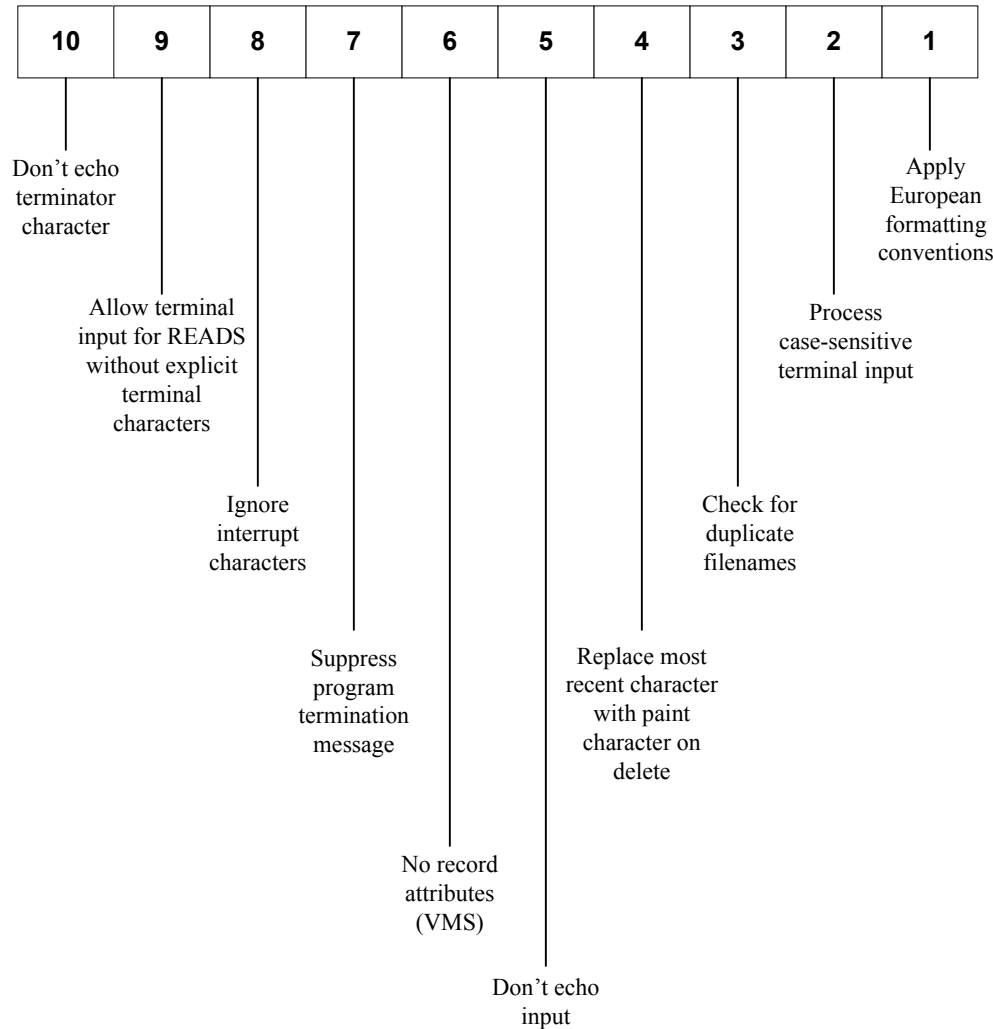


Figure 9-2. FLAGS runtime option flags.

All rules that apply when storing to a decimal variable apply here. For example, if you specify

```
xcall flags(00100)
```

Synergy Language interprets your argument as 0000000100. (All flags corresponding to these zeros are cleared.)

A more detailed description of the flags is provided below. The flag specification digit position (which we refer to as the flag number) is shown on the left, and the description of the flag is listed on the right.

Flag	Description
1	When flag 1 is set, the runtime applies European conventions to alpha variables during formatted transfers of decimal values. Specifically, the use of commas and periods are reversed (commas are used as decimal points, and periods are used as separators). For example, <pre>alpha = 123456 , "Z,ZZZ.XX"</pre> normally assigns the value 1,234.56 to alpha. But when flag 1 is set, the European conventions are used, the assignment statement is coded as <pre>alpha = 123456 , "Z.ZZZ,XX"</pre> and alpha has a value of 1.234,56. When flag 1 is reset, the runtime applies American conventions to alpha variables. Commas are used as separators, and periods are used as decimal points. If you are using xServerPlus and you set flag 1, you need to set it in <i>each</i> routine. (xServerPlus resets flag 1 after every call so that data passed across the wire is formatted properly.) We recommend you use the LOCALIZE subroutine instead of flag 1. (See LOCALIZE on page 9-206.) The LOCALIZE subroutine overrides this flag.
2	When flag 2 is set, the runtime processes case-sensitive terminal input. When the flag is reset, the runtime translates all lowercase input to uppercase before processing it.
3	Setting flag 3 protects the runtime against accidentally replacing an existing file with a new file. Normally, the runtime does not check for duplicate filenames, and OPENing a file in output mode or using the RENAM or ISAMC subroutines can overwrite an existing file. When flag 3 is set, the runtime checks that the file about to be created does not already exist. If it does, a "Cannot supersede existing file" error (\$ERR_REPLAC) is generated. When flag 3 is reset, the runtime won't check for duplicate filenames. Flag 3 works differently on OpenVMS with the RENAM and OPEN statements. To generate an \$ERR_REPLAC error, you must also specify the output filename's version number in the RENAM or OPEN statement.

Flag	Description
4	Flag 4 is only useful with display terminals. When typing terminal input, the user can delete the most recently typed character by entering an ASCII delete code (using the DEL, DELETE, or RUBOUT key, depending on the keyboard). The FLAGS subroutine uses the current terminal setting for the default ASCII delete code. When flag 4 is set, typing a DEL code sends a backspace, a paint character, and another backspace to the screen. The most recent character is replaced with the paint character, and the cursor is left in the deleted character's position. The default paint character is a blank, but you can redefine it using the PAINT subroutine. Use the %PAINT intrinsic function to determine the decimal character code for the current paint character. Pressing CTRL+U clears and repaints the current line and positions the cursor at its leftmost character position. CTRL+R is ignored. When flag 4 is reset, the characters are echoed back in the order in which they are deleted (the reverse of the order in which they were input), with the first deleted character preceded by backslash and the last deleted character followed by a backslash. For example, if the user enters ABDCEFG, presses the RUBOUT key three times, and then enters 123, the sequence appears on the screen as follows: ABCDEFG\GFE\123 and the input is ABCD123. This method of handling character deletions is inappropriate for most CRT applications. When the user presses CTRL+U to delete a line, the two-character ^U sequence is transmitted to the terminal, followed by a carriage return and a line feed. The cursor is thus positioned at the beginning of the next line to accept further input. When the user presses CTRL+R, a ^R is echoed at the terminal, followed by a carriage return, a line feed, and the currently accumulated input line.
5	Setting flag 5 prevents input received from any channels opened to terminal-type devices from being echoed back to the primary terminal. This flag is useful when the input is confidential, like a password. When flag 5 is reset, the characters that the user types are echoed on the screen.
6	Flag 6 is ignored on all systems except OpenVMS. It changes a file open for output to have no record attributes and disables carriage control on O mode files (except stream files). These files cannot be TYPED or PRINTed.
7	Flag 7 suppresses the program termination message that is displayed at the terminal running the program when the STOP statement returns control to the monitor. When flag 7 is reset, the program termination message is displayed after the STOP statement is issued. If the program terminates abnormally, the appropriate error message is displayed regardless of the value of this flag.

Flag	Description
8	Setting flag 8 ignores interrupt character(s) (CTRL+C on Windows, the stty setting for interrupt on UNIX, and CTRL+C and CTRL+Y on OpenVMS) typed at the terminal. When flag 8 is reset, interrupt characters perform their normal functions. On OpenVMS, we recommend using the SETCTL subroutine instead. (See SETCTL on page 9-301 .)
9	Setting flag 9 permits terminal input for a READS statement without an explicit terminator. A field entry is terminated implicitly when the field is filled, but a user can still terminate input with a terminator character before the field is full. (In other words, flag 9 causes READS to terminate the same way GETS does.) When flag 9 is reset, the READS statement ignores any nonterminating character that the user types once the field is full and echoes an ASCII bell character to warn the user.
10	When flag 10 is set, the terminator character won't be echoed on the screen during terminal input. When flag 10 is reset, the terminator character is echoed on the screen.



To make your code easier to maintain, use the flag number as the nonzero digit that sets the flag. For example, if *flags* contains a value of “7004301,” you know at a glance that flags 7, 4, 3, and 1 are set. (Of course, this tip won't work with flag 10.)

See also

[DFLAG on page 9-98](#).

Examples

The example below sets flag 3 and resets all other flags.

```
xcall flags(300)
```

The example below resets flag 3 and leaves all other flags unmodified.

```
xcall flags(300, 0)
```

The example below sets flag 3 and leaves all other flags unmodified.

```
xcall flags(300, 1)
```

The example below loads the current flag settings into a decimal variable called **now**.

```
xcall flags(now, 2)
```

FORK – Split the current process

UNIX

```
xcall FORK(child_id)
```

Arguments

child_id

Returned with the process ID of the child process or with a zero, depending on which process is returning. (**n**)

Discussion

The FORK subroutine splits the current process by invoking the UNIX/POSIX system call **fork(2)**.

This subroutine invokes **fork(2)**, which splits the current process into two processes, both of which return from the FORK call.

The original process is the parent. When it returns, the return value is the process ID of the created subprocess.

The created subprocess, or child, is an exact copy of the parent except that all channels opened by the parent have been closed in the child. (Therefore, you must reopen any files needed by the child.) It also returns from the FORK call, but the return value is 0, so your program can detect the fact that it is the child process.

Both processes return from the call with exactly the same memory contents and files open. Your program must examine the *child_id* argument to decide on the different actions for the two processes.

The child process can call the JBNO subroutine to find its parent.



When using the Synergy socket API with FORK, you may want to specify the SS_PURGE option on the %SS_CLOSE routine to ensure that the FORK'd process does not force the connection to be terminated for the parent process when it closes the socket.

The accepted socket (with %SS_ACCEPT) on the server needs to be purged after (and only after) it knows the client has cloned the socket and is using it, because the child now owns the socket. The server socket should be purged in the child program.



If channels are open to a Synergy/DE™ xServer from a program using FORK, and the server channel for a file is closed by either the parent or child program, subsequent I/O to the channel will generate errors.

Child_id should be no smaller than an **i4** or **d10** to ensure getting the entire process ID.

Examples

This example routine forks a process and returns 1 or 0. On 1, the CHILD returns its process ID and its parent's; on 0, the PARENT returns its process ID and the new child's.

```
.define CHILD      ,1
.define PARENT     ,2

subroutine split
    a_which ,n
    a_parent ,n
    a_child ,n

record
    pid      ,i4

proc
    xcall fork(pid)
    a_which = .not.pid                ;Set a_which to parent or child
    if (pid) then                    ;This is the parent process
        begin
            a_child = pid
            a_parent = %jbno()
        end
    else                                ;This is the child process
        a_child = %jbno(a_parent)
    xreturn
endsubroutine
```

FREE – Release record locks on a channel

```
xcall FREE(channel)
```

Arguments

channel

The number of the channel for which you want to release locks. (**n**)

Discussion

The FREE subroutine releases all record locks on a channel.

See also

[“Record Locking” on page 4-243.](#)

Examples

```
subroutine clean_up
  .define MAXCHN      ,50
  record
    chn                ,d3
    opn                ,d1

  proc
    for chn from 1 thru MAXCHN
      begin
        xcall chnopen(chn, opn)
        if (opn)
          xcall free(chn)
      end
    return
  endsubroutine
```

%FSTAT – Return the value of the last floating point call

VMS

value = %FSTAT ([*source_type*])

Return value

value

The return value of the last floating point call as an implied-decimal field. (**d**)

Arguments

source_type

(optional) The format in which the result was returned from the floating point function invoked using the FXSUBR subroutine.

Discussion

This function returns the value of the result of the last floating point function called through the FXSUBR subroutine, as an implied-decimal field. If you want to retrieve the result in another format, use the FSTAT subroutine instead of the %FSTAT function.

Source_type specifies what type of floater the result was, or what kind of routine was called. This can either be specified as a letter (“F”, “D”, “G”) or a number indicating the OpenVMS data type as defined in DBLSTARLET:\$DSCDEF.DBL, DSC\$K_DTYPE_F, DSC\$K_DTYPE_D, DSC\$K_DTYPE_G.

See also

[FSTAT on page 9-137.](#)

Examples

The following program demonstrates how to encode floating point values from Synergy Language, how to pass them to an OpenVMS routine, how to retrieve the result as a floating point value for passing into another OpenVMS routine, and how to retrieve the value into a Synergy Language implied-decimal field.

```
main MATH

.include "$DSCDEF" library "dblstarlet" ;Needed for creating descriptor

.align QUAD
record
    group flt_desc ,a                ;Descriptor pointing to
        flt_length ,i2              ; d_float, "casting" it as a
```



```

        flt_dtype      ,i1              ; OpenVMS D_FLOAT
        flt_class      ,i1
        flt_addr       ,i4
    endgroup
    d_float             ,i8              ;Where we store the native D_FLOAT
    sin_rtn             ,d_addr          ;Address of sine routine
    asin_rtn            ,d_addr          ;Address of arcsine routine
    number              ,f28.10         ;Start number
    result              ,f28.10         ;Result

proc
    xcall flags(7000000,1)
    open(1, o, "TT:")

    flt_desc.flt_class = DSC$K_CLASS_S
    flt_desc.flt_dtype = DSC$K_DTYPE_D
    flt_desc.flt_length = ^size(d_float)
    flt_desc.flt_addr = ^addr(d_float)
;   Start standard OpenVMS descriptor for a D_FLOATING data type

    sin_rtn = ^xtrnl(MTH$DSIND)
    asin_rtn = ^xtrnl(MTH$DASIND)
;   Store addresses of routines for fxsubr

    number = 30.00
;   Set up Synergy Language implied-decimal

    xcall lib$cvtdx_dx(number, ^ref(flt_desc))
;   Convert the number to the D_FLOAT - descriptor tells routine
;   what to convert from and to.

    xcall fxsubr(sin_rtn, ^ref(d_float))
;   Call the OpenVMS math function.

    xcall fstat(^ref(flt_desc), "D")
;   Return the floating point return value into
;   our floating point descriptor. The "D" tells the routine to
;   convert FROM D_FLOAT format

    xcall lib$cvtdx_dx(^ref(flt_desc), result)
;   Convert it to decimal.

    writes(1, "The sine of " + %string(number) + " is " + %string(result))

    xcall fxsubr(asin_rtn, ^ref(d_float))
;   Call the inverse function.

```

System-Supplied Subroutines, Functions, and Classes

%FSTAT

```
        result = %fstat(DSC$K_DTYPE_D)
;   Return the floating point return value into our implied-dec.
;   We must inform FSTAT what float format to convert FROM.

        writes(1, "The arc sine of the sine of " + %string(number) + " is " +
&           %string(result))

        stop
endmain
```

FSTAT – Return the value of the last floating point call

VMS

```
xcall FSTAT(^REF(descrip)[, source_type])
```

Arguments

descrip

A user-created descriptor that must be fully constructed by the programmer and passed by reference. The descriptor specifies the address where the data is to be written and the data type that is to be written. (**n**)

source_type

(optional) The format in which the result was returned from the floating point function invoked using the FXSUBR subroutine. (**n**)

Discussion

This subroutine returns the value of the result of the last floating-point function called through the FXSUBR subroutine. The data type field of *descrip* determines how FSTAT translates the value. You can set up a descriptor for OpenVMS floating-point data types, and thereby retrieve the result in a format suitable for passing directly back into another floating-point function.

Source_type specifies what type of floater the result was, or what kind of routine was called. This can either be specified as a letter (“F,” “D,” “G”) or a number indicating the OpenVMS data type as defined in DBLSTARLET:\$DSCDEF.DBL, DSC\$K_DTYPE_F, DSC\$K_DTYPE_D, DSC\$K_DTYPE_G.

See also

- ▶ [%FSTAT on page 9-134.](#)
- ▶ [FXSUBR on page 9-138.](#)

Examples

See the example for [%FSTAT on page 9-134.](#)

FXSUBR – Dispatch to a floating-point function

VMS

```
xcall FXSUBR (routine, arguments[, ...])
```

Arguments

routine

The address of the OpenVMS floating-point function to call. (**n**)

arguments

The arguments to the specified routine, passed using the same method as to that routine.

Discussion

This function enables Synergy Language code to dispatch to a native OpenVMS floating-point function, saving the return value for later retrieval. This is the only way to enable retrieval of any floating-point result on Alpha and I64.

To capture the result from a floating-point function called through FXSUBR, use the %FSTAT intrinsic function or the FSTAT subroutine. See [%FSTAT on page 9-134](#) and [FSTAT on page 9-137](#).



FXSUBR does not work on the MATH\$ routines (or on any other routine that passes floating-point arguments by value) on either the Alpha or the I64. In addition, on the I64, the return value can only be an F or G floating-point value. An IEEE floating-point value will not be retrieved correctly.

GETCM – Load the contents of a common area into a variable

VMS

xcall GETCM(*variable*)

Arguments

variable

A variable up to 254 characters long that is loaded with the current contents of the process common area. (**a**)

Discussion

The GETCM subroutine calls the OpenVMS system service LIB\$GET_COMMON with the specified argument, thereby enabling your program to transfer data using a common area.

The GETCM subroutine, in conjunction with the PUTCM subroutine, manipulates data within a common area that is maintained by OpenVMS for each process on the system. Because this common area is unique to each process, Synergy Language programs can use this area as a private “scratch pad” space for communication between programs that will be chained to later.

See also

- ▶ [PUTCM on page 9-248.](#)
 - ▶ The LIB\$GET_COMMON section of your OpenVMS runtime library manual.
-

GETDFN – Return the default file specification

```
xcall GETDFN (file_spec[, option])
```

Arguments

file_spec

A field or record returned with the default file specification currently being used by the runtime. **(a)**

option

(optional) Determines whether the ISAM or non-ISAM file specification is returned: **(n)**

0 The non-ISAM default file specification is returned. (default)

1 The ISAM default file specification is returned.

Discussion

The GETDFN subroutine returns the current default file specification being used by the runtime.

Two default file specifications are used by the runtime library, one for opening ISAM files and one for all other operations (opening non-ISAM files; the DELET, PURGE, and RENAM external subroutines; and so forth).

WIN, UNIX

The extension **.ddf** is the initial default for non-ISAM operations and **.ism** is the initial default for ISAM operations.

VMS

SY\$DISK:[].DDF is the initial default used for non-ISAM operations and **SY\$DISK:[].ISM** is the initial default used for ISAM operations.

The default file specification is moved to *file_spec* according to the rules for moving alpha data. See [“Moving alpha data to an alpha destination” on page 3-26](#) for more information.

GETFA – Return file attributes

```
xcall GETFA(channel, keyword, field)
```

Arguments

channel

An expression that evaluates to a channel number, as specified in a previous OPEN statement. **(n)**

keyword

An expression that contains the file attribute to be returned. (See the Discussion section below for a list of valid keywords.) **(a)**

field

The variable that will be loaded with the numeric code of the file attribute specified in *keyword*. **(n or a)**

Discussion

The GETFA subroutine returns the attributes of a file. You will get unpredictable results if you specify attributes in *keyword* that are not characteristic of the file or device.

The following table lists the valid keywords, their corresponding file attributes, the operating systems on which they are available, and whether they require a local file system or will work on both local and remote.

Keyword	File attribute	Value returned in <i>field</i>	OS	System
ALQ	Number of 512-byte blocks	The number of 512-byte blocks allocated to the file. ^a (n)	All	Local
BDT	Backup date and time	The backup date and time, or blanks if none exists. ^b (a)	VMS	Local
BKS	Bucket size	0–63 for indexed or relative files. 0 for all other files.	VMS	Local
CDT	Creation date and time	The creation date and time, or blanks if none exists. ^b (a)	All	Local
DEQ	Default extension quantity	The default extension quantity. (n)	VMS	Local
EBK	End-of-file block	The end-of-file block. (n)	VMS	Local

System-Supplied Subroutines, Functions, and Classes

GETFA

Keyword	File attribute	Value returned in <i>field</i>	OS	System
EDT	Expiration date and time	The expiration date and time, or blanks if none exists. ^b (a)	VMS	Local
FAB	Address of the file access block for the channel	The FAB address. (n)	VMS	Local
FFB	First free byte	The first free byte. ^a (n)	All	Local
GRP	Group decimal number of the file owner	The group number. (n)	All	Local
HBK	Highest virtual 512-byte block	The highest virtual 512-byte block. ^a (n)	All	Local
IDK ^c	Duplicate key flag (indicates whether duplicate keys are allowed for the current key of reference)	0 Duplicate keys not allowed 1 Duplicate keys allowed	All	Local, Remote
IKL ^c	Length for the primary key	The length of the primary key within the record. If the key is segmented, this is the length of the first segment. (n)	All	Local, Remote
IKP ^c	Position for the primary key	The relative position of the primary key within the record. The position of the first byte is 1. If the key is segmented, this is the position of the first segment. (n)	All	Local, Remote
ISZ	Native integer size of the operating system	0 32 bit 1 64 bit 2 128 bit	All	Local, Remote
KRF ^c	Key of reference	The current key of reference number. (n)	All	Local, Remote
MEM	Member decimal number of the file owner	The member number. (n)	All	Local
NDN	Endian system	0 File resides on a little-endian system 1 File resides on a big-endian system	All	Local, Remote
NKY ^c	Number of keys	The number of keys defined for the file. (n)	All	Local, Remote
OMD	Synergy Language OPEN mode	<i>mode:submode</i> (where <i>mode</i> is A, U, I, or O, and <i>submode</i> is S, R, I, B, P, or C). (a)	All	Local, Remote

Keyword	File attribute	Value returned in <i>field</i>	OS	System
ORG	File organization	0 Sequential file 16 Relative file 32 Indexed file On OpenVMS, the code returned is the RMS FAB ORG value.	All	Local, Remote
OST	Operating system type of the file system	1 UNIX 2 OpenVMS 3 Windows	All	Local, Remote
PLG ^c	File prologue level	The file prologue level for an indexed file (2 or 3). (n)	VMS	Local
PRG	Programmer octal number of the file owner	The programmer number. (n)	All	Local
PRJ	Project octal number of the file owner	The project number. (n)	All	Local
PRO	Protection code returned as a 16-digit string of ones and zeros	The protection code. (n)	VMS	Local
RAB	Address of the record access block for the channel	The RAB address. (n)	VMS	Local
RAT	Record attributes	0 No carriage control 1 FTN - FORTRAN carriage control 2 CR - RMS carriage control 4 PRN - print file VFC carriage control 8 BLK - records do not cross block boundaries; sequential files only The code returned is the RMS FAB RAT value. BLK can be combined with other values.	VMS	Local
RDT	Revision date and time	The revision date and time, or blanks if none exists. ^b (a)	All	Local
RFA	Record file address	The six-byte binary record address that uniquely identifies a record within a file of any organization. (a6)	All	Local, Remote

System-Supplied Subroutines, Functions, and Classes

GETFA

Keyword	File attribute	Value returned in <i>field</i>	OS	System
RFM	Record format	0 UDF - undefined 1 FIX - fixed-length 2 VAR - variable-length 3 VFC - variable-length with fixed control 4 STM - stream format 5 STMLF - stream format delimited by LF character 6 STMCR - stream format delimited by CR character The code returned is the RMS FAB RFM value.	VMS	Local
RMT	Remote or local status	0 File is local 1 File is remote	All	Local, Remote
ROP	RMS record options flags	The ROP bits from the RAB. (n)	VMS	Local
RSZ	Record size	The record size in bytes. This value is the actual size of fixed-length records or the maximum size of variable-length records. On OpenVMS, the value returned is the RMS FAB MRS value. (n)	All	Local, Remote
SCH	System channel number	The channel number assigned by the system (used for terminals only). (n)	VMS	Local
SCI	System input channel	The input channel number. (n)	VMS	Local
SCM	System mailbox channel	The mailbox channel number. (n)	VMS	Local
SCO	System output channel	The output channel number. (n)	VMS	Local
SLC	Socket-level cipher	The cipher being used (for example, "AES256-SHA"). (a)	All	Remote
SLE	Socket-level encryption	1 Channel is encrypted 0 Channel is not encrypted	All	Remote

Keyword	File attribute	Value returned in <i>field</i>	OS	System
UIC	Owner UIC (as numeric)	The owner UIC. (n)	All	Local
VER	Synergy version of the facility accessing the file or device	The seven-byte alpha Synergy version. (a)	All	Local, Remote
XAB	Address of the extended attribute block for the channel	The XAB address. (n)	VMS	Local

- a. On ISAM (Windows and UNIX), this is the size of the index file only.
- b. The date and time are represented in the following format: *DD-MMM-YYYY hh:mm:ss[.cc]*, where *DD* is the day, *MMM* is the month abbreviation (for example, JAN), *YYYY* is the four-digit year, *hh* is the hour, *mm* is the minute, *ss* is the second, and *cc* is the hundredth of a second, if supported.
- c. Applies to indexed files only.

We suggest you use the GETRFA qualifier on I/O statements like READ, FIND, and so forth, instead of using the RFA keyword.

You can use the %ISINFO routine to access additional ISAM-specific information. See [%ISINFO on page 9-181](#) for more information.



See the note on [page 6-2](#) for information about inconsistencies that may occur after an I/O error is encountered.

UNIX

PRJ and GRP both return the group ID of the file’s group, and PRG and MEM both return the user ID of the file’s owner. (The only difference is that PRJ and PRG return an octal value, while GRP and MEM return a decimal value.) The UIC value is comprised of the MEM and GRP values (in other words, the group ID and the user ID concatenated together).

WIN

PRJ, GRP, PRG, MEM, and UIC all return 0.

Examples

```
main build_msg_file
record
    src_date          ,a23
    obj_date          ,a23
    exe_date          ,a23

proc
    open(1, i, "vms_msgs.msg") [$ERR_FNF=quit]
    open(2, i, "vms_msgs.obj") [$ERR_FNF=make_um]
    open(3, i, "vms_msgs.exe") [$ERR_FNF=make_um]
    xcall getfa(1, "RDT", src_date)      ;Check revision date of msg src
    xcall getfa(2, "RDT", obj_date)      ;Check revision date of msg obj
    xcall getfa(3, "RDT", exe_date)      ;Check revision date of msg exe
    if (src_date.gt.exe_date) .or. (src_date.gt.obj_date)
        begin

make_um,
    xcall make_msgs                      ;Create new message files
    end

quit,
    stop
endmain
```

GETFNM – Return the file specification of the open channel

The GETFNM subroutine returns the file specification of the file that is currently open on a specified channel. It is identical to the FILNM subroutine. See [FILNM on page 9-124](#) for more information.

GETLOG – Translate an environment variable

```
xcall GETLOG (logical, translation, length)
```

Arguments

logical

An expression that contains the environment variable or logical assignment to translate. **(a)**

translation

The variable that will be loaded with the translation or value of the environment variable or logical assignment. **(a)**

length

The variable that will be loaded with the length of the variable or logical. If no variable or logical is found, *length* has a value of zero. **(n)**

Discussion

The GETLOG subroutine translates an environment variable (or logical assignment on OpenVMS) and stores the translation in an alpha variable. If no environment variable or logical is found, the *translation* value will not be altered.

You can use the GETLOG subroutine to alter a program's behavior based on the value of a particular environment variable. On OpenVMS, you can also use it to check whether logical assignments are set up incorrectly, in which case you can issue an appropriate error message rather than waiting for the program to fail on an OPEN statement.

Length must be at least three digits.

WIN

GETLOG only returns a value for environment variables that are set in the environment. For a list of environment variables that are not set in the environment, see [“Synergy initialization files”](#) in the “Environment Variables” chapter of *Environment Variables and System Options*.

UNIX

Environment variables are case-sensitive.

VMS

Only logicals are checked on OpenVMS; DCL symbols are not.

Examples

```
subroutine chk_log
  a_log      ,a
record
  trans      ,a40
  len        ,d3

proc
  xcall getlog(a_log, trans, len)      ;Try to translate the logical
  if (.not. len)                      ;If no translation
    begin
      display(TTCHN, "Translation for " + a_log)    ;Prompt for a new
      reads(TTCHN, trans, done)                   ; translation for
                                                    ; the logical
      xcall setlog (a_log, trans, len) ;Set logical to new translation
      if (.not. len)                          ;If setting logical doesn't work
        writes(TTCHN, "Error setting" + a_log)      ;Display error
    end
  return
endsubroutine
```

GETRFA – Retrieve the current position in a file

```
xcall GETRFA(channel, rfa[, krf])
```

Arguments

channel

The number of the open channel. (**n**)

rfa

The variable that will be loaded with the file position (or RFA). (**a6**)

krf

(optional) The variable that will be loaded with the key of reference of the last READ or FIND on an ISAM file. (**n**)

Discussion



GETRFA is only available for consistency with DBL version 4. We suggest you use the GETRFA qualifier on I/O statements such as READ, FIND, and so forth, instead of using the GETRFA subroutine.

The GETRFA subroutine retrieves the position of the last record accessed in a file. All read and write operations update this position. For example, after a STORE, the GETRFA subroutine retrieves the RFA of the stored record, not the last record read. You can also retrieve the RFA of the record just stored using the GETRFA qualifier on the STORE statement.



See the note on [page 6-2](#) for information about inconsistencies that may occur after an I/O error is encountered.

The return value (stored in *rfa*) is stored in binary form to be used as an argument to the POSRFA subroutine. The data type of this binary value must not change before POSRFA uses it.



The contents of *rfa* are a special binary value and must be preserved. Be careful in doing assignments or moving the contents into another variable.

If an RFA refers to a deleted record, a “Deleted record” error (\$ERR_DELREC) will be generated when a read is attempted.

See also

- ▶ [GETRFA on page 4-218.](#)
- ▶ [POSRFA on page 9-244.](#)
- ▶ [RFA keyword on page 9-143.](#)

Examples

The following subroutine creates a temporary tag file to sort data from an ISAM file that is in a nonkey field. It reads through the file searching for records with a nonblank entry in the nonkey field. It then creates a temporary tag file and saves the RFA position of the ISAM file and the nonblank field. It sorts the temporary file using the nonblank field as the sort key, then extracts the appropriate records from the ISAM file in sorted order to pass to a print subroutine.

```

subroutine report
.define ISMCHN      ,10
.define TAGCHN      ,11
.define TAGFILE     , "tagfile"
record ism_rec
    key              ,a10
    ism_fld          ,a5
record tag_rec
    tag_rfa          ,a6
    tag_sort         ,a5

proc
    open(ISMCHN, i:i, "isam")
    open(TAGCHN, o, TAGFILE)
    repeat
        begin
            reads(ISMCHN, ism_rec, done1)
            if (ism_fld)                                ;If nonblank...
                begin
                    xcall getrfa(ISMCHN, tag_rfa)
                    tag_sort = ism_fld
                    puts(TAGCHN, tag_rec)                ;Save RFA in tag file
                end
            end
        done1,
        close TAGCHN
        sort(input=TAGFILE, key=tag_sort, record=tag_rec,
        &    options="recsize=11/e")                    ;Note that the /e "exact" switch
        open(TAGCHN, i, TAGFILE)                        ; is necessary since the RFA is
        repeat                                           ; a binary value
            begin
                gets(TAGCHN, tag_rec, done2)
                read(ISMCHN, ism_rec, RFA:tag_rfa)
            end
        end
    end
end

```

System-Supplied Subroutines, Functions, and Classes

GETRFA

```
        xcall print_it(ism_rec)
    end
done2,
    close TAGCHN
    close ISMCHN
    return
endsubroutine
```

GLINE – Get a line of input

```
xcall GLINE ([length, ]buffer[, prompt])
```

Arguments

length

(optional) The variable that will be loaded with the length of the input line. **(n)**

buffer

The variable that will be loaded with the input line stored left-justified over blanks. If the buffer area is too short, only the leftmost portion of the input line is stored; the remainder is discarded. **(a)**

prompt

(optional) The message displayed as a prompt for input. **(a)**

Discussion

The GLINE subroutine gets a line of input from the terminal device. It performs a READS statement and is included to provide source-code compatibility with obsolete Synergy Language products on PDP-11 systems.

The *length* argument contains the same value that %RDLEN returns when a READS statement is used.



If the *prompt* argument is specified, you must also specify the *length* argument.

VMS

The GLINE subroutine uses the OpenVMS LIB\$GET_INPUT routine to get an input line from the device assigned to the logical name SYS\$INPUT.

Examples

```
.define TTCHN      ,1
record
    msg             ,a50
    lng             ,d2
    alng            ,a2 @lng
```

```
proc
  open(TTCHN, o, "tt:")
  writes(TTCHN, "Tracing input until *end*")
  while (msg .ne. "*end*") do
    begin
      xcall gline(lng, msg)
      writes(TTCHN, "lng = " + alng + ", Msg: " + msg)
    end
  stop
end
```

GTPPN – Return information about the current process

UNIX, VMS

`xcall GTPPN (project, programmer, privilege)`

Arguments

project

The number of the project. (**n**)

programmer

The number of the programmer. (**n**)

privilege

Returned with one of the following values: (**n**)

1 The current process has system privileges.

0 The current process does not have system privileges.

Discussion

The GTPPN subroutine checks whether the current process has system privileges and returns the project and programmer number of the current process.

UNIX

Project is the group ID and *programmer* is the user ID. *Privilege* is set to 1 if you have root privileges.

VMS

Project is the group ID and *programmer* is the member ID from the current user's UIC. *Privilege* is set to 1 if the current process has the SYSPRV privilege.

Examples

The following program displays the user identification code and either a 1 if the current process has system privileges or a 0 if the current process does not have system privileges.

```
.define TTCHN          ,1
record output

    project             ,a*, "UIC = ["
                        ,d5
                        ,a*, ", "
    programmer           ,d5
                        ,a*, "], priv = "
    privilege            ,d3

proc
    xcall flags(7004020, 1)
    open(TTCHN, o, "tt:")
    xcall gtpn(project, programmer, privilege)
    writes(TTCHN, output)
    close TTCHN
    stop
end
```

%HEX – Return a hexadecimal string representation of a numeric value

string = %HEX (*value* [, *magnitude*])

Return value

string

A hexadecimal, uppercase ASCII digit string representation of *value*. (**a**)

Arguments

value

The value to be converted to a hexadecimal, ASCII representation. (**n**)

magnitude

(optional) A value that determines the number of characters in the returned string, as follows: (**n**)

1	2-character string
2	4-character string
4	8-character string
8	16-character string

Discussion

%HEX converts a decimal data value to a 32-bit longword value.

If *magnitude* is not present or if it contains a value other than 1, 2, 4, or 8, the resulting string is eight characters long (if *value* can be represented in 32 bits) or sixteen characters long (if *value* cannot be represented in 32 bits).

Examples

The following code fragment compares data from two files. One has data stored in ASCII format, the other in hexadecimal.

```
if (%hex(ascval, 1) .eq. hexval)
```

The code fragment below checks for a specific hexadecimal value in the ASCII data.

```
if (%hex(ascval, 1) .eq. 'F')
```

^I – Access data as integer

$\text{^I}(\text{expression})$

Arguments

expression

An expression to access as integer. (**a** or **n**)

Discussion

^I is a casting function that accesses data as an integer data type.

^I does not convert *expression* to an integer value; the data itself remains unchanged. Only the data type is changed for the reference. (If you need to convert an expression to an integer value, use the `%INTEGER` function as described on [page 9-169](#).)



Any time you cast a data area as an integer, if the high-order bit is set, you will end up with a negative number.

Examples

The following example retrieves the character terminating the `READS` statement as an alpha character and passes the character as an **i1** value to the subroutine **sub**.

```
record
  rec      ,a80
  size     ,i2
  trm      ,a1

proc
  reads(TTCHN, rec)
  xcall rstat(size, trm)
  xcall sub(rec, size, ^i(trm))
  .
  .
  .
end
```


%IMPLIED – Convert an expression to zoned numeric implied-decimal

imp_dec_value = %IMPLIED(*expression*)

Return value

imp_dec_value

An implied-decimal value for the specified expression in zoned numeric representation. (**d**)

Arguments

expression

The expression to convert. (**a** or **n**)

Discussion

%IMPLIED converts an expression to an implied-decimal value in zoned numeric representation.

Expression is converted following the rules for moving data to an implied-decimal destination. (See “[Assignment Statements](#)” on page 3-26 for a description of these rules.) If *expression* is integer or decimal data type, the result won’t have any fractional precision. If *expression* is alpha, the fractional precision equals the number of digits to the right of the decimal point in *expression*.

Rounding and truncation come into effect if you have an alpha variable containing an implied-decimal value that has more than 28 places to the right of the decimal point.

Examples

The following example uses alpha input as data for an implied-decimal arithmetic computation. Note that if the alpha variable contains any invalid hexadecimal characters, a “Bad digit encountered” error (\$ERR_DIGIT) is generated.

```
subroutine math
  a_avar          , a
record
  val1            , d5.2
  val2            , d5.2
  result          , d10.2

proc
  result = (val1*val2)/%implied(a_avar)
  .
  .
  .
  xreturn
endsubroutine
```

INITPORT – Initialize a serial port

WIN, UNIX

```
xcall INITPORT (port, [baud], [parity], [stop_bits], [length], [rts_cts], [xon_xoff], [dtr],
&               [rts][, modem])
```

Arguments

port

An expression that contains a channel number from a Synergy Language OPEN statement. (**n**)

baud

(optional) An expression that contains the desired baud rate (bps) setting. Possible values are 50, 75, 110, 134, 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200, 9600, 19200, and 38400. The default is 9600. (**n**)

parity

(optional) An expression that indicates the desired parity: (**n**)

- 1** No parity (default)
- 2** Even parity
- 3** Odd parity

stop_bits

(optional) An expression that indicates the number of stop bits: (**n**)

- 1** One stop bit (default)
- 2** Two stop bits

length

(optional) An expression that contains the desired word length (number of data bits). Possible values are 5, 6, 7, and 8. The default is 8. (**n**)

WIN

rts_cts

(optional) An expression that enables or disables the Request to Send/Clear to Send software flow control: (**n**)

- 1** Disable RTS/CTS protocol. (default)
 - 2** Enable RTS/CTS protocol.
-

xon_xoff(optional) An expression that enables or disables the XON/XOFF software flow control: **(n)**

- 1** Disable XON/XOFF protocol. (default)
- 2** Enable XON/XOFF protocol.
- 3** Enable receive XON/XOFF only.
- 4** Enable transmit XON/XOFF only.

WIN

dtr(optional) An expression that asserts a Data Terminal Ready signal (RS232 pin 20): **(n)**

- 1** Assert the signal. (default)
 - 2** Retract the signal.
-

rts(optional) An expression that asserts a Request to Send signal (RS232 pin 4): **(n)**

- 1** Assert the signal. (default)
- 2** Retract the signal.

modem(optional) An expression that specifies modem control: **(n)**

- 1** Modem control (default)
- 2** No modem control

Discussion

The INITPORT subroutine initializes a serial (asynchronous) port for subsequent I/O. It gives you more control over communication parameters for your serial port or channel because you can establish these settings from your Synergy Language program.

INITPORT can only be used after an OPEN statement.

WIN

Data Terminal Ready (DTR) is usually asserted by the Data Terminal Equipment (DTE) to indicate a powered-up and ready condition to the attached Data Communications Equipment (DCE). The DTE asserts Request to Send (RTS) to indicate that it is ready to send data. The *rts* argument differs from the *rts_cts* argument, which lets you control the RTS/CTS hardware flow on Windows. This flow is a protocol in which the RTS modem control signal is turned on or off automatically by the

amount of data in the receiving buffer, and in which the CTS modem status signal can be directed to control the transmitter. The *rts* argument simply enables you to assert or retract the signal.

Use of XON/XOFF disables RTS/CTS flow control.

UNIX

INITPORT sets up the port as if an **stty -raw** had been performed.

The *rts_cts* and *dtr* arguments serve no function on UNIX. RTS/CTS flow control is not implemented because it is not available on most UNIX systems. The DTR signal control is not available because it is automatically raised when a channel is opened to a port and lowered when a channel is closed to a port.

Using the sleep command in INITTAB with an **stty** implies that DTR is always high. To control DTR, use *off* in INITTAB, open a channel with a Synergy Language OPEN statement, and then use INITPORT to set the speed, and so forth. When the channel is closed, DTR drops, and the port resets to UNIX default speed settings.

There are two ways to access a serial port on UNIX systems—one with uppercase letters (such as /dev/tty1A) and one with lowercase letters (such as /dev/tty1a). The uppercase form indicates use of CCITT modem control signal checking and time-outs.

To open a port using the uppercase letter form, use the /NODELAY OPEN OPTIONS qualifier. If you do not specify this qualifier, the OPEN fails after a time-out period.

You must also use the INITPORT subroutine to set the optional *modem* argument to 1 (modem control, which is the default) so that the “hupcl” and “clocal” **stty** flags are correctly set.

When using the lowercase form to access a serial port, set the INITPORT *modem* argument to 2 (no modem control). This implies “-hupcl clocal” **stty** settings for correct operation.



The serial communications set up by INITPORT fully support hardware and software flow control. Most machines reliably receive data at rates up to 19,200 bauds per second. In some cases, a hardware upgrade to a 16550x UART or faster machine may be necessary. However, you must use a GETS statement, not an ACCEPT statement, with a multicharacter buffer to receive the data at higher rates.

For correct operation of a serial line, including acceptance of all possible character codes, you must use the INITPORT subroutine to set up the terminal characteristics. When using the INITPORT subroutine, make sure it follows the OPEN statement *as soon as possible* in case data arrives on the serial port before the port speed is set up in the UART by INITPORT. After a CLOSE, you must once again use the INITPORT subroutine (after a subsequent OPEN).

Synergy Language uses the default settings if you don't call INITPORT when doing serial port I/O. If you call INITPORT but omit an argument, the default setting for that argument is used.

In all cases, avoid doing single-character GETS/ACCEPT sequences with the TTSTS subroutine (which slows down the system). A computer may be slowed noticeably by trying to accept characters at 9600 baud. You should always use GETS with the WAIT and MASK qualifiers for timed I/O to a serial port. In this mode, several devices at 9600 baud can be handled simultaneously.

See also

[OPEN on page 4-124.](#)

Examples

The following program is a terminal emulation program that uses the serial port interface and the INITPORT subroutine.

```
.define TTCHN      ,1
.define SRVCHN     ,2
.align
record
    ch              ,a10
    mask            ,[8]i4
    len             ,i4
    stat            ,i4
    data            ,a80

proc
    xcall flags(54020)                ;No echo, lowercase
    open(TTCHN, i, "tt:")              ;Channel 1 is console chan
    open(SRVCHN, i, "com1:")           ;Channel 2 is serial chan
    xcall initport(SRVCHN, 9600, 1, 1, 8, 1, 2)
    xcall fill(%char(0), ^a(mask[]))
    repeat
        begin
            xcall ttsts(stat, TTCHN)    ;See if operator pressed key
            while (stat)
                begin
                    accept(TTCHN, ch, exit)
                    display(SRVCHN, ch)  ;Send it to serial line
                    xcall ttsts(stat, TTCHN) ;See if more operator input
                end
            xcall ttsts(stat, SRVCHN)    ;If data is available from
            if (stat)
                begin
                    gets(SRVCHN, data, WAIT:Q_NOWAIT, MASK:mask)
                    len = %rdlen
                    if (len)              ;If data was received
                        display(TTCHN, data(1:len))
                end
            end
        end
    end
```

System-Supplied Subroutines, Functions, and Classes

INITPORT

```
exit,  
    close TTCHN  
    close SRVCHN  
end
```

%INSTR – Search for a substring within a string

position = %INSTR(*start*, *string*, *substring*[, *position2*])

or

xcall INSTR(*start*, *string*, *substring*, *position2*)

Return value

position

The starting position of the first occurrence of *substring* within *string*. **(n)**

If any of the following conditions are true, the result is 0:

- ▶ *Start* is less than or equal to 0.
- ▶ *String* is null.
- ▶ *Substring* isn't found within the search range of *string*.
- ▶ *Start* is greater than the length of *string*.

If *substring* is null but *string* is not, the result is 1. If *position2* is specified, the result is also returned in that variable.

Arguments

start

The position in the string at which the search will begin. **(n)**

string

The string in which to search for the substring. **(a)**

substring

The substring for which to search. **(a)**

position2

(optional) A variable in which to return the starting position of the first occurrence of substring. If INSTR can't find an exact match of the *substring* value or matches one of the other conditions described by *position*, *position2* is returned with a value of 0 to indicate that the search failed. **(n)**

Discussion

Searching from left to right, %INSTR searches for *substring* in *string* from *start* through the end of the string. The length of *substring* must be less than or equal to the length of *string*.

To search from right to left and find the last occurrence of the substring, use %RVSTR.

The position returned is relative to the beginning of *string*.

Examples

This function checks if the entered state abbreviation is in the western part of the United States.

```
function chk_state
    a_state ,a
record
    western ,a*, "AZ|CA|CO|ID|MT|NM|NV|OR|UT|WA|WY"

proc
    freturn %instr(1, western, a_state)
endfunction
```

In the example below, assume your data division looks like this:

```
record
    codes ,a30, "help.add.sub.mul.div"
    loc ,d3
    target ,a3, "sub"
```

The following subroutine returns a **loc** value of 7, because the first **d** appears at character position 7 in the string stored in the **codes** variable.

```
xcall instr(4, codes, "d", loc)
```

The subroutine below assigns the value 8 to **loc**, because when the search begins at the eighth character stored in **codes**, the first **d** appears at character position 8. This result differs from the preceding XCALL, because the starting position value causes the search to begin after the **d** in character position 7.

```
xcall instr(8, codes, "d", loc)
```

The following subroutine searches **codes** for the first occurrence of **sub**, beginning at character position 1. **Loc** is set to 10.

```
xcall instr(1, codes, target, loc)
```

The example below uses one field for all of the error messages, rather than a separate field or array entry for each message. This saves data space, because the field is the exact size of the error message, with no filler. The processing method uses delimiters that make it easy to add, delete, or change error messages.

```
subroutine error_msg
    err_code ,a ;Error code, including colon
.define TTY ,1 ;Terminal channel
.define FILEOF ,1 ;End-of-file error #
.define ROW ,12 ;Row # to display error message
record err_msg ;Capitalized word is err_code
    ,a*, "Opener:Could not open file\"
    & "FLESIZ:File full\"
    & "DUPKEY:Duplicate key exists\"
    & "KEYSIZ:Incorrect key size\"
```



```

&                                "NOTFOUND:Record not found\"
&                                "LOCKED:Record locked--try again\"
&                                "ILLRECNBR:Invalid record number\"
&                                "WRONG:Invalid error code\"
&                                "CNTLC:Internal error--call developer!\\"

record msg_buffer
    ret_msg      ,a22, "Press <CR> to Continue"
    input        ,a1      ;Dummy input variable
    start_pos    ,d4      ;Start position of error message
    end_pos      ,d4      ;End position of error message
    len          ,d2      ;Length of error message
record window
    width        ,d2      ;Width of error msg display
    col          ,d2      ;Column position of error msg

proc
onerror (FILEEOF)done, exit      ;Look for valid error code
xcall instr(1, err_msg, err_code, start_pos)
len = %len(err_code)
if (start_pos .eq. 0) then
    begin
        len = 6                ;Length of "WRONG:"
        xcall instr(1, err_msg, "WRONG:", start_pos)
    end
xcall instr(start_pos + len, err_msg, '\', end_pos)

decr end_pos                ;Back up to end of msg
start_pos = start_pos + len
width = end_pos - start_pos + 1;Error msg width
col = %rnd(40.0 - width/2.0)

display(TTY, $scr_pos(ROW, col),
&      err_msg(start_pos, end_pos),
&      $scr_pos(ROW + 1, 28), ret_msg)
reads(TTY, input, done)      ;Wait for user response

done,                        ;Error displayed
offerror
return

exit,                        ;If anything unexpected happens, stop program
offerror
xcall exit_sys                ;External wrap up
stop
endsubroutine

```

%INT – Return the whole number in a numeric expression

number = %INT (*expression*)

Return value

number

The whole number portion of the specified numeric expression. (**n**)

Arguments

expression

The expression whose whole number portion is to be returned. (**n**)

Discussion

%INT returns the whole number portion of a numeric expression.

%INTEGER – Convert an expression to an integer value

integer = %INTEGER (*expression* [, *magnitude*])

Return value

integer

The integer value of *expression*. (**i**)

Arguments

expression

The expression to convert to an integer value. (**a** or **n**)

magnitude

(optional) The size of the result. (**n**)

Discussion

%INTEGER converts an expression to an integer value.

Expression is converted according to the rules for moving data to an integer destination. (See [“Assignment Statements” on page 3-26](#) for a description of these rules.) If the converted expression is too long to fit into the requested integer size, the high-order bits that don’t fit are ignored. If *magnitude* is not specified or contains a value other than 1, 2, 4, or 8, the resulting string is four bytes long.

Examples

This example uses alpha input as data for an integer arithmetic computation. If the alpha variable contains any nonnumeric data other than a blank, a decimal point, or a sign (+ or –), a “Bad digit encountered” error (\$ERR_DIGIT) is generated.

```
subroutine math
  a_avar      ,a
record
  val1        ,i2
  val2        ,i2
  result      ,i4

proc
  result = (val1*val2)/%integer(a_avar)
  .
  .
  .
  return
endsubroutine
```

ISAMC – Create an ISAM file

```
xcall ISAMC (file_spec, rec_size, num_keys, key_spec[, ...])
```

Arguments

file_spec

The specification for the ISAM file you want to create and any additional options. The format for a file specification is described in the Discussion section below. **(a)**

rec_size

The maximum size of the data records. The minimum record size is 4 (or 5 for terabyte files). **(n)**

num_keys

The number of keys in the ISAM file. You can specify between 1 and 255 keys. **(n)**

key_spec

One of the following: **(a)**

- ▶ A variable that contains a single key specification. You must include a *key_spec* for each key in the ISAM file.
- ▶ A dimensioned argument of at least *num_keys* elements defining each key being created.

The format for a key specification is described in the Discussion section below.

Discussion

The ISAMC subroutine creates an ISAM file. You can use the ISAMC subroutine to create any type of Synergy Language–supported ISAM file: Synergy ISAM or RMS ISAM.

ISAMC tries to gain exclusive access to the specified file. If the file is currently in use, the ISAMC request is rejected with a “File in use by another user” error (\$ERR_FINUSE).

File specification

```
filename[, record_type][[, COMPRESS][, DENSITY=file_density][, I=pos:len[, ...]]  
&      [, NETWORK_ENCRYPT][[, PAGE = page_size][, STATIC_RFA][[, TBYTE]
```

filename

The name of the ISAM file you want to create. The default extension is **.ism**. **(a)**



To make your code more portable, we recommend you do not use a file extension on the name of the ISAM file if there's a chance you'll ever use a different ISAM file structure on another operating system, since some file structures require a specific extension.

record_type

(optional) One of the following types of ISAM files:

FIXED	The records are of fixed length. (default)
MULTIPLE	The records are of multiple fixed length.
VARIABLE	The records are of variable length.

COMPRESS

(optional) The record data will be compressed. You cannot specify this option with a record type of MULTIPLE.

DENSITY

(optional) Indicates that the file density follows.

file_density

A number between 50 and 100 that represents the default density percentage for each key in the file, which is the percentage each index block is filled. (See “[ISAM index density](#)” in the “Synergy DBMS” chapter of *Synergy Language Tools* for more information and suggestions about setting the file density.)

I

(optional) Indicates that a portable integer definition follows. You can specify this option more than once to define up to 255 portable integers per file.

pos

The starting position of nonkey integer data. The first position of the record is 1.

len

The length of nonkey integer data. The following values are valid:

- 1
- 2
- 4
- 8

WIN, UNIX

NETWORK_ENCRYPT

(optional) Sets the network encryption flag on the new file, which ensures that any client accessing that file must use encryption. (See “[Specifying the data to encrypt for slave encryption](#)” in the “Configuring xfServer” chapter of the *Installation Configuration Guide* for more information.)

PAGE

(optional) Indicates that the page size follows.

page_size

The size of each index block in the ISAM file. The following values are valid:

512

1024 (default)

2048

4096

8192

(See [“Page size”](#) in the “Synergy DBMS” chapter of *Synergy Language Tools* for more details and suggestions about setting the page size.)

STATIC_RFA

(optional) The records will have constant RFAs over the life of the specified ISAM file if RFAs are to be used on this file.

TBYTE

(optional) Create a file capable of holding one terabyte (1024 GB) of data or index.

WIN

Terabyte files are only supported on Windows systems with the NTFS file system. They are not supported on FAT or FAT32 file systems.

UNIX

Terabyte files are not supported on SCO OpenServer. Some operating systems require you to set a large-file option on the file system being used. Some (AIX in particular) require this to be done when the file system is first created, and others allow the option to be added later.

VMS

The PAGE and TBYTE options are ignored. The I and STATIC_RFA options cause an “Invalid option” error (\$ERR_INVOPT). OpenVMS files operate as if STATIC_RFA was specified.

If the specified file already exists, the ISAMC subroutine replaces it by default. To change this default, set the FLAGS subroutine’s flag 3. If flag 3 is set and ISAMC attempts to create a file that already exists, a “Cannot supersede existing file” error (\$ERR_REPLAC) is generated.

You can abbreviate all file options for the ISAMC subroutine. However, for program clarity and to eliminate confusion with options that may be added in the future, we recommend that you specify the full option name.

Key specification

If *key_spec* is a single key specification or a dimensioned argument, the number of *key_spec* arguments or dimensions should be equal to or greater than the number specified by *num_keys*. A key specification in *key_spec* should have the following form:

```
START=pos_1[:pos_n], LENGTH=len_1[:len_n][, NAME=key_name]
&      [, TYPE=key_type_1[:key_type_n]][, ORDER=key_order_1[:key_order_n]]
&      [, DENSITY=key_density][, [NO]DUPS][, [NO]MODIFY][, [NO]ASCEND]
&      [, [NO]ATEND][, COMPRESS=[NO]INDEX | [NO]RECORD | [NO]KEY | ALL]]
&      [, NULL=null_type[, VALUE_NULL=null_val]]
```

START

Indicates the position where the key begins follows. You can specify up to eight positions, separated by colons.

pos_1

The starting position of the key (or the first segment of the key).

pos_n

(optional) The position of additional key segments, if any exist.

LENGTH

Indicates the length of the key follows. You can specify up to eight lengths, separated by colons. The number of segment lengths must equal the number of start positions.

len_1

The length of the key or the first segment of the key. The key (all segments combined) can be up to 254 characters long on Windows and UNIX (251 if the key allows duplicates or 250 if the key allows duplicates and this is a terabyte file) or 255 characters on OpenVMS.

len_n

(optional) The length of additional key segments, if any exist.

NAME

(optional) Indicates the named key of reference for Synergy ISAM access follows.

key_name

The name of the key. If the name contains spaces, the key name must be enclosed in quotation marks. The maximum key name length is 15 for Synergy ISAM. RMS ISAM has no maximum length, but ISAMC only accepts a maximum of 32 characters.

TYPE

(optional) Indicates the type of key or segment follows. You can define one type for the key and all of its segments, or you can define a type for each segment. If you define the segments individually, you must include a type for every segment.

key_type_1

One of the following values that defines the type of the key (or the type of the first segment):

ALPHA (default)

NOCASE

DECIMAL

INTEGER

UNSIGNED

VMS

RMS supports alpha keys, unsigned integer keys (**i1**, **i2**, **i4**, and **i8**), and signed integer keys (**i2**, **i4**, and **i8**).

key_type_n

A value listed under *key_type_1* above that defines the type of any additional segments.

ORDER

(optional) Indicates the key order follows.

key_order_1

One of the following values that defines the order of the key (or the order of the first segment):

DESCENDING (or **D**)

ASCENDING (or **A**)

You can define one key order for the key and all of its segments, or you can define a key order for each segment. If you define the segments individually, you must include a key order for every segment.

This value overrides the [NO]ASCEND option.

key_order_n

One of the values listed under *key_order_1* that defines the order of any additional segments.

VMS

The ORDER option is ignored. RMS does not support key segments of varied order. Use the [NO]ASCEND option to specify the key order instead.

DENSITY

(optional) Indicates the key density follows. This value overrides the file density for each key for which it is specified.

key_density

A number between 50 and 100 that represents the density percentage for that key. (See “[Key density](#)” in the “Synergy DBMS” chapter of *Synergy Language Tools* for more information and suggestions about setting the key density.)

/NO/DUPS

(optional) Specifies whether duplicate keys are allowed. The default is no duplicates (NODUPS).

/NO/MODIFY

(optional) Specifies whether the key is modifiable. The default is not modifiable (NOMODIFY). The primary key cannot be modified.

/NO/ASCEND

(optional) Specifies whether the key data will be sorted in ascending or descending order. The default is ascending order (ASCEND). This option is overridden by the ORDER option if it is specified.

/NO/ATEND

(optional) Specifies whether records that contain duplicate keys will appear at the end or at the beginning of a list of matching records. The default is at the end (ATEND).

VMS

Records that contain duplicate keys are always inserted at the end of a list of matching records, which means that the ATEND option is ignored. If you specify NOATEND on RMS ISAM, an “Invalid option” error (\$ERR_INVOPT) is generated.

COMPRESS

(optional) Specifies that the ISAM key will be compressed. This key compression option differs from the data compression option in the filename specification, discussed above. Only RMS ISAM uses this option. The default option is ALL.

You can specify the following options:

/NO/INDEX	Compresses the index. The default is INDEX.
/NO/KEY	Compresses the key within the data. The default is KEY.
/NO/RECORD	Compresses the record within the data. The default is RECORD.
ALL	Compresses all of the applicable options above.

NULL

(optional) Specifies that the key is a null key. In this context, a null key means that when a record is stored, and this key has a null value, no entry is placed in the index, thus saving file space, I/O, and processing. By default, a key is assumed to be null if it is filled with spaces. You can change this assumption using the VALUE_NULL option. The default null value is blank for alpha keys and 0 for numeric keys. You can specify null keys on alternate keys only, not on a primary key.

null_type

Can be one of the following types on all operating systems except OpenVMS. See “Null keys” in the “Synergy DBMS” chapter of *Synergy Language Tools* for more information.

- REPLICATE

Specifies that *null_val* is a single character and must match each byte of the specified key.
- NOREPLICATE

If the key is alpha, specifies that *null_val* is a string that must match the key, from the beginning of the key, for the length of the string. If the key is numeric, specifies that *null_val* must match the numeric key value.
- SHORT

Specifies that the key won’t be stored if the record doesn’t include the entire key on a STORE or WRITE. The file must have variable-length records.

VMS

The *null_type* argument is optional, and, if specified, may only be REPLICATE.

VALUE_NULL

(optional) Indicates the null value for a null key follows.

null_val

The null value, which can be specified as either a single character or a string. The default null value is a space. Enclosing the value in quotation marks is optional. For example:

"chr"	:	"0", " ", "\0"	ASCII representation
val	:	48, 32, 0	Binary representation
string	:	abc, "abc"	Nonreplicating string
numeric	:	100, "100"	Nonreplicating numeric string

VMS

Null keys are only allowed on alpha keys and *null_val* can only specify a single character. When every byte of the key matches this character, the key is considered to be null, and no index entry is created. If a string is specified as *null_val*, the first character of the string is considered to be the null value character.

If conflicting qualifiers and settings are specified, the last one takes precedence.

See also

- ▶ [“Synergy DBMS”](#) chapter of *Synergy Language Tools*.
- ▶ Your RMS ISAM documentation.

Examples

The example below creates a compressed ISAM file named **cusmas.ism** with variable-length records that have a maximum record size of 2000. It specifies the following four keys:

- ▶ The primary key is named “name” and has two segments, which start at positions 1 and 16. The length of each segment is 15. This key does not allow duplicates, and it is not modifiable. Data for this key is sorted in descending order.
- ▶ The first alternate key is named “company” and has one segment, which starts at position 31 and is 30 characters long. This key allows duplicates, and it is not modifiable. Key data is sorted in descending order, and records that contain duplicates appear at the beginning of a list of matching records.
- ▶ The second alternate key is named “address” and has three segments, which start at positions 61, 81, and 91. The length of the first segment is 20, and the length of the second and third segments is 10. This key allows duplicates, and it is modifiable. Key data is sorted in ascending order, and records that contain duplicates appear at the end of a list of matching records.
- ▶ The third alternate key is named “acct code” and has one segment, which starts at position 101 and is five characters long. (Note that the key name contains a space and therefore must be quoted. Remember that two quotation marks in a string place one quotation mark in the string.) This key allows duplicates, and it is not modifiable. Key data is sorted in ascending order, and records that contain duplicates appear at the beginning of a list of matching records. This key is also a replicate null key with a null value of 32. When all five bytes of this key are blank, no key entry will exist for that record in the index. We could easily have left out the **value_null** spec and accomplished the same result.

```
keyspec1="start=1:16, len=15:15, name=name"
keyspec2="start=31, len=30, name=company, dups, noascend"
keyspec3="start=61:81:91, len=20:10:10, name=address, dups, modify,
&          atend"
```

```
keyspec4="start=101, len=5, name=""acct code"", dups, null=replicate,  
    &      value_null=32"  
  
xcall isamc("cusmas, variable, compress", 2000, 4, keyspec1, keyspec2,  
    &      keyspec3, keyspec4)
```

The example below creates an ISAM file named **temp.ism** that has a record size of 20. It has one key that starts at position 1 and is five characters long. This file is compressed, allows variable-length records, and uses static RFAs. (This example illustrates how you can abbreviate all file options for the ISAMC subroutine.)

```
xcall isamc("temp, c, v, s", 20, 1, "start=1, len=5")
```

The example below shows *key_spec* as a dimensioned argument.

```
record  
    keyspecs , [3]a25      , "start=1, length=5, dups"  
    &              , "start=10, length=5, dups"  
    &              , "start=20, length=5, dups"  
    .  
    .  
    .  
    xcall isamc("file1", 128, 3, keyspecs)
```

ISCLR – Clear an ISAM file

```
xcall ISCLR(filename, status)
```

Arguments

filename

An expression that names the ISAM file you want to clear. The alpha value is a standard file specification. The default filename extension is **.ism**. (**a**)

status

Currently required but ignored. (**n**)

Discussion

The ISCLR subroutine empties an ISAM file. It is extremely useful in programs that temporarily post information stored in ISAM transaction files. After the information is posted to another file, you can use ISCLR to clear the transactions.

ISCLR tries to gain exclusive access to the specified file. If the file is currently in use, the ISCLR request is rejected with a “File in use by another user” error (\$ERR_FINUSE).



Use the ISCLR subroutine with caution so you don't accidentally clear your ISAM files; clearing an ISAM file deletes all existing records.

In previous versions of Synergy Language, you could also specify a channel argument on the ISCLR subroutine. This syntax is still valid, but the channel argument is ignored. ISCLR no longer closes the channel if the channel is open.

See also

[OPEN](#) on page 4-124.

Examples

The following example prompts the user for the name of an ISAM file to clear and uses the ISCLR subroutine to clear the specified file.

```
.define TTCHN ,1
record
    errname ,a40
    status ,d1
    name ,a40

proc
    xcall flags(7004020, 1)
    open(TTCHN, i, "tt:")
    display(TTCHN, "Enter name of ISAM file to clear: ")
    reads(TTCHN, name)
    onerror clrerr
    xcall isclr(name, status)
    offerror
    writes(TTCHN, "Cleared successfully")
    stop

clrerr,
    display(TTCHN, "Operation failed, error was: ")
    writes(TTCHN, %string(%ernum))
    stop
end
```

If the **isclr** operation fails, the program displays the following message:

Operation failed, error was: *nn*

where *nn* indicates the runtime error number.

Otherwise, the program displays the message

Cleared successfully

%ISINFO – Return numeric ISAM file status and key information

info = %ISINFO(*channel*, *request*[, *key*][, *segment*])

or

xcall ISINFO(*info*, *channel*, *request*[, *key*][, *segment*])

Return value

info

The information that is specified by the *request* keyword. (n)

Arguments

channel

The channel number. (n)

request

One of the following keywords that specifies what information is to be returned. (See [page 9-183](#) for OpenVMS-specific information.) (a)

"SIZE"	The record size. (n)
"NUMRECS"	The number of records in the file. (n)
"NUMKEYS"	The number of keys for the file. (n)
"KPOSITION"	The position of the start of the key or segment. (n)
"KEYLENGTH"	The length of the key, including all segments. If the <i>segment</i> argument is specified, this value automatically becomes a SEGLENGTH value rather than a KEYLENGTH value. (n)
"KEYSEGMENTS"	The number of key segments. (n)
"KEYDENSITY"	The value of the index density for a key. (n)
"DUPS"	One of the following values: (n)
1	Duplicates
0	No duplicates
"MODIFY"	One of the following values: (n)
1	Modifiable
0	Not modifiable

"ASCEND"	One of the following values: (n) 1 Ascending 0 Descending If a key is segmented, ASCEND returns the value of the specified segment. (n)
"ATEND"	One of the following values: (n) 1 Duplicates at end 0 Duplicates at front The length of the key segment. (n)
"REVISION"	The ISAM revision number of the file (2, 3, or 4). (See "ISAM file types" in the "Synergy DBMS" chapter of <i>Synergy Language Tools</i> for more information.) (n)
"PAGESIZE"	The page size for the file. (n)
"KEYDEPTH"	The depth of the specified key's index. (n)
"NETCRYPT"	One of the following values: (n) 1 File enabled for network encryption 0 File not enabled for network encryption

key

(optional) The key the information is about: (n)

0	Primary key (default)
1	First alternate key
2	Second alternate key

etc.

segment

(optional) The segment the information is about: (n)

1	First segment (default)
2	Second segment

etc.

Discussion

If *request* is SIZE, NUMRECS, NUMKEYS, REVISION, or PAGESIZE, both the *key* and *segment* arguments are ignored.

If *request* is KEYSEGMENTS, KEYDENSITY, DUPS, MODIFY, ATEND, or KEYDEPTH, the *segment* argument is ignored.

VMS

The following values are returned on OpenVMS:

Keyword	Returned value
NUMRECS	"09999999"
KEYDENSITY	The percentage that corresponds to the FDL attribute KEY INDEX_FILL
ATEND	1
REVISION	The PROLOG version of the file
PAGESIZE	0
KEYDEPTH	0

Only the supported ISAM features are returned.

The following error conditions are possible:

- ▶ If a key or segment is invalid, an "Illegal key specified" error (\$ERR_BADKEY) is generated.
- ▶ If the *request* option is invalid, an "Invalid option" error (\$ERR_INVOPT) is generated.

See also

[%ISINFOA on page 9-185.](#)

Examples

```
.define RPTCHN,1
.define ISMCHN,2

record
    nkeys      ,i4
    nsegs      ,i1
    wopt       ,a15
    keynum     ,i4
    segnum     ,i1
```

System-Supplied Subroutines, Functions, and Classes

%ISINFO

```
len      ,i4
type     ,a1
i        ,i4
j        ,i1

proc
  open(RPTCHN, o:s, "ismrpt.txt")
  open(ISMCHN, i:i, "myfile.ism")
  wopt = "NUMKEYS"
  xcall isinfo(nkeys, ISMCHN, wopt)
  for i from 0 thru (nkeys - 1)
    begin
      nsecs = %isinfo(ISMCHN, "KEYSEGMENTS", i)
      for j from 1 thru (nsecs)
        begin
          xcall isinfo(len, ISMCHN, "SEGLLENGTH", i, j)
          xcall isinfoa(type, ISMCHN, "KTYPE", i, j)
          writes(RPTCHN, "Key:" + %string(i) + " Seg:" + %string(j) +
&          " Length=" + %string(len) + ", Type=" + type)
        end
      end
    close(RPTCHN)
    close(ISMCHN)
  end
```

%ISINFOA – Return alpha ISAM file status and key information

```
info = %ISINFOA(channel, request[, key][, segment])
```

or

```
xcall ISINFOA(info, channel, request[, key][, segment])
```

Return value

info

The information that is specified by the *request* keyword. (a)

Arguments

channel

The channel number. (n)

request

One of the following keywords that specifies what information is to be returned. (See [page 9-183](#) for OpenVMS-specific information.) (a)

"FILENAME" The filename. (a)

"FOPTS" A file option string containing one or more of the following, separated by commas: (a)

"COMPRESS" Data is compressed.

"FIXED" Records are of fixed length.

"MULTIPLE" Records are of multiple fixed length.

"VARIABLE" Records are of variable length.

"PAGE=*size*" Page size is *size*.

"STATIC_RFA" Records have constant RFAs.

"TBYTE" File is 40-bit.

"KTYPE" One of the following key types:

"A" Alpha

"N" Nocase

"D" Decimal

"I" Integer

"U" Unsigned

If no segment is specified, KTYPE returns the type of the first segment (which is the type of the key if the key is not segmented). (a)

"NULL"

One of the following values: (a)

"SHORT"

"REPLICATE"

"NOREPLICATE"

"NONE"

"NULLVALUE"

A string containing the null value as defined in the VALUE_NULL ISAMC option. If there is no null value for the key, an error is generated. (a)

"KEYNAME"

The key name. (a)

"STRING"

A string that contains the key information in ISAMC routine format. (a)

"CREATEDATE"

The date the file was created. If no date is available, the string "NODATE" is returned. (a)

"CLEARDATE"

The date the file was last cleared (using **isclr**). If no date is available, the string "NODATE" is returned. (a)

"REINDEXDATE"

The date the file was last reindexed (using **isutl**). If no date is available, the string "NODATE" is returned. (a)

"VERIFYDATE"

The date the file was last verified (using **isutl**). If no date is available, the string "NODATE" is returned. (a)

key

(optional) The key the information is about: (n)

0

Primary key (default)

1

First alternate key

2

Second alternate key

etc.

segment

(optional) The segment the information is about: (n)

1

First segment (default)

2

Second segment

etc.

Discussion

If *request* is FILENAME, FOPTS, CREATEDATE, CLEARDATE, REINDEXDATE, or VERIFYDATE, both the *key* and *segment* arguments are ignored.

If *request* is NULL, NULLVALUE, KEYNAME, or STRING, the *segment* argument is ignored.

Data is moved into *info* according to the rules for moving alpha data. No error is generated on data overflow.

VMS _____

The following values are returned on OpenVMS:

Keyword	Returned value
FOPTS	A file option string that contains one or more of the following values: "COMPRESS" "FIXED" "VARIABLE"
KTYPE	One of the following key types: "U" Unsigned "I" Integer "C" Collating "P" Packed "A" Alpha (string) The type is associated with the key and not the segment.
NULL	One of the following values: "REPLICATE" "NONE"
CLEARDATE, REINDEXDATE, VERIFYDATE	"NODATE"

Only the supported ISAM features are returned.

The following error conditions are possible:

- ▶ If a key or segment is invalid, an “Illegal key specified” error (\$ERR_BADKEY) is generated.
- ▶ If the *request* option is invalid, an “Invalid option” error (\$ERR_INVOPT) is generated.
- ▶ If no null string is specified for the key, or if a null string could not be found using the NULLVALUE or STRING option, an “” error (\$ERR_ISINFO) is generated.

See also

[%ISINFO on page 9-181.](#)

Examples

```
a_value = %isinfoa(ismchn, "KTYPE", key0, seg1)
```

For a more complete example, see [page 9-183](#).

ISKEY – Retrieve ISAM key information

`xcall ISKEY(channel, keynum, status)`

Arguments

channel

The channel on which the ISAM file is open. **(n)**

keynum

The number of the key about which information is to be returned. The primary key is 0, the first alternate key is 1, the second alternate key is 2, and so forth. If the key doesn't exist, an "Illegal key specified" error (\$ERR_BADKEY) occurs. **(n)**

status

The variable that will be loaded with status information about the key. The format of this information is shown in the table in the Discussion section below. **(a)**

Discussion

The ISKEY subroutine returns information about a specified key from the ISAM file that's currently open on the specified channel. (See [%ISINFO on page 9-181](#) for additional status information.)

If the *status* variable is shorter than 90 characters, only the leftmost portion of the information is transferred. If *status* is longer than 90 characters, only its leftmost characters are filled and the remaining character positions are filled with blanks. *Status* is filled according to the rules for moving alpha data to an alpha destination. (See ["Assignment Statements" on page 3-26](#) for a description of these rules.)

The following error conditions are possible:

- ▶ If the channel is not open, a "Channel has not been opened" error (\$ERR_NOOPEN) is generated.
- ▶ If the channel is open but is associated with a non-ISAM file, a "Not an ISAM file" error (\$ERR_NOTISM) is generated.

The status information is returned as follows:

Character position	Data type	Information
1 – 5	d5	Position of key
6 – 8	d3	Length of key
9	d1	Duplicates allowed flag (1 = duplicates, 0 = no duplicates)
10	d1	Ascending or descending flag (1 = ascending, 0 = descending)
11	d1	Modifiable flag (1 = modifiable, 0 = nonmodifiable)
12 – 26	a15	Name of key
27 – 66	8d5	Positions of segments
67 – 90	8d3	Lengths of segments

The first segment length of *status* is always nonzero. If the second segment length is nonzero, the key is segmented, and all remaining nonzero lengths represent the segmented keys.

Examples

The following example uses the ISKEY subroutine to retrieve information about the second alternate key in an ISAM file named **demo.ism**. The program then displays that information to the screen.

```
.define TTCHN          ,1
.define ISMCHN         ,2
.define MAX_KEYS       ,8
record keyinfo
    pos      ,d5
    len      ,d3
    dup      ,d1
    asc      ,d1
    mod      ,d1
    nam      ,a15
    spos     ,MAX_KEYS d5
    slen     ,MAX_KEYS d3
record
    seg      ,d1
```



```

proc
  open(TTCHN, o, "tt:")
  open(ISMCHN, i:i, "demo.ism")

;Request information about the second alternate key

  xcall iskey(ISMCHN, 2, keyinfo)
  writes(TTCHN, "Information on the key " + nam + ':')
  writes(TTCHN, nam + " start at char: " +
&      %string(pos))
  writes(TTCHN, nam + "'s length is:" + %string(len))
  if (dup.eq.1) then
    writes(TTCHN, "Duplicates are allowed")
  else
    writes(TTCHN, "No duplicates are allowed")
  if (asc.eq.1) then
    writes(TTCHN, nam + "is in ascending order")
  else
    writes(TTCHN, nam + "is in descending order")
  if (mod.eq.1) then
    writes(TTCHN, nam + "is modifiable")
  else
    writes(TTCHN, nam + "is not modifiable")
  for seg from 2 thru 8
    if (spos(seg).ne.0)
      begin
        writes(TTCHN, "Segment #" + %string(seg) +
&          " starts at " + %string(slen(seg)) +
&          " and is " + %string(slen(seg)) + " long")
      end
  stop
end

```

ISSTS – Return status information about an ISAM file

`xcall ISSTS(channel, status)`

Arguments

channel

The channel on which the ISAM file is open. (**n**)

status

The variable that will be loaded with status information about the file. The format of this information is shown in the table in the Discussion section below. (**n**)

Discussion

The ISSTS subroutine returns status information about the ISAM file that is currently open on the specified channel. (See [%ISINFO on page 9-181](#) for additional status information.)

The channel must be open to an ISAM file.

The following error conditions are possible:

- ▶ If the channel is not open, a “Channel has not been opened” error (\$ERR_NOOPEN) is generated.
- ▶ If the channel is open but is associated with a non-ISAM file, a “Not an ISAM file” error (\$ERR_NOTISM) is generated.
- ▶ If the number of keys exceeds 99, an “Illegal key specified” error (\$ERR_BADKEY) is generated. You can trap this error, in which case the number of keys is set to 0. Use the [%ISINFO](#) function to access the correct number of keys.

Although 31 characters of status information can be returned, *status* can be as long as you want. If *status* is shorter than 31 characters, ISSTS fills the buffer with as much information as will fit. (For example, if *status* is 20 characters, ISSTS returns the first 20 characters of status information.) If *status* is longer than 31 characters, only its leftmost characters are filled. *Status* is filled according to the rules for moving alpha data to an alpha destination. (See [“Moving alpha data to an alpha destination” on page 3-26](#) for a description of these rules.)

The status information is returned as follows:

Character position	Data type	Information
1 – 5	d5	Maximum length of data records
6 – 10	d5	Position at which the primary key field starts
11 – 13	d3	Length of primary key field

Character position	Data type	Information
14 – 21	d8	Approximate capacity of files (in records)
22 – 29	d8	Number of unused records
30 – 31	d2	Total number of primary and alternate keys

Positions 14 through 21 are filled with nines (“99999999”).

If the total number of records meets or exceeds 99,999,999, the number of unused records (positions 22–29) returns 0. Use the %ISINFO function to access the correct number of records.

VMS

Positions 22 through 29 are filled with “09999999”.



With Synergy ISAM, you can calculate the number of records by subtracting the number of unused records from the approximate capacity, as shown in the example below. On OpenVMS, the number of records actually contained in an RMS indexed file cannot be determined without reading the entire file and counting the records.

Examples

The example below uses the ISSTS subroutine to retrieve the current status of an ISAM file called **demo.ism** and then displays that information to the screen.

```
.define TTCHN      ,1
.define ISMCHN     ,2
record status
    reclng         ,d5
    keystart       ,d5
    keylng         ,d3
    recsmax        ,d8
    recsleft       ,d8
    numkeys        ,d2
record
    recs           ,d8

proc
    open(TTCHN, i, "tt:")
    open(ISMCHN, i:i, "demo.ism")
    xcall issts(ISMCHN, status)
    writes(TTCHN, "Data record length:  " + %string(reclng))
    writes(TTCHN, "Key starts at char:  " + %string(keystart))
```

```
        writes(TTCHN, "Key length is:          " + %string(keylng))
        recs = recsmax - recsleft
        writes(TTCHN, "Total records in file:" + %string(recs))
        writes(TTCHN, "Number of keys:         " + %string(numkeys))
        stop
    end
```

JBNAM – Provide the image name of the calling program

```
xcall JBNAM(image)
```

Arguments

image

Returned with the image name. It should be large enough to hold 63 characters. (a)

Discussion

The JBNAM subroutine retrieves the image name of the calling program.

WIN, UNIX

JBNAM returns the module name of the main module.

VMS

JBNAM performs a \$GETJPI(JPI\$_IMAGENAME) and returns the result.

Examples

```
record
    image      ,a63
    chan       ,i4 ,2

proc
    open(chan, o, "TT:")
    xcall jbnam(image)
    writes(chan, %atrim(image))
    close chan
    stop
end
```

%JBNO – Return the job number for the current process

```
id = %JBNO ([parent_id][, group_id])  
or  
xcall JBNO (id, [parent_id][, group_id])
```

Return value

id

The job or process identification number for the current job or process. **(n)**

Arguments

parent_id

(optional) The variable to be loaded with the current parent process identification number. **(n)**

WIN

group_id

(optional) The variable to be loaded with the network adapter address on Windows if a network adapter is present. **(a)**

UNIX, VMS

group_id

(optional) The variable to be loaded with the current process group identification number. **(n)**

Discussion

%JBNO is evaluated at runtime. You can use %JBNO to create a unique name for a temporary file.

WIN

Id is the low-order 20 bits of the process ID number.

When Windows Terminal Services are enabled, *parent_id* is the WTS session ID, which is unique between WTS sessions; otherwise, *parent_id* is 0.

Group_id is obtained from the GetAdaptersInfo function.



The *group_id* obtained from a Windows Terminal Services client reflects the server's network address and thus is identical to all other WTS clients.

UNIX

The current process group ID number remains constant throughout a log-in session unless a process that creates a new process group ID is executed. On some systems, **ksh** generates a new process group ID.

VMS

Parent_id is loaded with the process ID of the process that created the current process. *Group_id* is loaded with the group number of the current process's UIC.

%JBNO can also be called as a subroutine, with the syntax

```
xcall JBNO ([id], [parent_id][, group_id])
```

The JBNO subroutine requires at least one argument, but you can specify any or all of the three arguments. *Id* should be an **i4** or **d8** variable on all platforms to prevent overflow. *Id* is unique for each program on a machine but is not guaranteed to be unique on different machines on the network. For a unique number for a work file on a network, we suggest using a combination of *id* and *group_id*.

Examples

This example creates a temporary file using **id** as the file extension and **time** as the filename.

```
subroutine tmpfile
    a_name          ,a
record
    time            ,d6
    id              ,i4

proc
    xcall time(time)
    xcall jbno(id)
    a_name = %string(time) + '.' + %string(id)
    return
endsubroutine
```

%JPERIOD – Return the Julian period value for a date

period = %JPERIOD([*date*])

Return value

period

The Julian period value for the specified date. (**n**)

Arguments

date

(optional) The date for which you want to determine the Julian period date. The default is the current date. (**a** or **n**)

Discussion

%JPERIOD validates a date and (if the date is valid) returns a numeric value that uniquely identifies the passed date. This value is the Julian period value, or the number of days since December 31, 4715 BC. Consecutive dates have consecutive Julian period values. Dates prior to December 31, 4715 BC are not supported.

The dates Wednesday, September 2, 1752, and Thursday, September 14, 1752, have consecutive Julian period values, because the changeover from the Julian to the Gregorian calendar occurred at this time, and September 2 (Julian) was followed by September 14 (Gregorian). Leap years are fully accounted for according to the Gregorian calendar.



Synergy provides no conversions between Julian and Gregorian calendar dates.

If *date* is passed as a numeric value, the format must be YYYYMMDD. Some examples are

19991013 October 13, 1999

18120101 January 1, 1812

If *date* is passed as an alpha value, the format can be

[DD][-/][MMM][-/][YYYY][BC]

or

+|-*n*

The hyphen delimiters in the first format are optional, but they must be specified if leading components are omitted. Any omitted component defaults to the current value. For example, “-JUN” refers to today’s date, but in June. If the year is present, it must be fully specified with the century value; it does not default to the current century. For example, “17-JUN-62” occurred when Nero was emperor in Rome.

The second format indicates plus or minus n days from the current date.

Sample alpha date formats include

“17JUN1962”	June 17, 1962
“25-DEC”	December 25 of this year
“01”	The first day of the current month in the current year
“--1999”	Today’s date in 1999
“--200BC”	Today’s date in the year 200 BC
+365	The date that is 365 days after the current date
-30	The date that is 30 days before the current date

%JPERIOD is useful for validating keyboard input in a flexible way, giving keyboard operators freedom in date input formats. If an invalid date is passed, an “Invalid date” error (\$ERR_INVDATE) is generated.

%JPERIOD also makes it easier to calculate dates. For example, you can find the number of days elapsed between two dates by subtracting the Julian period value of the earlier date from that of the later date, or you can determine the date that is n days from a given date by adding n to the date’s Julian period value to get the Julian period value of the desired date.



%JPERIOD is not intended to provide a means of storing dates; it is merely a means of validating and comparing dates.

To convert back to alphabetic format, use %DATE. To convert back to numeric format, use %NDATE.

%JPERIOD can also be called as a subroutine.

Examples

The example below checks for expiration of a license period, calculates both elapsed time and future dates, and converts to and from Julian period values.

```

subroutine license_check

record
    jper1                ,i4
    jper2                ,i4
    remain               ,i4                ;Days remaining in license
    license_dur          ,i2, 14           ;License length in days
    license_start        ,d8, 19950317     ;License start date
                                           ; (March 17)
    todays_date          ,a11              ;Today's date
    expire_date          ,a11              ;Expiration date

proc
    jper1 = %jperiod(license_start)
    ;Returns the Julian period value for license start period
    ; Use (hardcoded!) numeric date as an argument

    jper1 = jper1 + license_dur
    ;Calculate Julian period of the expiration date

    expire_date = %date(jper1)
    ;Find out date string of expiration date

    todays_date = %date
    ;Called with no arguments, returns today's date as an alpha string

    jper2 = %jperiod(todays_date)
    ;Returns the Julian period value for today
    ;Use alphabetic date as an argument

    remain = jper1 - jper2
    ;How many remaining days has license left?

    xcall warn("Warning: license expires on " +
&          expire_date + ", only " +
&          %string(remain) + " days left")
    ;Use the things we've created to warn of impending DOOM!

    xreturn
endsubroutine

```

%KEYVAL – Extract the value of an index file’s key from a record

contents = %KEYVAL(*channel*, *record*, *keynum*[, *result*])

Return value

contents

The contents of the referenced key, concatenated in key segment order. **(a)**

Arguments

channel

The number of the channel opened to an ISAM file on which the record is being processed. **(n)**

record

The record from which the key value will be extracted. **(a)**

keynum

The number of a key for the opened file. **(n)**

result

(optional) The variable that will be loaded with the contents of the referenced key. **(a)**

Discussion

%KEYVAL returns the value of an index file’s key from an existing record matching the layout in the file specified by channel. No file I/O is actually performed.

This function enables you to get an entire segmented key. If *result* is present, the return value is also returned in *result*.

You can use %KEYVAL even if your key isn’t segmented.

Examples

The following subroutine creates a temporary tag file to process data from an ISAM file. It reads through the file searching for records with a nonblank entry in a nonkey field. It then creates a temporary tag file and saves the RFA position of the ISAM record with the nonblank field. It sorts the temporary file using the alternate segmented key as the sort key, then extracts the appropriate records from the ISAM file in sorted order to pass to a print subroutine.

```
subroutine report
  .define ISMCHN      ,10
  .define TAGCHN      ,11
  .define TAGFILE     , "tagfile"
  record ism_rec
    key0              ,a10
    key1a             ,a2
```

System-Supplied Subroutines, Functions, and Classes

%KEYVAL

```
ism_fld      ,a5
key1b        ,d4
record tag_rec
  tag_rfa_1    ,a8
  tag_rfa_2    ,a4
  tag_sort     ,a6
record
  group rfa    ,a
  part1        ,a4
  part2        ,a2
endgroup

proc
  open(ISMCHN, i:i, "isam")
  open(TAGCHN, o, TAGFILE)
  repeat
    begin
      reads(ISMCHN, ism_rec, done1, getrfa:rfa) ;Read sequentially
                                              ; by primary key

      if (ism_fld)
        begin
          tag_sort=%keyval(ISMCHN, ism_rec, 1)
          tag_rfa_1=%hex(rfa.part1, 4)
          tag_rfa_2=%hex(rfa.part2, 2)
          writes(TAGCHN, tag_rec)
        end
      end
    done1,
      close TAGCHN
      sort(input=TAGFILE, key=tag_sort, record=tag_rec)
      open(TAGCHN, i, TAGFILE)
      repeat
        begin
          reads(TAGCHN, tag_rec, done2)
          rfa.part1=%x(tag_rfa_1)
          rfa.part2=%x(tag_rfa_2)
          read(ISMCHN, ism_rec, match:Q_RFA, rfa:rfa)
          xcall print_it(ism_rec)
        end
      done2,
        close TAGCHN
        close ISMCHN
        xreturn
      endsubroutine
```

KILL – Terminate the calling program

```
xcall KILL
```

Discussion

The KILL routine terminates the current process. KILL works differently on each operating system.

WIN

The currently executing application is stopped as if a STOP statement were executed.

UNIX

The KILL subroutine terminates the calling process and any other processes in its process group with a HANGUP signal. The process group includes all processes spawned by the original process started from the command line. The KILL subroutine is equivalent to executing a STOP statement. In the calling program, all other programs in the process group will be terminated at the point they are currently executing, with a signal trap.

VMS

The KILL subroutine terminates the calling process by making a call to the \$DELPRC system service. The KILL subroutine is equivalent to executing a STOP statement followed by the LOGOUT command. If the calling Synergy Language program is running as a subprocess (created by the RUNJB or VMCMD subroutines), that subprocess is terminated and control returns to the parent process.

See also

[STOP on page 4-175.](#)

Examples

The example below can be called to kill the program after an invalid password is entered.

```
subroutine badpasswd
.define TTCHN,1

proc
    open(TTCHN, o, "tt:")
    writes(TTCHN, "Invalid password, logging off")
    xcall cleanup    ;Prepare app. to stop
    xcall kill
endsubroutine
```

%LEN – Return the length of the data area

length = %LEN(*variable*)

or

xcall LEN(*length*, *variable*)

Return value

length

The variable that will be loaded with the length of the specified variable. If the variable is a null argument, a length of -1 is returned. (**n**)

Arguments

variable

A variable whose length will be returned. (**a** or **n**)

Discussion

%LEN returns the length of the data area referenced by the specified variable.

%LEN is functionally identical to the ^SIZE data reference operation. %LEN and LEN are only available for compatibility with older versions of Synergy Language; we highly recommend that you use ^SIZE, which is more efficient than %LEN or LEN.

See also

[^SIZE on page 9-310.](#)

%LINE – Return the current line number

```
line_num = %LINE
```

Return value

```
line_num
```

%LINE returns the line number of the current statement. (i)

Discussion

%LINE is useful if you're using internal trace features to indicate which statement is being executed. Such tracing techniques are often helpful during the initial debugging of a complicated program.

See also

[MODNAME](#) on page 9-218.

Examples

The following debugging routine clears the bottom line of the screen and displays the line number and routine name of the program's current execution position.

```
subroutine show_line
record
    name      ,a10
    line_n    ,d4

proc
    xcall modname(0, name)
    line_n = %line
    display(TTCHN, $scr_pos(24,1), $scr_clr(line),
    &          "At line ", ^a(line_n), "in routine ", name)
    forms(TTCHN, 1)
    xreturn
endsubroutine
```

LOCALIZE – Localize currency

```
xcall LOCALIZE ([symbol_source], [sep_source], [dec_pt_source], [symbol_dest],  
&               [sep_dest], [dec_pt_dest][, case])
```

Arguments

symbol_source

(optional) The currency symbol used in the format string. **(a)**

sep_source

(optional) The separator character used in the format string. **(a)**

dec_pt_source

(optional) The decimal point character used in the format string. **(a)**

symbol_dest

(optional) The currency symbol placed in the formatted output. **(a)**

sep_dest

(optional) The separator character used in the formatted output. **(a)**

dec_pt_dest

(optional) The decimal point character placed in the formatted output. **(a)**

case

(optional) Controls the effect of the UPCASE and LOCASE statements and also affects the S_MATCH and S_PARSE routines wherever the operation depends on (or is independent of) the case of a character. If *case* is nonzero, the characters [, \,], and ^ are treated as the uppercase versions of {, l, }, and ~, respectively. **(n)**

Discussion

The LOCALIZE subroutine modifies the format in which currency is displayed. It enables you to change the behavior of certain Synergy Language output formatting to comply with local standards.

Omitted arguments default to the current settings so that single characters can be changed without affecting other settings.

LOCALIZE overrides flag 1 of the FLAGS subroutine.

Examples

In the following example, **field** contains the value f123.456,00.

```
xcall localize("f", " ", " ", ".", "f", ".", " ", " ", 1)
field = 12345600, "fff,fff.XX"
```

In the example below, **yarg** contains the value f1.234.567,89, and **xarg** contains the value {}~.

```
record
  zarg          ,d10.2,      1234567.89
  yarg          ,a15
  xarg          ,a10,        "[\]^"

proc
  xcall localize("$", " ", " ", ".", "f", ".", " ", " ", 1)
  yarg = zarg, "$$$,$$$,$$X.XX"
  upcase xarg
  xreturn
end
```

^M – Map a structure to a data area

^M ([*structure_field*,]*data_area*)

Arguments

structure_field

(optional) Defines the structure to map and the data field to access. It can be any valid field expression based on a field within a structure declaration. It can have multiple path elements, any of which can be dimensioned, and the field can be indexed, dimensioned, ranged, and so forth.

data_area

Specifies the data area to be referenced. It can be any valid Synergy Language variable or memory handle. If this expression is alpha, the data area referenced is exactly that of *data_area*. If it is numeric, its value is considered to be a memory handle whose associated memory segment is the data area to reference. (**a** or **n**)

Discussion

^M “maps” a structure to a data area or memory handle and then accesses a field from within that data structure.

The result of this operation is a field whose data type and size is the same as *structure_field* and whose location (and, therefore, contents) is determined by the data area reference by *data_area*.

Semantically, this operation does three things:

- ▶ The structure whose field is used in *structure_field* is logically “aligned” to the first character of the data area specified by *data_area*.
- ▶ Based on this alignment, the logical position and length of the field represented by *structure_field* are determined.
- ▶ If the position and size lie entirely within the referenced data area, a descriptor is generated that references this data field. If the position and size lie outside the data area, a “Map outside bounds of field or handle” error (\$ERR_HSIZEERR) is generated.

If *data_area* is numeric but is not a memory handle, an “Invalid memory handle” error (\$ERR_INVHDL) is generated.

If *data_area* is the only argument, the entire allocated memory area is accessed. If the size of the memory accessed is greater than 65,535 bytes, a “Map outside bounds of field or handle” error (\$ERR_HSIZEERR) is generated.

Report handles, pen handles, RCB handles, and NET handles cannot be used as a memory handle with ^M to access the associated memory.

Examples

```
record
    hnd          ,D_HANDLE
    var          ,i4

structure str
    fld1         ,i4
    fld2         ,i4

proc
    hnd = %mem_proc(DM_ALLOC, ^size(str))
    clear ^i(^m(hnd))                ;Clear entire structure
    var = ^m(str.fld2, hnd)
    ^m(str.fld1, hnd) = var
    hnd = %mem_proc(DM_FREE, hnd)    ;Release memory
.end
```

%MEM_PROC – Implement memory management functions

result = %MEM_PROC (*function*, *arg*[], ...)]

Return value

result

The result of the defined operation. (i)

Arguments

function

A value that defines the function to be performed. These functions are described further on [pages 9-212 through 9-217](#).

DM_ALLOC	Allocate memory.
DM_FREE	Release a memory handle.
DM_GETSIZE	Return the size of memory allocated to an existing memory handle.
DM_REG	Register an external memory segment.
DM_RESIZ	Reallocate a memory handle.
DM_TRIMSIZE	Return the blank-trimmed size of an existing memory handle.

arg

One or more arguments that are defined by *function*.

Discussion

%MEM_PROC implements memory management functions such as allocating memory, releasing or reallocating a memory handle, and registering an external memory segment.

The *function* value always has the function code in the low-order eight bits. You can specify additional qualifiers in *function*, with the constraint that the qualifier information be in the low eight bits. See the individual *function* definitions on the following pages for details on what, if any, additional qualifiers are allowed.

Note that all dynamic memory, static or otherwise, is constrained to the scope of the executing program. When you terminate one program and start another (for example, when you use a STOP chain), the memory allocated by %MEM_PROC is freed.

The following Synergy Language definition file is assumed:

```
.define DM_ALLOC,    ^X(01)    ;Functions
.define DM_REG,      ^X(02)
.define DM_RESIZ,    ^X(10)
.define DM_FREE,     ^X(11)
```

```
.define DM_GETSIZE, ^X(12)
.define DM_TRIMSIZE, ^X(13)
.define DM_STATIC, ^X(0100) ;Qualifiers
.define DM_BLANK, ^X(0200)
.define DM_NULL, ^X(0400)
.define DM_ZERO, ^X(0800)
```

%MEM_PROC – (Continued)

DM_ALLOC – Allocate memory

```
result = %MEM_PROC (DM_ALLOC [.bor .DM_STATIC]
& [ .bor .DM_BLANK | DM_NULL | DM_ZERO], size)
```

Return value

result

The identifying handle associated with the allocated memory. This value is positive if DM_STATIC was specified; otherwise, it is negative.

If the memory request cannot be honored, a “Not enough memory for desired operation” error (\$ERR_NOMEM) is generated.

Arguments

DM_STATIC

(optional) Classifies the returned memory handle as static. If this flag is not present, the handle is classified as volatile, or dynamic. If it is classified as volatile and the handle has not been explicitly released (by a DM_FREE call) at the time the current activation of the current routine is exited, DM_FREE for the returned handle is called automatically.

DM_BLANK

DM_NULL

DM_ZERO

(optional) Specifies that the allocated memory is to be initialized to either blanks or nulls or zeros, respectively. Each qualifier can be used in conjunction with DM_STATIC. If none of these qualifiers are specified, the allocated memory is uninitialized.

size

An expression that specifies the size of the memory segment to allocate. (**n**)

Examples

```
function allocate
  a_size    ,n
stack record
  hnd       ,i4

proc
  hnd = %mem_proc(DM_ALLOC.bor.DM_STATIC, a_size)
  freturn(hnd)
endfunction
```

%MEM_PROC – (Continued)

DM_FREE – Release a memory handle

result = %MEM_PROC (DM_FREE, *handle*)

Return value

result

A null (0) value because the memory handle is now invalid.

We recommend that you equate a memory handle to the %MEM_PROC(DM_FREE) that uses that handle. For example, if *hnd* contains a memory segment handle, the following code should be used to release that handle:

```
hnd=%mem_proc (DM_FREE, hnd)
```

Then, any incorrect reuse of *hnd* as a memory handle generates an error rather than cause unpredictable results.

Arguments

handle

A memory segment handle. (**D_HANDLE**)

Discussion

If *handle* is invalid, an “Invalid memory handle” error (\$ERR_INVHDL) is generated.

%MEM_PROC – (Continued)

DM_GETSIZE – Return the size of memory allocated to a memory handle

result = %MEM_PROC (DM_GETSIZE, *handle*)

Return value

result

The size of memory allocated to the handle. (**n**)

Arguments

handle

An existing memory segment handle allocated with DM_ALLOC. (**i**)

%MEM_PROC – (Continued)

DM_REG – Register an external memory segment and assign a new memory handle

```
result = %MEM_PROC (DM_REG/.bor.DM_STATIC], size, address)
```

Return value

result

The identifying memory handle that is associated with the referenced user data. This allows Synergy Language programmers to access data made available by address. **(i)**

Arguments

DM_STATIC

(optional) Classifies the memory handle as static. If this flag is not present, the handle is classified as volatile.

size

The size of an existing memory segment. **(n)**

address

The address of an existing memory segment. **(D_ADDR)**

Examples

```
static record
    hnd      , i4
    var      , a1024

proc
    hnd = %mem_proc (DM_REG.bor.DM_STATIC, ^size(var), ^addr(var))
```

%MEM_PROC – (Continued)

DM_RESIZ – Reallocate (change the size of) a memory handle

```
result = %MEM_PROC (DM_RESIZ[.bor.DM_BLANK | DM_NULL | DM_ZERO], size, handle)
```

Return value

result

The reallocated memory handle.

Arguments

size

The new size of the memory handle. (**n**)

handle

An existing memory segment handle allocated with DM_ALLOC. (**i**)

DM_BLANK

DM_NULL

DM_ZERO

(optional) Specifies that if *size* is greater than the existing length of the memory segment associated with the handle, the extended memory will be initialized to blanks or nulls or zeros, respectively. If none of these qualifiers are specified, the extended memory is uninitialized.

Discussion

The contents of the original data area are copied to the new data area.

Examples

```
record
    hnd          , i4

proc
    hnd = %mem_proc(DM_ALLOC, 1024)
    hnd = %mem_proc(DM_RESIZ, 2048, hnd)
```

%MEM_PROC – (Continued)

DM_TRIMSIZE – Return the trimmed size of a memory handle

```
result = %MEM_PROC (DM_TRIMSIZE/.bor.DM_BLANK | DM_NULL | DM_ZERO], handle)
```

Return value

result

The blank-trimmed length of the memory associated with the handle. If the entire memory is blank, a value of 1 is returned. (**n**)

Arguments

handle

An existing memory segment handle associated with DM_ALLOC. (**i**)

DM_BLANK

DM_NULL

DM_ZERO

(optional) Specifies that a blank or a null or a zero, respectively, will be used instead of the default blank character to determine the trimmed length.

Examples

The example below returns the blank-trimmed length of **handle**.

```
trim_size = %mem_proc(DM_TRIMSIZE.bor.DM_BLANK, handle)
```

MODNAME – Get the name of a routine

```
xcall MODNAME (choice, routine, [line][, file_number])
```

Arguments

choice

An expression that determines which routine's name is returned: **(n)**

0 Current routine

1 Previous routine

2 Routine before the previous routine

and so on...

routine

The variable that will be loaded with the name of the specified routine. **(a)**

line

(optional) The variable that will be loaded with the line number of the XCALL statement in the specified routine. **(n)**

file_number

(optional) The variable that will be loaded with the number of the .INCLUDEd source file that contains the XCALL statement in the specified routine. **(n)**

Discussion

The MODNAME subroutine returns the name of the current or a previous routine. If the name returned is that of the main routine, it is in the form "MAIN\$name," where *name* is the name of the main routine.

Routine is returned with blanks if you specify a nonexistent routine in *choice*. For example, if *choice* is 10 but the current call chain only has six routines, MODNAME returns blanks. With objects, the method signature is returned, and it can be quite large.

If *choice* is 0, *line* is the line number of the MODNAME statement. If *choice* is 1, *line* is the line number (in the previous routine) of the XCALL statement that called the current routine; and so forth.

When using the .INCLUDE compiler directive to include source code, use *file_number* in addition to *line* to uniquely identify the source file in which *line* exists. Because the file that contains the MODNAME routine is number 1, the first .INCLUDEd file is number 2, the second is number 3, and so forth. Each new file that is processed gets a new file number, even if it has been .INCLUDEd previously. (On .NET, *file_number* is always returned 0.)

Examples

The following subroutine provides the same type of traceback information that you'd get using the TRACE command in the debugger.

```
subroutine showtrace
.define TTCHN      ,1
record
    i                ,d4
    modnam           ,a32
    blanks           ,a32
    line             ,d5

proc
    writes(TTCHN, "Calling chain is:") ;TTCHN opened by calling routine
    xcall modname(i, modnam, line)      ;Start with current routine
do
    begin
        writes(TTCHN, "in routine " + modnam)
        incr i
        xcall modname(i, modnam, line)
        if (modnam .ne. blanks)
            display(TTCHN, "at line " + %string(line))
    end
until(modnam .eq. blanks)
xreturn
endsubroutine
```

MONEY – Specify the currency symbol

```
xcall MONEY(currency)
```

Arguments

currency

An expression that contains the new currency symbol to be used when decimal data is transferred to an alpha variable. If *currency* is longer than one character, only the leftmost character is used. (**a**)

Discussion

The MONEY subroutine changes the dollar-sign formatting character (\$) to another character.

You can change the dollar sign used in explicit decimal to alpha formatting to any ASCII character. However, to avoid confusion, we recommend that you don't substitute another standard formatting character for the dollar sign: Don't use a comma (,), period (.), asterisk (*), hyphen (-), or the uppercase letters X or Z.

The designated currency symbol remains in effect until you execute another MONEY subroutine or until the program terminates.



The MONEY subroutine is provided for compatibility only. We recommend you use the LOCALIZE subroutine, which allows full customization of numeric-to-alpha formatting behavior without changing the format string.

See also

[LOCALIZE on page 9-206.](#)

Examples

The following example converts the dollar sign to a British pound sign.

```
.define TTCHN          ,1
record
    price              ,a10

proc
    open(TTCHN, o, "tt:")
    xcall money('£')
    price = 99.99, "£££,£££.££"
    display(TTCHN, "Price in pounds: ", price)
    stop
.end
```

%NDATE – Return a numeric date

```
date = %NDATE([source])
```

or

```
xcall NDATE(date[, source])
```

Return value

date

A date value in the format YYYYMMDD. (**n**)

Arguments

source

(optional) The source from which %NDATE calculates the returned date. (**a** or **n**)

Discussion

%NDATE returns a numeric date either for the current operating system date or for the Julian period or date string passed.

If *source* is numeric, it is assumed to be the Julian period for the date to be returned. If *source* is alphanumeric, it is interpreted as a standard alpha date in the following format:

```
[DD][-/][MMM][-/][YYYY]
```

or

```
+|-n
```

The hyphen delimiters in the first format are optional, but they must be specified if leading components are omitted. Any omitted component defaults to the current value.

The second format indicates plus or minus *n* days from the current date (for example, +5 or -10).

The default is the current date. See [%JPERIOD on page 9-198](#) for more information.

Examples

```
record
  group date      ,d8
    year          ,d4
    month         ,d2
    day           ,d2
  endgroup
  adate           ,a10
```

System-Supplied Subroutines, Functions, and Classes

%NDATE

```
proc
    open(1, o, "TT:")
    xcall ndate(date)
    adate = ^a(day) + "/" + ^a(month) + "/" + ^a(year)
    writes(1, "Today is " + adate)
    date = %ndate("+7")
    adate = ^a(day) + "/" + ^a(month) + "/" + ^a(year)
    writes(1, "Next week is " + adate)
    close 1
    stop
end
```


%NUMARGS – Return the number of arguments passed to a routine

number = %NUMARGS

Return value

number

The number of formal arguments that have associated activation arguments. See [“How Arguments and Parameters Are Handled”](#) on page 7-14 for a description of the different types of arguments. (i)

Discussion

%NUMARGS returns the number of arguments passed to the current subroutine or function.



For a non-^VAL function, %NUMARGS returns a value that is one greater than the number of arguments passed, because it includes the function's return value in its count. We recommend that all functions that return an integer value be defined as ^VAL.

Examples

If you call the subroutine below using the following line:

```
xcall sub("one", "two")
```

%NUMARGS returns a value of two.

```
subroutine sub
  arg1      ,a
  arg2      ,a
  arg3      ,a
record
  i          ,d2

proc
  for i from 1 thru %numargs
    process arguments
```

^O – Interpret a string as an octal number

octal = ^O(*expression*[, *size*])

Return value

octal

The integer value of an alpha *expression*, evaluated as an octal number. If *expression* is null, the result is 0. (i)

Arguments

expression

The alpha expression to interpret as octal. (a)

size

(optional) A value that specifies the size of the resulting integer. Valid values are

- 1** The result is returned in an **i1** value.
- 2** The result is returned in an **i2** value.
- 4** The result is returned in an **i4** value.
- 8** The result is returned in an **i8** value.

Discussion

If *expression* does not contain only valid octal digits (0 through 7), an “Octal digits expected in argument (string)” error (\$ERR_ODIGXP) is generated.

By default, ^O produces an **i4** value if the return value fits in 32 bits or an **i8** value if the return value is larger than 32 bits. If *size* is specified, the value is returned in an integer of that size. If the value exceeds the specified size, it is truncated to fit.

An ^O value that fits in 32 bits but has the high bit set will be negative. If you want a positive value when a negative **i4** would normally be returned, you can specify a value of 8 in the optional *size* argument.

Examples

The following example uses alpha input as data for an octal arithmetic computation. Note that if the alpha variable contains any nonnumeric data other than a blank, a decimal point, or a sign (+ or –), a “Bad digit encountered” error (\$ERR_DIGIT) is generated.

```
subroutine math
    a_avar          ,a
record
    val1            ,d5
    val2            ,d5
    result          ,d10

proc
    result = (val1*val2) / ^o(a_avar)
    .
    .
    .
    xreturn
endsubroutine
```

%OCT – Return an octal string representation of a numeric value

string = %OCT(*value*[, *magnitude*])

Return value

string

The octal string representation of *value*. (**a**)

Arguments

value

The value to convert to an octal character string. (**n**)

magnitude

(optional) A value that determines the number of characters in the returned string, as follows: (**n**)

- 1** 3-character string for an 8-bit value
- 2** 6-character string for a 16-bit value
- 4** 11-character string for a 32-bit value
- 8** 22-character string for a 64-bit value

Discussion

%OCT converts a decimal data value to a 32-bit longword value.

If *magnitude* is not present or if it contains a value other than 1, 2, 4, or 8, the resulting string is 11 characters long (if *value* can be represented in 32 bits) or 22 characters long (if *value* cannot be represented in 32 bits).

Examples

The following example builds and spawns a command string to check for the protection mode of a specific file. The protection modes are stored in a file named **chkmod.txt**.

```
function chk_mode
    a_file          ,a
    .define MODCHN   ,18
    .define MODFIL   , "WRK:chkmod.txt"

    record
        string      ,a80
        slen        ,d2
        file        ,a30
        modes       ,a9
```

```

value          ,d9
i              ,i1

proc
  file = a_file
  open(MODCHN, i, a_file) [err=nofile];Does file exist?
  close MODCHN
  xcall s_bld(string, slen, "ls -salt %A >%A", a_file, MODFIL)
  xcall spawn(string)           ;Get full listing
  clear string
  file = MODFIL
  open(MODCHN, i, MODFIL) [err=nofile]
  reads(MODCHN, string, nofile)
  modes = string(5:9)           ;Get protection modes
                                ; (check position!)

  for i from 1 thru 9
    value(i:1) = .not.(modes(i:1).eq.'-')
  freturn %oct(value, 1)

nofile,
  writes(TTCHN, "Error: " + file + " not found")
  freturn ("-1")
endfunction

```

OPENELB – Make an ELB’s subroutines available to executing program

```
xcall OPENELB(elb_spec)
```

Arguments

elb_spec

The file specification of the ELB to attach to. (a)

Discussion

The OPENELB subroutine attaches to the specified ELB, making the ELB’s subroutines available to the executing program. In a single routine, you can call OPENELB for a library and then XCALL the library’s routines using XSUBR. In multiple routines, you can call OPENELB for a library, and then in a separate subroutine that is XCALLED later, you can directly execute subroutines from the library. (When a subroutine is first XCALLED, all routines that are directly XCALLED from that routine must be in libraries that are already open.)

Each call to OPENELB opens the specified ELB and any ELBs that are linked to it. If a specified ELB or any linked ELB is not found, the runtime generates a “File not found” error (\$ERR_FNF).

WIN, UNIX

The maximum number of ELBs that can be open at any one time is 256, including all ELBs that are specified on the **dblink** command line or linked against the specified ELBs and automatically opened when the program is started. The maximum length of *elb_spec*, including the directory path and/or logical, is 31 characters.

VMS

Elb_spec must be a filename or logical name; do not include a device, directory, or file type field.

Synergy/DE for OpenVMS uses shared executable images to implement ELBs. The OPENELB subroutine adds the referenced shared images to the active list of shared images for subsequent access through calls to the XSUBR subroutine. It performs no other action in the OpenVMS environment. By default, OpenVMS attempts to locate the ELB in SYSS\$SHARE unless *elb_spec* is a logical. The only way to use an ELB that is not in SYSS\$SHARE is by assigning a logical to refer to it and using that logical name in the OPENELB subroutine.

Examples

```
record
  elb_name           ,a30    ;Executable library name
  end_of_modlist     ,d1     ;Have we reached the end of the
                               ; list of installed modules?
                               ; 1 yes, 0 no

proc
  repeat
    begin
                                     ;From installation data base, get name
                                     ; of executable library for next module
      xcall nxt_module(elb_name, end_of_modlist)
      if end_of_modlist               ;If no more modules installed,
        exitloop                     ; we're finished
      xcall openelb(elb_name)         ;Open executable library
                                     ; for current module
    end
  xreturn
end
```

%OPTION – Determine if a system option is set

status = %OPTION(*option* [, *reset*])

Return value

status

%OPTION returns a value of 1 if the specified Synergy Language system option has been set or a value of 0 if the option has not been set. (i)

Arguments

option

The number of a Synergy Language system option. (See the “[System Options](#)” chapter of *Environment Variables and System Options* for a complete list of Synergy Language system options.) (n)

reset

(optional) Sets or resets a system option at runtime: (n)

0 Reset the system option.

nonzero Set the system option.

Discussion

%OPTION determines whether or not a specific system option is set and optionally changes the option.

Examples

In the example below, %OPTION checks whether the CRT-mode system option (#5) has been set, possibly by the DBLOPT environment variable. If it's not set, the FLAGS subroutine sets the CRT-mode flag.

```
if (.not. %option(5))
  xcall flags(4000, 1)
```

The following subroutine lists the system options set.

```
subroutine show_opts
.define MAXOPT ,54
record
  i           ,i2
  setflg     ,i1,  0
```



```
proc
  display(15, "These system options are set: ")
  for i from 1 thru MAXOPT
    begin
      if (%option(i))
        begin
          if (.not. setflg)
            setflg = 1
            writes(15, "Option " + %string(i))
          end
        end
      if (.not. setflg)
        writes(15, "none")
      xreturn
    endsubroutine
```

The following example sets system option #16 and then resets the option to its previous state:

```
savopt16 = %option(16, 1)
.
.
.
savopt16 = %option(16, savopt16)
```

%PACKED – Convert a numeric expression to packed decimal data

converted = %PACKED(*expression*[, *precision*])

Return value

converted

A converted numeric expression. The result is packed decimal data format or implied-packed decimal data format. (**p**)

Arguments

expression

The numeric expression to be converted. (**n**)

precision

(optional) The precision value for an implied-packed decimal number. (**n**)

Discussion

%PACKED converts a numeric expression to the packed or implied-packed decimal data type.



The %PACKED function has been superseded by the **p** data type.

Examples

```
val = %packed(324)
impval = %packed(436.7, 3)
```

%PAINT – Return the code for the current paint character

code = %PAINT

Return value

code

The decimal character code of the current paint character. (i)

Discussion

The default paint character code is 32 (an ASCII blank). You can change the paint character by calling the PAINT subroutine.

When you enable CRT mode using the FLAGS subroutine (runtime option flag 4), your program can initially display a data entry field as a sequence of paint characters. As the user enters data over the paint characters, they disappear, but if he or she deletes a character using CTRL+U or DELETE, the paint character reappears.

See also

- ▶ [PAINT on page 9-234.](#)
- ▶ [FLAGS on page 9-126.](#)

Examples

The following function is called to change the paint character. The new paint character is passed as an argument. If the current paint character is correct, no change is made.

```
function chk_paint
    a_paint      ,a
record
    cur_paint    ,a1
    dval         ,d3

proc
    if (%decml(%paint) .ne. %decml(a_paint))
        xcall paint(a_paint)
        freturn (%true)
    endifunction
endfunction
```

PAINT – Redefine the paint character

```
xcall PAINT(paint_char)
```

Arguments

paint_char

An expression whose leftmost character will become the new paint character. (a)

Discussion

The PAINT subroutine changes the current paint character to another character.

Although you can substitute any character for the paint character, we recommend that you select a printing character.

When program execution begins, the paint character defaults to a blank. During program execution, you can use the %PAINT intrinsic function to obtain the decimal character code for the current paint character.

The paint character is completely ignored unless the FLAGS subroutine sets runtime option flag 4 or you set system option #5 with the DBLOPT environment variable to indicate that the program is communicating with a CRT-type terminal. When the CRT flag is set, pressing the DELETE key (or its equivalent) during input deletes the most recently typed character and “paints” the position with the paint character. The cursor remains positioned at the deleted character. If the user presses CTRL+U, the entire current line is deleted, every character of its contents is replaced with the paint character, and the cursor is positioned at the first character on the line.

When you call the PAINT subroutine, your program must initialize the appropriate input area to the correct paint character. The user then enters data over the paint characters. When the user deletes either a character or the entire line, Synergy Language automatically repaints the deleted positions with the current paint character.

Examples

The example below enables CRT terminal mode by setting runtime option flag 4, sets the paint character to a period (.), and initially fills the **input** field with periods. The user types over the periods when entering input.

```
.define TTCHN,1
record
    input      ,a20

proc
    open(TTCHN, i, "tt:")
    xcall flags(4000, 1)           ;Enable CRT terminal mode
    xcall paint('.')              ; and set paint character
```

```
xcall fill('.', input)
repeat
  begin
    display(TTCHN, "Name: ", input, $scr_mov(0,-10))
    reads(TTCHN, input) [eof=done]
    forms(TTCHN, 2)
    display(TTCHN, "Press RETURN to continue: ")
    reads(TTCHN, input) [eof=done]
  end
done,
  stop
end
```

PAK – Compress numeric data

```
xcall PAK(record, size, field[, ...])
```

Arguments

record

The record that contains the fields to be compressed. (**a**)

size

A variable that contains the size of the compressed record after the PAK operation. (**n**)

field

One or more decimal fields in the specified record. These fields must exist in *record* and must be listed in ascending order as they appear in the record. (**d**)

Discussion

The PAK subroutine compresses fields of numeric data to save disk space.

Each packed field produced by the PAK subroutine has the following length:

$(\text{number of characters}/2) + 1$

Use the UNPAK subroutine to restore the fields' original values.



The PAK subroutine has been superseded by the **p** data type.

Examples

If we have the following data division:

```
record rec1
  fld1      ,d5
  fld2      ,a4
  fld3      ,d18
  fld4      ,d9
  fld5      ,a33
  fld6      ,d3

record ,x
  fld1x     ,a3
  fld2x     ,a4
  fld3x     ,a10
  fld4x     ,a5
  fld5x     ,a33
  fld6x     ,a2
```

```
record
    size      ,d2
```

the statement

```
xcall pak(rec1, size, fld1, fld3, fld4, fld6)
```

compresses **rec1** down to the fields described in the overlay record. This shortened record can then be output to a file to save space. When you're ready to read the record back, the following statement restores the original values to the fields.

```
xcall unpack(rec1, fld1, fld3, fld4, fld6)
```

Assuming the above layout is in the file **layout.def**, the following subroutine save the data from one file to a packed backup file.

```
subroutine bck_restore
    a_flag      ,d
    a_source    ,a
    a_bckup     ,a
    .include "layout.def"
    .define BCKUP      ,1
    .define RESTORE    ,2
    .define SRCCHN     ,10
    .define BCKCHN     ,11

proc
    if (a_flag.eq.BCKUP) then
        begin
            open(BCKCHN, o, a_bckup)
            open(SRCCHN, i, a_source)
            repeat
                begin
                    reads(SRCCHN, rec1) [eq = done]
                    xcall pak(rec1, size, fld1, fld3, fld4, fld6)
                    puts(BCKCHN, rec1)
                end
            end
        end
    else
        begin
            open(BCKCHN, i, a_bckup)
            open(SRCCHN, o, a_source)
            repeat
                begin
                    gets(BCKCHN, rec1) [eq = done]
                    xcall unpack(rec1, fld1, fld3, fld4, fld6)
                    writes(SRCCHN, rec1)
                end
            end
        end
    end
```

```
done,  
    close SRCCHN  
    close BCKCHN  
    xreturn  
endsubroutine
```


PARSE – Extract components of a file specification

```
xcall PARSE (file_spec, [translation], [node], [device], [directory], [name], [extension],
&           [version][, expanded_spec])
```

Arguments

file_spec

The variable, literal, or record that contains the input file specification whose component elements are to be extracted. **(a)**

translation

(optional) A flag to inhibit the replacement of default file specification components, such as a default extension. If *translation* contains a nonzero value, translation does not occur. The default value for *translation* is zero, which means translation occurs. **(n)**

node

(optional) The variable that will be loaded with any node or remote computer name specified in the input file specification. If the node is not specified in the input string, the field is set to blanks. On OpenVMS, if a node is returned, it is terminated with two colons (::). On Windows and UNIX, if a remote computer name is returned, it is preceded by an “at” sign (@). **(a)**

WIN, VMS

device

(optional) The variable that will be loaded with the device portion of the file specification. On OpenVMS, the default device is SYS\$DISK. The returned device is terminated by a colon (:). **(a)**

directory

(optional) The variable that will be loaded with the directory portion of the file specification. The default is the home directory. The format of *directory* depends on the operating system: **(a)**

System	Directory specification
Windows	\directory
UNIX	/directory
OpenVMS	[directory]

name

(optional) The variable that will be loaded with the name portion of the file specification. The name does not include a period or semicolon from the input field. **(a)**

extension

(optional) The variable that will be loaded with the file extension portion of the file specification. The extension includes a period. The default extension is either **.ddf** or the extension set by the SETDFN subroutine. **(a)**

VMS

version

(optional) The variable that will be loaded with the version portion of the file specification. The version includes a semicolon (;). **(a)**

expanded_spec

(optional) Returned with the expanded file specification. **(a)**

Discussion

The PARSE subroutine enables you to access individual elements of a file specification.

If *translation* is zero (either explicitly or by default), PARSE translates logical names and supplies the Synergy Language default filename to fill in missing components.

If the file specification doesn't contain an extension (either explicitly or implicitly through the default name), a period is returned for the extension. If default name processing is disabled, blanks are returned.

If any other components of the file specification are missing, blanks are returned.

Expanded_spec returns the specification of the file that is opened if *file_spec* is used in an OPEN statement.

Data is moved to the subroutine arguments (except *file_spec* and *translation*) according to the rules for moving alpha data.

WIN, UNIX

If system option #35 is set, the PARSE subroutine stops at the semicolon (;) and doesn't include it in the filename or file extension.

VMS

The PARSE subroutine uses the RMS \$PARSE facility to translate and isolate the node, device, directory, filename, extension, and version fields from the specified file descriptor string. If the file specification doesn't contain a version number (either explicitly or implicitly through the default name), a semicolon is returned for the version number. If default name processing is disabled, blanks are returned.



You cannot parse a remote filename, except to its lowest local translation. See below for an example.

Examples

```
record
    node          ,a10
    device         ,a30
    dir            ,a80
    name           ,a20
    ext            ,a4
    vers           ,a2

proc
.
.
    xcall parse("DTKMAPFIL:", 0, node, device, dir, name, ext, vers)
```

If DTKMAPFIL is assigned to **SYSS\$DISK:[DBLV5.TOOLKIT] dtkmap.ism;1**, for example, the above XCALL on OpenVMS returns the following data:

```
node          <blank>
device        SYS$DISK:
dir           [DBLV5.TOOLKIT]
name          DTKMAP
ext           .ISM
vers          ;1
```

Using the same program on Windows or UNIX with DTKMAPFIL set to **DAT:DTKMAP.ism**, where DAT is set to DTKMAPFIL:@myserver, the XCALL returns the following:

```
node          @myserver
device        <blank>
dir           DTKMAPFIL:
name          dtkmap
ext           .ism
vers          <blank>
```

^PASSED – Determine whether an argument was passed

`^PASSED(parameter)`

Arguments

parameter

The routine parameter in question. (**a** or **n**)

Return value

If the specified argument has been passed to the current routine, ^PASSED returns a value of one. If the argument has not been passed, the returned value is zero. (**i**)

Discussion

^PASSED determines whether or not a routine argument has been passed.

Parameter can be specified as the field name of a parameter, or by using ^ARG(*n_expression*) or ^ARGN(*n_expression*).

^PASSED cannot be present in a main routine.

Examples

```
subroutine sub
  arg1      ,a           ;Optional parameter
  arg2      ,a           ;Optional parameter
  arg3      ,a           ;Optional parameter
record
  i          ,d2

proc
  for i from 1 thru %numargs
    if (^passed(^arg(i)))
      .
      .   process argument(i)
      .
```

^PASSTHRU – Access a parameter as its calling type

`^PASSTHRU (parameter)`

Arguments

parameter

The parameter that will be accessed as its calling type.

Discussion

^PASSTHRU accesses the parameter as its calling type. It is only useful when passing an unspecified parameter on to another routine.

Parameter can be ^ARG() instead of a parameter name if you want to get the parameter by its position.

POSRFA – Restore the saved position in an ISAM file

```
xcall POSRFA(channel, rfa[, krf])
```

Arguments

channel

The number of the open channel. (**n**)

rfa

The previously saved file position (as returned by the GETRFA subroutine) that will be restored. (**a6**)

krf

(optional) The key of reference. (**n**)

Discussion

POSRFA works in conjunction with the GETRFA subroutine. Once the current position in an ISAM file has been saved using GETRFA, POSRFA can restore that position. Synergy Language stores the file position returned from GETRFA in binary form to be used as the *rfa* argument to POSRFA.



POSRFA is only available for consistency with DBL version 4. It has been superseded by the RFA qualifier on I/O statements like READ, FIND, STORE, and so forth.



The contents of *rfa* are a special binary value and must be preserved. Be careful in doing assignments or moving the contents into another variable.

If the record at a saved position has been deleted, you'll get an "Invalid record's file address" error (\$ERR_INVALRFA) on OpenVMS. If an RFA refers to a deleted record, a "Deleted record" error (\$ERR_DELREC) will be generated when a read is attempted.

If the channel is opened to an ISAM file and you don't specify *krf*, the POSRFA subroutine sets the key of reference to the primary key. All subsequent I/O operations access that key until it is explicitly changed by a READ, a FIND, or another call to POSRFA.

VMS

If you specify a nonzero *krf* value, Synergy Language generates an "Invalid operation for file type" error (\$ERR_FILOPT). You can use the POSRFA subroutine on any file type. The *rfa* variable must be declared as data type **a6**.

See also

- ▶ [GETRFA on page 9-150.](#)
- ▶ [RFA on page 4-237.](#)

Examples

The following subroutine creates a temporary tag file to sort data from an ISAM file that is in a nonkey field. It reads through the file searching for records with a nonblank entry in the nonkey field. It then creates a temporary tag file and saves the RFA position of the ISAM file and the nonblank field. It sorts the temporary file using the nonblank field as the sort key, then extracts the appropriate records from the ISAM file in sorted order to pass to a print subroutine.

```

subroutine report
  .define ISMCHN      ,10
  .define TAGCHN      ,11
  .define TAGFILE,"tagfile"
  record ism_rec
    key              ,a10
    ism_fld          ,a5
  record tag_rec
    tag_rfa          ,a6
    tag_sort         ,a5

  proc
    open(ISMCHN, i:i, "isam")
    open(TAGCHN, o, TAGFILE)
    repeat
      begin
        reads(ISMCHN, ism_rec, done1, GETRFA:tag_rfa)
        if (ism_fld)
          begin
            tag_sort = ism_fld
            puts(TAGCHN, tag_rec)
          end
        end
      end

    done1,
      close TAGCHN
      sort(input=TAGFILE, key=tag_sort, record=tag_rec,
        & options="recsize=11/e")
      open(TAGCHN, i, TAGFILE)
      repeat
        begin
          gets(TAGCHN, tag_rec, done2)
          xcall posrfa(ISMCHN, tag_rfa)

```

System-Supplied Subroutines, Functions, and Classes

POSRFA

```
        reads(ISMCHN, ism_rec)
        xcall print_it(ism_rec)
    end

done2,
    close TAGCHN
    close ISMCHN
    xreturn
endsubroutine
```


PURGE – Delete previous versions of a file

VMS

xcall PURGE (*file_spec*)

Arguments

file_spec

A field, literal, or record that contains the specification of the file or files to be deleted. (a)

Discussion

The PURGE subroutine deletes all but the most recent version(s) of the specified file(s) from the specified directory.

The version with the highest version number is not deleted.

File_spec can contain wildcards, but it cannot contain a version number. The default directory is established at log-in or by the SET DEFAULT command. If the PURGE subroutine cannot find the specified file(s), a “File not found” error (\$ERR_FNF) or “Bad filename” error (\$ERR_FILSPC) is generated.

The rules associated with the DELET subroutine also apply to the PURGE subroutine.

If the PURGE system service fails, a “DCL purge error” (\$ERR_PURGE) is generated, and the OpenVMS error that caused the failure is available through the %SYSERR function.



Because PURGE requires the command line interpreter, if you are using x/ServerPlus, you cannot XCALL this routine from within a shared image.

See also

[DELET on page 9-95.](#)

PUTCM – Put the contents of a variable into a common area

VMS

xcall PUTCM(*variable*)

Arguments

variable

A variable up to 254 characters long. The PUTCM subroutine places the contents of this variable into the process common area. (**a**)

Discussion

The PUTCM subroutine calls the OpenVMS system service LIB\$PUT_COMMON with the specified argument, thereby enabling your program to transfer data using a common area.

The PUTCM subroutine, in conjunction with the GETCM subroutine, manipulates data within a common area that is maintained by OpenVMS for each process on the system. Because this common area is unique to each process, Synergy Language programs can use this area as a private “scratch pad” space for communication between programs that will be chained to later.

See also

- ▶ [GETCM on page 9-139](#).
 - ▶ The LIB\$PUT_COMMON section of your OpenVMS runtime library manual.
-

QSORT – Perform an in-memory sort

```
xcall QSORT (array, elements[, function])
```

Arguments

array

The first element of an array of fields or structures to sort. (**a** or **n**)

elements

The number of elements in the array. (**n**)

function

(optional) The name of the function for element comparisons. The function is a ^VAL function with two arguments that are each an array element. See the example in the Discussion section below. (**a**)

Discussion

The QSORT subroutine is an in-memory sort subroutine.

QSORT is a very simple comparison routine; the complexity is up to you.

If *function* is not present, the comparison is alpha for the total length of the array elements. If *function* is present, the named ^VAL function must be accessible when the QSORT subroutine is invoked. The function must have two arguments of the same type that match the type of an array element, and it must have the same format as the following:

```
function COMPARISON_ROUTINE, ^val
    element_1 ,a                ;Argument type must be the same as the
    element_2 ,a                ; array element

proc
    if (element_1 .lt. element_2)
        freturn(-1)            ;Return element #1 < element #2
    if (element_1 .gt. element_2)
        freturn(1)             ;Return element #1 > element #2
    freturn(0)
endfunction
```

Examples

In the following example, an array of 100 records containing client information that consists of an ID, name, and address is sorted by address. In the comparison function, the two arguments are each templates of a single array element.

```
function COMPARE_CLIENTS, ^val
  group a_g1 ,a          ;Comparison function passed two arguments
    id      ,d6
    name    ,a40
    addr    ,a20
  endgroup
  group a_g2,a          ;Both elements of the array to be sorted.
    id      ,d6          ; The comparison function has complete
    name    ,a40         ; control over deciding exactly how the
    addr    ,a20         ; array elements collate. The comparison
  endgroup              ; function simply returns the following:
                        ;   ARG1 = ARG2:  0
                        ;   ARG1 < ARG2: <0
                        ;   ARG1 > ARG2: >0

proc
  if (a_g1.addr .lt. a_g2.addr)
    freturn(-1)
  if (a_g1.addr .gt. a_g2.addr)
    freturn(1)
  freturn(0)
endfunction
```

R5ASC – Convert a RAD50 word into ASCII

```
xcall R5ASC(rad50, ascii)
```

Arguments

rad50

A variable whose first two characters are treated as a single RAD50 word to be converted. (a)

ascii

The variable that will be loaded with three ASCII characters, loaded left-justified over blanks. (a)

Discussion

The R5ASC subroutine converts a two-character RAD50 word into three ASCII characters. (RAD50 was a compression technique used on PDP-11 systems.)

The R5ASC subroutine complements the ASCR5 subroutine.

RAD50-to-ASCII conversions are valid for the following ASCII characters:

- ▶ Uppercase letters A through Z
- ▶ Digits 0 through 9
- ▶ The following special characters: dollar sign (\$), blank (), and period (.)

See also

[ASCR5 on page 9-28.](#)

Examples

The following example prompts the user to enter three ASCII characters. It converts those characters to two RAD50 characters, converts the RAD50 characters back to ASCII, and displays the results to the screen.

```
.define TTCHN,1
record
    rad50      ,a2
    ascii      ,a3

proc
    open(TTCHN, i, "tt:")
    display(TTCHN, "Enter 3 characters: ")
    reads(TTCHN, ascii)
    xcall ascr5(ascii, rad50)
    xcall r5asc(rad50, ascii)
```

System-Supplied Subroutines, Functions, and Classes

R5ASC

```
        writes(TTCHN, "After applying the ASCR5 subroutine")
        writes(TTCHN, "and then the R5ASC subroutine, your")
        display(TTCHN, 'characters appear as "', ascii, "'")
        stop
    end
```

RANDM – Generate a random number

```
xcall RANDM(number, seed_1, seed_2)
```

Arguments

number

Returned with the next random number in the sequence. (**n**)

seed_1

seed_2

Variables that together determine the next random number to be generated. (**n**)

Discussion

The RANDM subroutine generates a random number from 0 to 32767.

RANDM generates a random number based on two seed values. *Seed_1* and *seed_2* must be at least five digits long.

After the number is generated, the RANDM subroutine automatically updates the values in *seed_1* and *seed_2*, so you can call RANDM several times successively to generate a sequence of random numbers.

You can start a new sequence by changing the values in *seed_1* and *seed_2*. Each seed value must be between 0 and 32767.

Examples

The following example generates a random number in the format ZZ,ZZX, based on two numbers entered by the user.

```
.define TTCHN,1
record
    random      ,d5
    seed1       ,d5
    seed2       ,d5
    io          ,a10
    count       ,d4
    i           ,d4

proc
    open(TTCHN, i, "tt:")
    display(TTCHN, "Enter value for seed 1: ")
    reads(TTCHN, io)
    seed1 = io
    display(TTCHN, "Enter value for seed 2: ")
```

System-Supplied Subroutines, Functions, and Classes

RANDM

```
reads(TTCHN, io)
seed2 = io
display(TTCHN, "How many numbers?  ")
reads(TTCHN, io)
count = io
for i from 1 thru count
  begin
    xcall randm(random, seed1, seed2)
    io = random, "ZZ,ZZX"
    writes(TTCHN, io)
  end
stop
end
```


RCFLG – Control the receive option flags

UNIX, VMS

`xcall RCFLG (flags [, control])`

Arguments

flags

The receive option flags. Each flag is a single digit, and the digit positions are assigned ordinal numbers from right to left. Thus, the low-order digit is digit number one. Synergy Language examines the flags value digit by digit to determine which flags are set, and hence, which functions are to be performed. The Discussion section below contains a detailed description of the flags and their meanings. (**n**)

control

(optional) A value that controls how the individual digit positions in the flag specification value are interpreted. If *control* is not specified, each digit in *flags* is interpreted as follows: (**n**)

- 0** Reset the associated flag.
- 1** Set the associated flag.
- n** Leave the associated flag unmodified, where *n* is any other decimal number.

We recommend that you use the *control* value when setting and resetting your *flags* values. If *control* is present, its value is interpreted as follows:

- 0** Reset all flags whose corresponding digit position contains a nonzero number. Don't modify the other flags.
- 1** Set all flags whose corresponding digit position contains a nonzero number. Don't modify the other flags.
- 2** Set *flags* to a value that reflects the current settings of the flags and leave all flags unmodified. Each digit of the *flags* variable is set to zero if the associated flag is currently reset or to one if the flag is set. If the length of the variable is less than three characters, the settings are returned as in a decimal-to-decimal assignment. (See [“Moving decimal/packed data to a decimal/packed destination” on page 3-33](#) for more information about decimal-to-decimal assignments.)

Discussion

The RCFLG subroutine controls the settings for the receive option flags, which affect how a RECV statement searches for local, group, and global message queues.

Synergy Language interprets the *flags* argument as if it were storing it to a **d3** variable. All rules that apply when storing to a decimal variable apply here. For example, if you specify

```
xcall rcflg(01)
```

Synergy Language interprets your argument as 001. (All flags corresponding to these zeros are cleared.)

The Synergy Language message facilities maintain the following queues for messages:

- ▶ A global message queue for the system
- ▶ One group message queue for each unique group code of all the UICs running Synergy Language programs. (A UIC is a unique project and programmer number combination for each account.)
- ▶ One local queue for each process executing a Synergy Language program

Normally, when a Synergy Language program executes a RECV statement, the message controller begins searching for the requested message ID in the local queue for that process. If it can't find a message that matches the ID, the message controller searches first in the group queue and finally in the system-wide global message queue. If the requested ID is not found in any of the queues, the RECV statement transfers control to a specified label.

The RCFLG subroutine enables you to change the searching algorithm of the message controller by setting one or more flags. Remember that the rightmost digit in the flag specification string corresponds to flag number one, as shown in [figure 9-3](#).

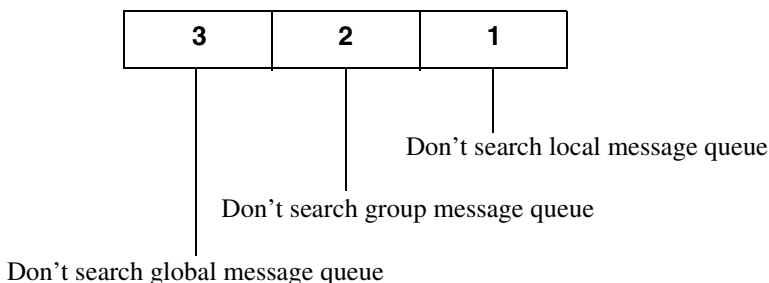


Figure 9-3. RCFLG runtime option flags.

The following list describes the flags in more detail. The initial state of the search flags is zero (reset).

Flag	Description
1	When flag 1 is set, the message controller won't search for the message ID in the local message queue.
2	When flag 2 is set, the message controller won't search for the message ID in the group message queue.
3	When flag 3 is set, the message controller won't search for the message ID in the global message queue.

Examples

The following subroutine restricts message queue access depending on the user's security level.

```

subroutine securmsg
    a_securlvl      ,d2           ;Security level: 1-local only
    .define NO_LOCAL      ,1       ; 2-group and local,
    .define NO_GROUP      ,10      ; 3-local, group, and global
    .define NO_GLOBAL     ,100

proc
    xcall rcflg(111,0)           ;Clear flags
    case a_securlvl of
        begincase
            xcall rcflg(NO_GROUP + NO_GLOBAL, 1)
            xcall rcflg(NO_GLOBAL, 1)
            nop
        endcase
    return
endsubroutine

```

RCVID – Change the default name referenced by RECV

UNIX, VMS

```
xcall RCVID([new_id])
```

Arguments

new_id

(optional) A field that contains the new default message ID. *New_id* can have up to 32 characters on OpenVMS systems and up to 6 characters on Windows and UNIX. **(a)**

Discussion

The RCVID subroutine changes the default name that the RECV statement references when “[SELF]” is specified in the statement as the message ID.

Normally, if the third argument (*id*) in the RECV statement is [SELF], Synergy Language assumes that the message ID is the first six characters of the receiving program’s name. You can change this default by calling the RCVID subroutine and passing a different message ID.

If you don’t specify *new_id* (or if it’s null), the first six characters of the main routine filename are restored as the default ID name.

See also

- ▶ [“Messaging” on page 1-38.](#)
- ▶ The [SELF] message ID under [RECV on page 4-160.](#)

Examples

The example below changes the default message ID to **test**, receives a message with that ID, changes the default ID back to the first six characters of the filename (**proga**, which in this case only has five characters), and then receives a message with the **proga** message ID.

```
.name proga
.define TTCHN,1
record
    newid      ,a6,    "test"
    data       ,a50
    leng       ,d3

proc
    open(TTCHN, o, "tt:")
    xcall rcvid(newid)                ;Change default ID to "test"
    recv(data, err, "[SELF]", leng)  ;Receive message with ID "test"
    xcall rcvid                      ;Change default ID back to "proga"
```

```
    recv(data, err, "[SELF]", leng)      ;Receive message with ID "proga"
    stop

err,
    forms(TTCHN, 1)
    display(TTCHN, "No message was received")
    close
    stop
end
```

%RDLEN – Return the length of the last record read

%RDLEN returns the length of the last record that was read by a READ, READS, ACCEPT, GET, GETS, or GLINE statement, or one of the Synergy windowing API I/O functions.

%RDLEN is equivalent to %RSIZE. For more information, see [%RSIZE on page 9-270](#).

%RDTRM – Return the terminating character of the last record read

%RDTRM returns the numeric code for the record terminator of the last record read by a READ, READS, ACCEPT, GET, or GETS statement, or one of the Synergy windowing API I/O functions.

%RDTRM is equivalent to %RTERM. For more information, see [%RTERM on page 9-275](#).

%RECNUM – Return the number of the last record processed

last = %RECNUM(*channel*)

Return value

last

The number of the last record processed on *channel*. If no operation has occurred on the file, %RECNUM returns a value of zero. **(i)**

Arguments

channel

The channel to check for the last record processed. **(n)**

Discussion

%RECNUM is valid for any non-ISAM file, but it is only updated on a successful I/O operation.

Encountering end-of-file does not change the value of %RECNUM. When you encounter end-of-file, you can use %RECNUM to get back to the last record that was processed.



%RECNUM is not updated on a FIND because FIND does not transfer data.

Examples

The following function returns the number of records in a file.

```
function numrecs
  a_data      ,a
  a_chn       ,d

proc
  read(a_chn, a_data, ^last)
  freturn %recnum(a_chn)
endfunction
```


^REF – Pass an argument by reference

^REF (expression)

or

^REF (argument) ,typesize

Arguments

expression

An argument to pass by reference. (**a** or **n**)

argument

An argument declared as type reference. (**a** or **n**)

typesize

The data type and size specification.

Discussion

^REF is used to pass an argument by reference or to declare an argument as type reference.

The result of *expression* is passed by reference, which means that the address of the data associated with the argument is passed by value.

^REF is primarily used to pass arguments to C routines. If you use ^REF when passing arguments to Synergy Language routines, the corresponding parameters in the Synergy Language routines must be declared as type ^REF with any valid data type.

Use the second syntax for ^REF to declare a subroutine parameter. Type **n** is not allowed.

Examples

```
record
    text      ,a8      "The Text"
proc
    xcall sub(^ref(text))      ;Pass argument by reference
end

subroutine sub
    ^ref(a_arg) ,a8      ;Receive argument by reference as a8
proc
    open(1, o, "TT:")
    writes(1, a_arg)
    close 1
    xreturn
endsubroutine
```

RENAM – Rename a file

```
xcall RENAM(new_name, old_name)
```

Arguments

new_name

An expression that contains the new specification to be assigned to the file. (**a**)

old_name

An expression that contains the file's current specification. (**a**)

Discussion

The RENAM subroutine changes the name of a file. See [OPEN on page 4-124](#) for the correct format of a file specification. If *new_name* has no extension, the extension defaults to the *old_name* extension.

If you specify a new filename that already exists, one of two things occurs, depending on whether the FLAGS subroutine runtime option flag 3 is set:

- ▶ If flag 3 is set, the runtime checks for an existing file with the same name, and a “Cannot supersede existing file” error (\$ERR_REPLAC) is generated.
- ▶ If flag 3 is *not* set, the runtime does not check for duplicate filenames, and the existing file is overwritten as part of the renaming process. For example, if file **a.b** already exists and the following statement is executed, the current **a.b** file is deleted, and the current **old.ddf** is renamed to become the new **a.b** file:

```
xcall renam("a.b", "old.ddf")
```

If the RENAM subroutine can't find the specified old filename, you'll get a “File not found” error (\$ERR_FNF). If *old_name* is an ISAM file and *new_name* doesn't have an extension, both the data file and the index file are renamed to *new_name*, with the same extension as *old_name*.

Old_name and *new_name* can also include a search list logical, which specifies one or more directory paths in which to search for the file. If a search list logical is used on the *new_name* specification, the renamed file is placed in the first directory specified in the search list.

When using remote specifications for *new_name* and *old_name*, any server name that is specified as part of *new_name* is ignored, and *new_name* is assumed to be on the same machine as *old_name*.

In previous versions of Synergy Language, you could also specify a *channel* argument on the RENAM subroutine. This syntax is still valid, but *channel* is ignored.

Before renaming a file, make sure you close any channels that your program has open to that file.

VMS

The new file *must* be on the same device as the old file.

If *new_name* doesn't contain an explicit version number, and a file already exists with the same name, the file is renamed to the next higher version number. If *new_name* does contain an explicit, nonzero version number and a file of the same name and version number already exists, the RENAM subroutine replaces the existing file (unless flag 3 is set).

If system option #35 is set for VAX DIBOL compatibility, and no version number is specified on *new_name*, all previous versions of *new_name* are deleted before the rename occurs (if the source is accessible). If, however, flag 3 is set, if *any* version of *new_name* exists, a fatal "Invalid option" error (\$ERR_INVOPT) is generated.

If *new_name* or *old_name* ends with a semicolon but no version number, Synergy Language assumes the version number is zero. For example, if you rename **file.ddf;** to **new.ddf**, the RENAM subroutine renames **file.ddf;0** to **new.ddf**.

You cannot rename across logical drives on OpenVMS.

If either the old or the new file is already open by any user, a "File in use by another user" error (\$ERR_FINUSE) is generated.

See also

- ▶ [#35](#) in the "System Options" chapter of *Environment Variables and System Options*.
- ▶ [COPY](#) on page 9-57.

Examples

The following example renames "beta" to "alpha" and **oldfile** to **newfile**. If a file called **newfile** already exists, you'll get an error message and **oldfile** won't be renamed.

```
literal
    oldfile          ,a*,    "dp3:abcde.xyz"
    newfile          ,a*,    "dp3:a12.c"

proc
    xcall flags(7000320)
    xcall renam("alpha", "beta")
    xcall renam(newfile, oldfile)
    xreturn
end
```

The example below uses a search list logical. We'll assume the MYFILES logical is defined as follows:

```
MYFILES=/usr/programs,/usr/programs/new
```

Synergy Language first looks in /usr/programs for the **from** file, and if it doesn't find it, it looks in /usr/programs/new. The renamed file is placed in /usr/programs.

```
xcall renam("MYFILES:to", "MYFILES:from")
```

%RFA – Return a record’s address

address = %RFA(*channel*[, *krf*])

Return value

address

A value that uniquely identifies a record within a file of any organization. The return value is a six-byte binary address. **(a6)**

Argument

channel

The channel of an open data file. **(n)**

krf

(optional) The variable that will be loaded with the key of reference of the last READ or FIND on an ISAM file. **(n)**

Discussion

%RFA is an alternative to the GETRFA subroutine and a partial alternative to the GETFA subroutine.



We suggest you use the GETRFA qualifier on I/O statements like READ, FIND, and so forth, instead of using the %RFA intrinsic function.

%RND – Round a value to the nearest whole number

whole_num = %RND(*expression*)

Return value

whole_num

The nearest whole-number value of a numeric expression. The result has the same numeric data type as *expression*. (**n**)

Arguments

expression

The numeric expression to round. (**n**)

Discussion

%RND rounds a numeric expression to the nearest whole number.

%RND follows normal rounding rules: fractional values less than .5 are rounded down, while fractional values of .5 and above are always rounded up. The result does not contain any fractional precision.

The difference between %RND and %ROUND is that with implied-decimal arguments, %RND returns a result with no fractional precision, and %ROUND returns a result with the same fractional precision as *expression*.

Examples

The following example returns a value of 35.

```
%rnd(34.56)
```

%ROUND – Round a value to the nearest whole number

whole_num = %ROUND(*expression*)

Return value

whole_num

The nearest whole-number value of a numeric expression. The result has the same numeric data type as *expression*. **(n)**

Arguments

expression

The numeric expression to round. **(n)**

Discussion

%ROUND rounds a numeric expression to the nearest whole number.

%ROUND follows normal rounding rules: fractional values less than .5 are rounded down, while fractional values of .5 and above are always rounded up. The fractional precision in the result is the same as the fractional precision in *expression*, and it is filled with zeros.

The difference between %RND and %ROUND is that with implied-decimal arguments, %RND returns a result with no fractional precision, and %ROUND returns a result with the same fractional precision as *expression*.

Examples

The following example returns a value of 35.00.

```
%round(34.56)
```

%RSIZE – Return the size of the last record read

```
size = %RSIZE
```

Return value

```
size
```

The size of the last record that was read. (i)

Discussion

%RSIZE returns the size of the data from the last READ, READS, ACCEPT, GET, or GETS operation, or the last window I/O function.

%RSIZE can be especially useful with the READS statement, because the length of the record that was read can vary. For example, you might use %RSIZE to determine the number of characters read from the terminal to limit the processing of an input record. %RSIZE is equivalent to %RDLEN.

Examples

```
repeat
  begin
    display(ttchn, "Enter next command: ")
    reads(ttchn, input, done)
    if (%rsize.eq.0) then
      writes(ttchn, "Null input ignored")
    else
      buf(1,5) = %rsize , "ZZZX"
    end
  end
```

The following subroutine reads a specified minimum amount of terminal input. If the input is too short, the prompt and input are cleared from the screen, and input is requested until the minimum length is entered.

```
subroutine input
  a_minsiz      ,n
  a_data        ,a
  a_prompt      ,a

proc
get_inp,
  display(TTCHN, a_prompt, ": ")
  reads(TTCHN, a_data, done)
  if (%rsize .lt. a_minsiz)
```



```
        begin
            writes(TTCHN, "Data too short")
            display(TTCHN, $scr_pos(-1,-80), $scr_clr(LINE),
&          $scr_pos(-1,-80), $scr_clr(LINE))
            call get_inp
        end
done,
    xreturn
endsubroutine
```

RSTAT – Return information about the last record read

```
xcall RSTAT(size[, term_char])
```

Arguments

size

The variable that will be returned with the size of the last record read. For READ and READS operations, the returned size doesn't include the record terminator that ended the input operation. **(n)**

term_char

(optional) The variable that will be returned with the character that terminated the input operation, loaded left-justified over blanks. *Term_char* should be a one-character field. **(a)**

Discussion

The RSTAT subroutine returns the size and terminating character of the last record read by a GET, GETS, READ, READS, ACCEPT, W_DISP, W_DISP(WD_READS), or W_FIELDS(WF_INPUT) operation.

The RSTAT subroutine is similar to the RSTATD subroutine, except that the record terminator is returned as the terminating character code (a numeric field) in RSTATD.

You can also obtain the same information using the %RSIZE (or %RDLEN) and %RTERM (or %RDTRM) intrinsic functions. %RSIZE returns the length of the last GET, GETS, READ, READS, or ACCEPT statement or one of the Synergy windowing API I/O operations, while %RTERM returns the terminating character of the last record read, without the overhead required by an external subroutine.

RSTAT returns a null character in *term_char* after a READS that does not require an explicit terminator is executed and the input field is full (after auto termination). For files, the terminator is always a null character.

The ACCEPT statement reads only one character or control code sequence per execution.

See also

- ▶ [RSTATD on page 9-274.](#)
- ▶ [%RSIZE on page 9-270.](#)
- ▶ [%RTERM on page 9-275.](#)

Examples

The following example gets the length and the terminating character of user input and displays that information to the screen.

```
.define TTCHN      ,1
record
    inarea          ,a500
    aflld            ,a5
    flng             ,d1
    inlng            ,d5
    trmchr           ,a1
    dcml             ,d3

proc
    open(TTCHN, i, "tt:")
    repeat
        begin
            display(TTCHN, "Enter an input string: ")
            reads(TTCHN, inarea) [eof=done]
            xcall rstat(inlng, trmchr)
            aflld = inlng, "zzzzx" [left:flng]
            xcall decml(trmchr, dcml)
            display(TTCHN, "... input was ", aflld(1, flng),
&                "long, trmchr was ", %string(dcml))
            forms(TTCHN, 1)
        end
    end

done,
    stop
end
```

RSTATD – Return numeric information about the last record read

```
xcall RSTATD(size[, term_code])
```

Arguments

size

The variable that will be returned with the size of the last record read. For READ and READS operations, the returned size doesn't include the record terminator that ended the input operation. **(n)**

term_code

(optional) The variable that will be returned with the code of the character that terminated the input operation, loaded left-justified over blanks. *Term_code* should be at least a three-digit field. **(n)**

Discussion

The RSTATD subroutine returns the size and terminating character of the last record read by a GET, GETS, READ, READS, W_DISP, W_DISP(WD_READS), or W_FIELDS(WF_INPUT) operation, or the character typed in at an ACCEPT statement.

The RSTATD subroutine is similar to the RSTAT subroutine, except that the record terminator is returned as the actual terminating character (an alpha field) in RSTAT.

RSTATD returns a zero in *term_code* after a READS that does not require an explicit terminator is executed and the input field is full (after auto termination).

The ACCEPT statement reads only one character or control code sequence per execution.

See also

[RSTAT on page 9-272.](#)

%RTERM – Return the terminating character of the last record read

character = %RTERM

Return value

character

The numeric value of the terminating character of the last record that was read. (i)

Discussion

%RTERM returns the numeric value of the character that terminated the most recently processed READ, READS, GET, or GETS operation; the last window I/O function; or the character typed in at an ACCEPT statement.

If you want, you can use the RSTAT subroutine to obtain the alpha form of the record terminator. %RTERM is equivalent to %RDTRM.

WIN

For Windows input, 13 is returned by %RTERM when the user presses ENTER. If you are using a service runtime (**db**s or **dbssvc**) or have redirected input from a file, %RTERM returns 10.

UNIX, VMS

If subroutine TTFLGS flag 4 is set and the character returned by the ACCEPT statement is an escape, %RTERM returns the value associated with the function key that was pressed, as shown in the table on [page 9-366](#).

For a terminal, pressing ENTER or RETURN results in 10 for %RTERM.

Examples

The following routine checks for function keys on a VT220.

```
record
    data,      a10
proc
    open(1, o, "TT:")
    xcall ttflgs(4000, 1)
    repeat
        begin
            reads(1, data)
```

System-Supplied Subroutines, Functions, and Classes

%RTERM

```
using %rdtrm select          ;TTFGLS 4 means we get a special
(256),                      ; keycode for function keys
    writes(1, "PF1 pressed")
(257),
    writes(1, "PF2 pressed")
(258),
    writes(1, "PF3 pressed")
(259),
    writes(1, "PF4 pressed")
(),                          ;All other function keys have
    stop                    ; their own unique values
endusing
end

end
```

RUNJB – Issue a system command or start a concurrent process

```
xcall RUNJB (program, [terminal], [pid], [subprocess][, io_flag])
```

Arguments

program

The file specification of the executable file to be run. (**a**)

VMS

terminal

(optional) A field that contains the device name or number of the terminal to be attached to the process. A negative value (-1) causes the job to be run as a detached process. *Terminal* is ignored on Windows and UNIX. (**a** or **n**)

pid

(optional) Returned with the process ID of the child or subprocess that runs concurrently with the initiating process. (**n**)

VMS

subprocess

(optional) A flag that determines what type of process is created: (**n**)

- 1** A subprocess is created.
- 0** A detached process is created.

Subprocess overrides system option #35 if it is present. It is not allowed on Windows or UNIX.

io_flag

(optional) Determines how SYSS\$INPUT, SYSS\$OUTPUT, and SYSS\$ERROR are set up when a process is created and also determines how that process is started. It can have the following values: (**n**)

- 0** The following are true:
 - ▶ SYSS\$INPUT is a mailbox.
 - ▶ SYSS\$OUTPUT is _NLA0: if *terminal* is less than zero; otherwise, it is the device.
 - ▶ SYSS\$ERROR is null.
 - ▶ The image for the created process is **SYSS\$SYSTEM:LOGINOUT.EXE**.
 - ▶ The following commands are written to the mailbox.
 - \$ RUN program
 - \$ LOGOUT

- ▶ The mailbox channel is then deassigned.
- ▶ DBL\$RUNJB_OUTPUT is an optional logical assignment of an output device.

non-0 The following are true:

- ▶ SYS\$INPUT, SYS\$OUTPUT, and SYS\$ERROR are all set to _NLA0: if *terminal* is less than zero; otherwise they are the terminal device.
- ▶ System option #35 is ignored.
- ▶ The image for the created process is the program name that is passed.

Io_flag is not allowed on Windows or UNIX.

Discussion

WIN

The RUNJB subroutine starts another process to run concurrently with the process that initiated it. All programs are run as background processes, so they should not be made to interact with the terminal.

If *program* is a Synergy program, it is not necessary to include the **dbf** command. (For example, “dbf myjob” can just be “myjob”.) *Program* cannot contain an environment variable, as in “DBLDIR:lpque.bat”. You must translate the environment variable using the GETLOG subroutine, unless **lpque** is a Synergy program. If RUNJB is used to execute a batch file, the command interpreter **start** keyword must precede the batch file (for example, “cmd.exe /c start c:\mybatch.bat”).

UNIX

The RUNJB subroutine starts another process to run concurrently with the process that initiated it. The STTY subroutine can be used to modify the terminal settings if the new program changes the terminal settings. All programs are run as background processes, so they should not be made to interact with the terminal.

If you try to run more than the maximum number of processes (which is determined by the configuration of your UNIX kernel and varies from system to system), you’ll get a “Too many processes” error (\$ERR_MAXPRC).

If *program* is a Synergy program, it is not necessary to include the **dbf** command. (In other words, “dbf myjob” can just be “myjob”.) *Program* cannot contain an environment variable, as in “DBLDIR:lpque”. You must translate the environment variable using the GETLOG subroutine, unless **lpque** is a Synergy program.

VMS

The RUNJB subroutine enables a Synergy Language program to start up an OpenVMS subprocess that is to execute concurrently with the initiating process. The subprocess continues to execute until either the parent process is terminated (for example, with a LOGOFF or the KILL subroutine) or the subprocess terminates itself. The subprocess has a base priority of four.

If you want a Synergy Language program to initiate a subprocess and wait for its completion before continuing execution, use the VMCMD or SPAWN subroutines.

If system option #35 is set, the created process is detached, unless this setting is overridden by a *subprocess* value of 1. Initiating a subprocess on another terminal requires both the DETACH privilege and that the terminal not be logged into the system or otherwise allocated.

Because RUNJB requires the command line interpreter, if you are using x/ServerPlus, you cannot XCALL this routine from within a shared image unless it creates another detached process, in which case it may be used.

If the *terminal* argument is numeric, it is translated into its corresponding terminal device name. If this argument is not passed, the calling program's internal TNMBR value is used.

Examples

The example below reads the name of a program that is to be run concurrently as a separate process on UNIX or OpenVMS.

```
.define TTCHN      ,15
record
    program        ,a64
    tty            ,d3

proc
    open(TTCHN, o, "tt:")
    display(TTCHN, "Which program do you want to run in background?")
    reads(TTCHN, program, done)
    tty = 0                      ;Terminal # ignored on UNIX
    onerror ($ERR_MAXPRC)nope, oops
    xcall runjb(program, tty);Start program
    offerror
    display(TTCHN, program, "has been started in background")
    forms(TTCHN, 1)
    close(TTCHN)
    stop

nope,
    writes(TTCHN, "Too many processes")
    goto done
```

```
oops,  
    display(TTCHN, "Cannot execute: ", program)  
    forms(TTCHN, 1)  
    goto done  
  
done,  
    stop  
end
```

%RVSTR – Search for the last occurrence of a substring within a string

position = %RVSTR(*start*, *string*, *substring*[, *position2*])

or

xcall RVSTR(*start*, *string*, *substring*, *position2*)

Return value

position

The position of the rightmost occurrence of *substring* within *string*. **(i)**

If any of the following conditions are true, the result is 0:

- ▶ *Start* is less than or equal to 0.
- ▶ *String* is null.
- ▶ *Substring* isn't found within the search range of *string*.
- ▶ *End* is greater than the length of *string*.

If *substring* is null but *string* is not, the result is 1. If *position2* is specified, the result is also returned in that variable.

Arguments

start

The position in the specified string at which the search will end. **(n)**

string

The string in which to search for the substring. **(a)**

substring

The substring for which to search. **(a)**

position2

(optional) A variable in which to return the starting position of the last occurrence of *substring*. If RVSTR can't find an exact match of the *substring* value, *position2* is returned with a value of 0 to indicate that the search failed. **(n)**

Discussion

Searching from right to left, %RVSTR returns the position of the rightmost occurrence of a substring within a string.

%RVSTR searches for *substring* in *string* from position one (the last character of *string*) through *start*. The length of *substring* must be less than or equal to the length of *string*.

To search from left to right and find the first occurrence of the string, use %INSTR. %RVSTR is equivalent to %BKSTR.

Examples

The example below checks whether a data record contains a specific string. The string may be in the record several times, but we're looking for the most recent entry. The function returns zero if the string is not found or the record position if the string is found.

```
function chk_rec
    a_data      ,a
    a_string    ,a

proc
    freturn %rvstr(1, a_string, a_data)
endfunction
```

As a second example, assume your data division looks like this:

```
record
    codes          ,a30,  "help.add.sub.mul.div"
    loc            ,d3
    target         ,a3,   "sub"
```

The following subroutine returns a **loc** value of 18, because the last **d** appears at character position 18 in the string stored in the **codes** variable.

```
xcall rvstr(4, codes, "d", loc)
```

S_BLD – Construct a formatted string

```
xcall S_BLD(destination, [length], control, argument[, ...])
```

Arguments

destination

The variable that will be loaded with the formatted string. (**a**)

length

(optional) The variable that will be loaded with the length of the formatted string. (**n**)

control

The processing control string for the build. The format of the *control* argument is explained in the Discussion section below. (**a**)

argument

A list of up to nine arguments required by the control string. (**a** or **n**)

Discussion

The S_BLD subroutine constructs a formatted string.

If the first two characters of *control* are “%&,” Synergy Language assumes *destination* already contains the number of characters in *length*, and the output string is appended to the existing contents. Otherwise, *destination* is “emptied” to blanks and loaded starting at the first position.

The *control* string consists of one or more text and/or format segments. A text segment is all of the characters between format segments. Text segments are copied directly into the *destination* field.



To achieve results equivalent to a Synergy decimal to alpha conversion (format string of the type ZZZZX.XX), use a control string of “%0.0nd” where *n* is the desired precision.

A format segment uses the next *argument* from the S_BLD call and formats it into the *destination* field. A format segment has the following syntax:

```
%[justification size].[precision]][=type]
```

justification

(optional) A justification code. It must have one of the following values:

- Left-justified within the specified size
- + Right-justified within the specified size

If *type* is **A**, the default is left-justified. If *type* is **D**, the default is right-justified.

size

(optional) The minimum width of the formatted result field. If *size* is omitted, or if it is present, nonzero, and smaller than the size of the consumed argument, the size of the result field is determined by the contents of the consumed argument. If *size* is present and contains a leading zero, leading zeros are appended to the result. If *size* is 0, the result is determined by the contents of the consumed argument. Otherwise, all leading zeros are set to blanks.

precision

(optional) The fractional precision that is displayed. If *precision* is omitted, no fractional precision is displayed for a decimal field, and the number of significant digits in the fractional part of an implied-decimal field is the fractional precision in the format. If *type* is **A** and you specify a *precision* value, the fractional precision is ignored.

A number of fractional zeros equal to the value in *precision* are logically added to the decimal field if the following conditions are true:

- ▶ *Precision* is present.
- ▶ The *type* is **D**.
- ▶ The passed argument is not implied or integer.

Trailing zeros are placed in the formatted result if the following condition is true:

- ▶ *Precision* is present and contains a leading zero.

Finally, the passed argument is logically rounded to the place number stored in *precision* if the following conditions are true:

- ▶ *Precision* is present.
- ▶ The *type* is **D**.
- ▶ The passed argument is implied-decimal or implied-packed.
- ▶ The size of the fractional part of the argument is greater than *precision*.

=

(optional) The strip-processing flag. If the “=” is present, no leading strip processing occurs. If the “=” is not present, leading, nonsignificant zeros are stripped from numeric fields; trailing blanks are stripped from alpha fields; and trailing, nonsignificant zeros (except the first nonsignificant zero) are stripped from implied-decimal or implied-packed fields.

type

The type of the next argument to consume. It must have one of the following values:

- A** The consumed argument is accessed as an alpha type.
- D** The consumed argument is accessed as a numeric type.

A numeric type can be decimal, implied-decimal, integer, packed, or implied-packed, while an alpha argument accessed as numeric is considered decimal.

Examples

In the example below, **line** contains the following

“Page #27 8/16/1992”

Linelen is returned with a value of 49. The %& causes “8/16/1992” to be appended to the existing contents of **line**, beginning at position **linelen** + 1.

```
.define TTCHN                    ,1
record
  line                            ,a80
  linelen                        ,d2
  pagenum                        ,d3,    27
  day                            ,d2,    16
  mo                             ,d2,    8
  yr                             ,d4,    1992

proc
  open(TTCHN, o, "tt:")
  xcall s_bld(line, linelen, "Page #&d", pagnum)
  linelen = 40
  xcall s_bld(line, linelen, "%&%d/%d/%d", mo, day, yr)
  writes(TTCHN, line(1:linelen))
  close TTCHN
  stop
end
```

The following table contains examples of the various output based on variable type (decimal or implied-decimal), given the following:

```
ivar,  d9.2,  1502.20
var,   d9,    1502
```

Mask	ivar value	var value
%09.2d	0001502.2	0001502.0
%09.02d	001502.20	001502.00
%0.02d	1502.20	1502.00
%09.2=d	0001502.20	000001502.0
%09.02=d	0001502.20	000001502.00
%0.02=d	0001502.20	000001502.00

S_DETAB – Expand tabs

```
xcall S_DETAB(string, [length][, expand_length])
```

Arguments

string

The alpha text whose tabs will be expanded. (**a**)

length

(optional) The length of the string to be expanded. (**n**)

expand_length

(optional) The variable that will be loaded with the length of the expanded string. (**n**)

Discussion

The S_DETAB subroutine expands tabs “in place.” Tab stops are set to 8.

If you specify *length*, that number of characters in *string* will have embedded tabs expanded. If *length* is not present, or if it has a value of zero, the string to expand is the contents of *string* with trailing blanks stripped.

If *string* isn’t long enough to expand all of the tabs, *string* is not changed. If *length* and/or *expand_length* are passed, they are returned with a value of zero on an overflow.

Synergy Language assumes that no more than 80 tabs will be expanded.

Examples

```
.define TTCHN          ,1
record
    line                ,a80
    length              ,d2,   15
    explen              ,d2

proc
    open(TTCHN, o, "tt:")
    reads(TTCHN, line)
    xcall s_detab(line, length, explen)
    if (explen) then
        writes(TTCHN, line(1,explen))
    else
        writes(TTCHN, "An overflow has occurred!")
    close TTCHN
    stop
end
```


If the string input looks like this:

DOE,<tab>JOHN<tab>,27

and the first tab is three characters in length and the second tab is five characters in length, the resulting string is as follows:

DOE, JOHN ,27

S_MATCH – Match a string

`xcall S_MATCH (match_string, field, result)`

Arguments

match_string

The match control string. (**a**)

field

The field that S_MATCH will attempt to match. (**a**)

result

The variable that will be loaded with one of the following values: (**n**)

1 The match is successful.

0 The match is not successful.

Discussion

The S_MATCH subroutine performs a string match.

The *match_string* argument contains one or more match expressions, where a match expression is zero or more asterisks (*), followed by zero or more strings, quoted strings, or question marks (?). The following interpretations apply:

- *** Match zero or more characters.
- ?** Match any single character.
- “quoted string”* Match this string exactly.
- unquoted string* Match this string, ignoring uppercase and lowercase.

For the field “This is a test string”, here are some examples of what matches and what doesn’t:

Match string	Result
“**test*”	1
“test”	0
“string”	0
“*string”	1

To specify a quote character within the *match_string*, use two consecutive quotation marks (“”).



S_MATCH is affected by the *case* value of the LOCALIZE routine wherever the operation depends on (or is independent of) the case of a character. It is also affected by the current operating system language locale.

Examples

The example below inputs a match control string, searches a file for that string, and displays the number of matches that were found.

```
.define TTCHN      ,1
.define NMCHN      ,2
record
    line            ,a80
    found           ,d1
    match_str       ,a80
    matches         ,d2

proc
    open(TTCHN, o, "tt:")
    open(NMCHN, i, "names.ddf")
    xcall flags(20, 1)
    display(TTCHN, "Enter a string to match: ")
    reads(TTCHN, match_str)
    repeat
        begin
            reads(NMCHN, line) [eof=done]
            xcall s_match(match_str(1:%trim(match_str)),
&                line(1:%trim(line)), found)
            if (found)
                begin
                    incr matches
                    writes(TTCHN, line)
                end
            end
        end
    done,
    display(TTCHN, %string(matches), " matches found.")
    close TTCHN
    close NMCHN
    stop
end
```

If **names.ddf** contains the following data and the match string is “B*”, three matches will be found:

```
Ben      ,Jones
Beverly ,Jones
Burt     ,Jones
```

S_PARSE – Parse a string and return information about it

```
xcall S_PARSE (string, start, dimension, item_position, item_length, item_type, #items, end  
&               [, no_quote])
```

Arguments

string

The string to parse. **(a)**

start

The beginning parse position within *string*. **(n)**

dimension

The dimension of the item position, item length, and item type arrays. **(n)**

item_position

The first element of the item position array. **(n)**

item_length

The first element of the item length array. **(n)**

item_type

The first element of the item type array. **(n)**

#items

The variable that will be loaded with the number of items parsed. **(n)**

end

The variable that will be loaded with either the ending parse position (if *string* contained more than *dimension* items) or zero (if all items were parsed). **(n)**

no_quote

(optional) Overrides the default handling of quote characters. **(n)**

Discussion

The S_PARSE subroutine parses a string and loads three arrays with information about each item (or token) in the string.

String is an alphanumeric literal or variable. Each item in string will be parsed, and the item characteristics will be loaded into *item_position*, *item_length*, and *item_type*.

The starting position (base one), the length, and the type of each token are loaded into the item position, item length, and item type arrays, respectively. Each array must have *dimension* elements or more. In other words, *item_position*(1), *item_length*(1), and *item_type*(1) contain the arguments for the first item within *string*; *item_position*(2), *item_length*(2), and *item_type*(2) contain the arguments for the second item; and so forth.

The type of each item is coded as follows. The I_ mnemonics are defined by the compiler.

Item	Mnemonic	Description
1	I_ANUM	Case-insensitive alpha character, followed by zero or more case-insensitive alphanumeric characters.
2	I_IDENT	Case-insensitive alpha character, followed by one or more case-insensitive alphanumeric, dollar sign (\$), or underscore (_) characters, with at least one of the characters being a dollar sign or an underscore.
3	I_DIGIT	One or more decimal digits.
4	I_FIXED	One or more decimal digits, followed by a period (.), followed by one or more decimal digits.
5	I_SPACE	One or more spaces and/or tabs.
6	I_SQUOTE	String enclosed in single quotation marks.
7	I_DQUOTE	String enclosed in double quotation marks.
8	I_SPECIAL	Any single character that is not part of one of the other item types.

Parsing continues either until all items within *string* have been parsed or until *dimension* items have been parsed. If *dimension* items are parsed and *string* still contains more items, *end* is the base one position within *string* at which the next item begins. (In other words, passing *end* as *start* on another S_PARSE call will continue the parsing.) If all of the items within *string* are parsed, *end* is returned with a value of zero.

When a quoted string item is parsed, the *item_position* array element is the position of the first character that follows the double or single quotation mark, and *item_length* is the length of the item up to the closing quotation mark. Thus, the delimiters of a quoted string are the only characters within *string* that aren't enclosed in one of the items.

Note that quoted strings can be implicitly terminated by the end of *string*, and that it is possible to have a quoted string with a length of zero (two successive quote characters).

If you want successive quotes in a string to represent a single occurrence of that quote (for example, 'O"Leary' to represent "O'Leary"), the calling program must detect successive occurrences of I_DQUOTE or I_SQUOTE items. In particular, if one of these items is the last item parsed, and

more items are on the line, the item should be “pushed back” before the next call to S_PARSE. In other words, set *start* to the following rather than *end* for the next call, and don’t process the last item on the current call:

item_position(dimension) – 1

On completion, *#items* is returned with the number of items that were parsed.

If *no_quote* is passed and nonzero, S_PARSE returns I_SPECIAL for single and double quotation marks instead of I_SQUOTE and I_DQUOTE.



S_PARSE is affected by the *case* value of the LOCALIZE routine wherever the operation depends on (or is independent of) the case of a character.

Examples

```
.define TTCHN,1
record
    line      ,a20
    start     ,i4,  1
    dim       ,i4,  20
    pos       ,20i4
    len       ,20i4
    type      ,20i4
    items     ,i4
    end       ,i4
    ix        ,i4

proc
    open(TTCHN, o, "tt:")
    writes(TTCHN, "Enter a string to parse: ")
    reads(TTCHN, line)
    xcall s_parse(line, start, dim, pos, len, type, items, end)
    writes(TTCHN, %string(items) + " items parsed")
    if (end) then
        writes(TTCHN, %string(end) + " is the end position")
    else
        writes(TTCHN, "All items were parsed!")
        for ix from 1 thru items          ;Display contents of arrays
            writes(TTCHN, "item " + %string(ix) + " pos=" +
&          %string(pos(ix), "ZX") + " len=" +
&          %string(len(ix)) + " type=" +
&          %string(type(ix)))
        close TTCHN
    stop
end
```

Let's assume the following line is input:

FRED EARNS \$17/HR

The program above produces the following output (based on the recommended .DEFINES):

```
Enter a string to parse:
FRED EARNS $17/HR
9 items parsed
All items were parsed!
item 1 pos= 1 len=4 type=1
item 2 pos= 5 len=1 type=5
item 3 pos= 6 len=5 type=1
item 4 pos=11 len=1 type=5
item 5 pos=12 len=1 type=8
item 6 pos=13 len=2 type=3
item 7 pos=15 len=1 type=8
item 8 pos=16 len=2 type=1
item 9 pos=18 len=3 type=5
```

S_WRAP – Wrap a string

```
xcall S_WRAP (string, max_length, left_length, start, right_length, wrap)
```

Arguments

string

The string to wrap. (**a**)

max_length

The maximum length of the left wrap segment. (**n**)

left_length

The variable that will be loaded with the length of the left wrap segment. (**n**)

start

The variable that will be loaded with the start position within *string* of the right wrap segment. (**n**)

right_length

The variable that will be loaded with the length of the right wrap segment. (**n**)

wrap

The variable that will be loaded with one of the following values to indicate whether wrap processing occurred: (**n**)

0 Wrap processing could not be performed.

1 Wrap processing occurred.

Discussion

The S_WRAP subroutine logically divides a string into two components: a left segment that will fit on a line whose length is defined by *max_length*, and a right segment that contains the rest of the string. The incoming string itself is not modified, but the segment start positions and lengths returned are relative to *string*.

The incoming string is “split” at the rightmost space character whose position is less than or equal to *max_length*, creating a left and a right “wrap segment.” S_WRAP logically removes trailing white space from the left wrap segment and leading white space from the right wrap segment.

If no white space is present to the left of *max_length*, or if the length of the incoming string with trailing white space removed is less than *max_length*, *wrap* is zero, *start* and *right_length* are undefined, and *left_length* is the next character position following the space-stripped incoming string.

Examples

```

        .define TTCHN      ,1
record
    line      ,a80
    wpos      ,i4      ,20          ;Strings longer than 20 wrap
    llen      ,i4
    rbeg      ,i4
    rlen      ,i4
    wrap      ,i4
    start     ,i4      ,1
    len       ,i4

proc
    open(TTCHN, o, 'tt:')
    reads(TTCHN, line)
    len = %rdlen
    do
        begin
            xcall s_wrap(line(start:len), wpos, llen, rbeg, rlen, wrap)
            writes(TTCHN, line(start:llen))
            start = start + rbeg - 1
            len = rlen
        end
    until (.not.wrap)
    close TTCHN
    stop
end

```

Let's assume that the following string is input:

THIS IS AN EXAMPLE TO DEMONSTRATE THE S_WRAP SUBROUTINE

Wrap processing will generate the following line segments:

**THIS IS AN EXAMPLE
TO DEMONSTRATE THE
S_WRAP SUBROUTINE**

SDMS_SEL – Set file type access behavior

WIN, UNIX

xcall SDMS_SEL (*file_type*, *behavior*)

Arguments

file_type

One of the following file types:

- 0** Default order
- 1** Synergy ISAM
- 3** Alternate (C-ISAM)
- 4** Alternate (Btrieve)

behavior

One of the following behaviors:

- 0** Set and retain the selected file access behavior.
- 1** Set the selected file access behavior, one time only.

Discussion



If alternate file access mechanisms (such as C-ISAM or Btrieve) are not in use, SDMS_SEL is not necessary, because Synergy ISAM is the only type available.

The SDMS_SEL subroutine allows altering the default file access behavior when an alternate file access mechanism has been enabled. When no alternate file access mechanism has been enabled, SDMS_SEL is ignored.

The following “select” file operations are affected: OPEN, XCALL ISAMC, XCALL ISCLR, XCALL DELET, and XCALL RENAM.

The default behavior for all “select” operations that access an existing file is to first attempt to open the file as an alternate (as ordered above), and then to attempt to open the file as Synergy ISAM.

The default behavior for a create operation is to create the specified file as the first enabled alternate file type (as ordered above).

To alter the default behavior and explicitly specify a single file type, call SDMS_SEL with the appropriate *file_type* before calling the desired “select” file operation. Specifying a nonexistent *file_type* will set the default behavior.

When *behavior* is set to 0, all “select” operations that follow will access files according to the selected behavior. Call SDMS_SEL with *file_type* set to 0 to cancel this behavior. When *behavior* is set to 1, only the next “select” file operation will be affected; subsequent “select” operations will revert to the default behavior.



Currently, the only supported file interfaces are C-ISAM (a trademark of Informix Corporation), available for UNIX, and Btrieve (a trademark of Pervasive Software), available for Windows. However, this subroutine is designed to handle several additional file interfaces at once.

Examples

The following example converts a Synergy error number into a text string using the Synergy message file. The call to SDMS_SEL tells the OPEN to look for a Synergy ISAM file rather than checking for alternate file types.

```
.define MSGCHN          ,23

subroutine interpret_runerror
    error_number      ,n
    error_text        ,a

record rec
    fac               ,a3
    num               ,d5
    code              ,a10
    text              ,a200

proc
    xcall sdms_sel(1, 1)
    open(MSGCHN, i:i, "DBLDIR:syntxt")
    num = error_number
    read(MSGCHN, rec, "RNT" + ^a(num))
    error_text = text(2, 200)
    close MSGCHN
    xreturn
endsubroutine
```

SERIAL – Return the registration string of the current runtime

WIN, UNIX

xcall SERIAL(*number*)

Arguments

number

The variable that will be loaded with the registration string of the current Synergy runtime. (a)

Discussion

The SERIAL subroutine returns the registration string of the current Synergy runtime. (This is the unique registration string used on your License Agreement form, *not* your license number.)

A registration string is valid at 10 characters; however, to receive the fully qualified registration string, 12 characters are required. If *number* is longer than 12 characters, the data is loaded left-justified over blanks. If *number* is shorter than 12 characters, the data is truncated.

Examples

The following subroutine displays an application time-out message and instructs the user to call his or her developer for help.

```
subroutine timeout
    a_dev      ,a           ;Developer to call
    a_phone    ,a           ;Phone number to call

    .include "DEF:workvar.def" ;Include terminal channel number: TTCHN

literal msg
    msg      ,a*,    "Your system has expired. Call "
    at       ,a*,    " at "
    ser      ,a*,    ". Your registration string is: "

record
    rstr      ,a15           ;Formatted string ####-##-####-##
    reg       ,a12           ;Fully qualified registration string

proc
    xcall serial(reg)
    xcall s_bld(rstr,,"%a-%a-%a-%a",reg(1:4),reg(5:2),reg(7:4),reg(11:2))
    display(TTCHN, msg, a_dev, at, a_phone, ser, rstr)
    xcall cleanup           ;Clean up files, etc.
    xreturn
endsubroutine
```

%SET_PRIORITY_CLASS – Set the priority class for a process

WIN*status* = %SET_PRIORITY_CLASS (*class*)

Return value

*status*One of the following values: (**n**)**1** Successful**0** Not successful

Arguments

*class*The priority class for the process: (**i**)

D_ABOVE_NORMAL_PRIORITY_CLASS

Process that has priority above normal-priority but below high-priority class processes.

D_BELOW_NORMAL_PRIORITY_CLASS

Process that has priority above idle-priority but below normal-priority class processes.

D_HIGH_PRIORITY_CLASS

Process that performs time-critical tasks that must be executed immediately. The threads of the process preempt the threads of normal- or idle-priority class processes. An example is the Task List, which must respond quickly when called by the user, regardless of the load on the operating system. Use extreme care when using the high-priority class, because a high-priority class application can use nearly all available CPU time.

D_IDLE_PRIORITY_CLASS

Process whose threads run only when the system is idle. The threads of the process are preempted by the threads of any process running in a higher priority class. An example is a screen saver. The idle-priority class is inherited by child processes.

D_NORMAL_PRIORITY_CLASS

Process with no special scheduling needs.

D_BACKGROUND_BEGIN

Begin background processing mode. The system lowers the resource scheduling priorities of the process (and its threads) so that it can perform background work without significantly affecting activity in the foreground. The function fails if the process is already in background processing mode. This value is not supported on Windows Server 2003.

D_BACKGROUND_END

End background processing mode. The system restores the resource scheduling priorities of the process (and its threads) as they were before the process entered background processing mode. The function fails if the process is not in background processing mode. This value is not supported on Windows Server 2003.

Discussion

Because scheduled tasks on Windows Server 2008 and Windows Vista run at a below-normal priority class and tasks take much longer than on Windows Server 2003 and Windows XP, you may want to raise the priority on these systems back to the Server 2003 level. The %SET_PRIORITY_CLASS function enables you to do this. Raising the priority can only be performed by a member of the administrator group who has elevated privileges, so you should use the **dbprpriv** or **dbspriv** runtime and ensure that the user specified in the task scheduler can elevate to do this. Scheduled tasks are run under the administrator account by default.

The D_XXX_PRIORITY_CLASS values affect the CPU scheduling priority of the process. For processes that perform background work such as file I/O, network I/O, or data processing, it is not sufficient to adjust the CPU scheduling priority; even an idle CPU priority process can easily interfere with system responsiveness when it uses the disk and memory.

Processes that perform background work should use the D_BACKGROUND_BEGIN and D_BACKGROUND_END values to adjust their resource scheduling priorities. Processes that interact with the user should not use D_BACKGROUND_BEGIN.

If %SET_PRIORITY_CLASS is not successful, you can get extended error information by calling %SYSERR.

SETCTL – Modify the operation of control characters

VMS

```
xcall SETCTL (character, action[, character, ...])
```

Arguments

character

A field, literal, or record that contains the control character whose action is to be modified: **(a)**

'C' CTRL+C

'T' CTRL+T

'Y' CTRL+Y

action

An expression whose value specifies how the control character should operate: **(n)**

If character is 'C':

0 Ignore CTRL+C.

1, 2 Generate fatal error.

If character is 'T':

0 Disable CTRL+T status display.

1 Enable CTRL+T status display.

If character is 'Y':

0 Disable DCL break or quit character, ignore CTRL+Y.

1, 2 Disable DCL break or quit character, generate fatal error.

3 Enable DCL break or quit character, ignore CTRL+Y.

4, 5 Enable DCL break or quit character, generate fatal error.

Discussion

The SETCTL subroutine modifies the operation of the characters CTRL+C, CTRL+T, and CTRL+Y when they are input from the terminal.

The SETCTL subroutine affects only the execution of the program that invokes its action. The original settings are restored when the program is completed normally. If the program is aborted, the original settings may not be restored.

The following actions are equivalent to the specified SETCTL calls:

- ▶ Calling the FLAGS subroutine with flag 8 set is equivalent to calling SETCTL with CTRL+C action equal to 0 (ignore) or CTRL+Y action equal to 0 (DCL disable break, ignore).
 - ▶ Calling the FLAGS subroutine with flag 8 reset is the default state for Synergy Language programs and is equivalent to calling SETCTL with CTRL+C action set to 2 (generate fatal error) and CTRL+Y action set to 3 (DCL enable break, ignore).
 - ▶ Setting system option #16 is equivalent to calling SETCTL with CTRL+Y action set to 2 (disable DCL break, generate fatal error).
-

SETDFN – Set the default filename extension

```
xcall SETDFN([file_spec][, option])
```

Arguments

file_spec

(optional) A field, literal, or record that contains the filename extension to be used as the new default by the runtime. **(a)**

option

(optional) Flag to indicate where *file_spec* will be applied: **(n)**

- 0** The non-ISAM default filename extension (default)
- 1** The ISAM default filename extension
- 2** Both the ISAM and non-ISAM default filename extensions

Discussion

The SETDFN subroutine sets the default filename extension to be used by the runtime.

If *file_spec* is missing or null, the original default filename extension is restored.



Setting the default extension only works on your local machine. x/Server always uses the default extension.

VMS

The initial default filename specifications used by the runtime library are **SY\$DISK:[].DDF** for non-ISAM operations and **SY\$DISK:[].ISM** for ISAM operations.

SETLOG – Set the value of an environment variable

```
xcall SETLOG (logical, [translation], status)
```

Arguments

logical

The case-sensitive logical name of the environment variable to set. (**a**)

translation

(optional) The environment variable's new value. (**a**)

status

The variable that will be loaded with one of the following values: (**n**)

- 1** The routine was successful.
- 0** The routine was not successful.

Discussion

The SETLOG subroutine sets the value of an environment variable or logical device in the current environment. (See the “[Environment Variables](#)” chapter of *Environment Variables and System Options* for information about specific environment variables.)



For Toolkit-specific environment variables, the U_START routine must be called before SETLOG.

You can use the SETLOG subroutine to process DBLOPT options at runtime, which enables you to toggle runtime options on and off. (We recommend using %OPTION to set system options.)

If you don't pass a *translation* value on Windows or OpenVMS, the SETLOG subroutine unsets (deletes) the specified environment variable. If you don't pass a *translation* value on UNIX, the SETLOG subroutine clears the specified variable, but the variable remains defined.

SETLOG only affects the current environment and any child processes; when your program terminates, the environment variable is the same as when your program began. Typically, environment variables are only read by the runtime when it starts up; therefore, setting any of them with SETLOG only affects child processes. However, the following environment variables change in response to a call to SETLOG: TNMBR, DBLOPT, TERM (UNIX only), DBLCASE (UNIX only), ISAMC_REV (UNIX and Windows), and SCSCOMPR (UNIX and Windows).

VMS

Because SETLOG requires the command line interpreter, if you are using x/ServerPlus, you cannot XCALL this routine from within a shared image.

You can use the SETLOG subroutine to create search list logicals. To do this, separate the individual elements of the list with commas within the translation value. If you want to create an ordinary logical that contains a comma, you must enclose at least the comma in quotation marks, just as you would with the DEFINE command in DCL.

For example:

```
$ DEFINE DBLOPT 35", "43
```

or

```
$DEFINE DBLOPT "35,43"
```

could be coded as

```
xcall setlog("DBLOPT", "35', '43", status)
```

or

```
xcall setlog("DBLOPT", "'35,43'", status)
```

To include a quotation mark in the translation value, use two consecutive quotation marks in the value string:

```
xcall setlog("LOGNAME", 'He said ""Hello""')
```

To include quotation marks of the same type as the string delimiting quotation marks, you must include four quotation marks. Each pair designates one quotation mark in the string that builds into the code. For example, the above setting could be coded

```
xcall setlog("LOGNAME", "He said """"HELLO""""")
```

Examples

The example below sets system option #25.

```
xcall getlog("DBLOPT", optstr, optlen)
if (optlen)
  if (.not. %instr(1, optstr(1, optlen), "25"))
    begin
      .ifndef OS_VMS
        optstr = '' + %atrim(optstr) + ',25''
      .else
        optstr = %atrim(optstr) + ",25"
      .endc
      optlen = %trim(optstr)
      xcall setlog("DBLOPT", optstr(1, optlen), sts)
    end
```

The following example unsets the environment variable specified by **lognam**.

```
xcall setlog(lognam,, status)
```

SHELL – Execute the system command processor or create a subprocess

```
xcall SHELL ([shell], [command][, mode])
```

Arguments

shell

(optional) An expression indicating which shell to invoke. The following values are valid: **(n)**

Value	Windows	UNIX	OpenVMS
0	COMSPEC environment variable (standard MS-DOS shell)	Bourne shell (sh)	Current CLI
1	Ignored	C Shell	DCL
2 (default)	SHELL environment variable if it exists; else COMSPEC environment variable if it exists; else cmd.exe	SHELL environment variable if it exists; else sh	DECSHELL

command

(optional) A command line to be executed by the shell on Windows and UNIX, or a new CLI on OpenVMS. **(a)**

WIN

mode

(optional) One or more of the following: **(n)**

- D_GAINFOCUS** Restore focus when the calling Synergy application resumes.
- D_INHERIT** Allow subprocess to inherit handles such as standard input and output.
- D_MINIMIZED** Start the program minimized.
- D_NOACTIVATE** Don't give the program keyboard focus.
- D_NOCONSOLE** Prevent new console from being created for new processes. A new process inherits the parent's console if possible.
- D_NOWAIT** Let the program run asynchronously like RUNJB.
- D_NOWINDOW** Prevent new console from being created for new processes. No DOS or command prompt window is displayed, not even on the task bar.

Discussion

The SHELL subroutine invokes a shell as a subprocess, which enables you to run more than one program simultaneously.

If *command* is specified, the command line is executed by the specified shell. If *command* is not specified, the shell specified by *shell* is invoked. If no arguments are specified, the default shell is invoked. (On Windows, the command prompt window appears.) Control returns to the calling program when the shell subprocess is finished.

If *shell* is not specified, the SHELL subroutine looks for the SHELL environment variable and uses that setting if it finds it. If the SHELL environment variable is not found, the SHELL subroutine uses the Bourne shell on UNIX or the default shell specified by the COMSPEC environment variable on Windows. If COMSPEC is not found, **cmd.exe** is used.

If you try to run more than the maximum number of processes, a “Too many processes” error (\$ERR_MAXPRC) is generated.

UNIX, VMS

When using the Synergy windowing API, if screen attributes are set and a program calls the SPAWN or SHELL subroutines, escape sequences to turn off the current attributes are sent to the screen.

UNIX

The Synergy runtime expects certain terminal (**tty**) settings for proper terminal I/O functionality. If a program run in the shell changes these settings to an undesirable state, the STTY subroutine can be used to restore the proper settings.

Command cannot contain a Synergy environment variable, as in “DBLDIR:lpque”, unless it’s being used as an argument to one of the Synergex™-supplied programs or utilities (such as **dbr**, **db**s, **isutl**, and so forth). If it’s not an argument to one of these, you must translate the environment variable using the GETLOG subroutine, or use the UNIX form “\$DBLDIR/lpque” instead.

Due to shells of varying behavior, a shell alias or function contained in a command may not be recognized by the command execution shell. Instead, use command executables or executable shell scripts directly.

WIN

The default SHELL behavior is to start an interactive process with a window associated with it. The calling program is suspended until the requested task is completed. The *mode* D_NOWAIT specifies that no wait should occur. You can add the D_xxx options to combine any of the modes.



The shell set by COMSPEC does not return the exit status of the program run; therefore, if it follows SHELL, the XSTAT subroutine always reports zero.

D_NOACTIVATE and D_MINIMIZED are only effective when invoking a non-GUI program. D_NOCONSOLE eliminates the unnecessary console window that is created when invoking a GUI program. Note that when using D_NOCONSOLE, no command interpreter is invoked. Therefore, any commands that require shell functionality, like redirection, will not work. Use an OPEN statement with a pipe command instead.



On Windows, using the D_NOCONSOLE option with a command that is redirected to output may have unexpected results. Instead we recommend that you use a pipe command and READS to read through the output.

Command cannot contain a Synergy environment variable, as in “DBLDIR:lpque”, unless it’s being used as an argument to one of the Synergex-supplied programs or utilities (such as **dbr**, **dbs**, **isutl**, and so forth). If it’s not an argument to one of these, you must translate the environment variable using the GETLOG subroutine.

VMS

The SHELL routine creates an interactive subprocess with either the current controlling Command Language Interpreter (CLI) or with DCL or DECSHELL as the controlling CLI. If you specify *command*, Synergy Language attempts to spawn to that CLI. If you don’t specify *command*, the *shell* argument specifies the CLI. When you enter the variable to which you want to spawn, you must provide the full path name of the CLI, such as **SYS\$SYSTEM:DCL.EXE**.

Because SHELL requires the command line interpreter, if you are using *xjServerPlus*, you cannot XCALL this routine from within a shared image.

Examples

The following program invokes the Bourne shell on UNIX.

```
.define TTCHN,15
record
    dsh          ,d1                      ;Shell to invoke
    achar        ,a1
record ,x
    ash          ,a1                      ;Overlay for DISPLAY
```

```

proc
    open(TTCHN, o, "TT:")
    dsh = 0                                ;Set to 0 for Bourne shell
    onerror ($ERR_MAXPRC) nope, oops      ;Trap any errors
    xcall shell(dsh)                      ;Invoke the shell
    offerror
    display(TTCHN, "[Press Return to continue] ")
    reads(TTCHN, achar)
    stop

nope,
    writes(TTCHN, "Too many processes")
    goto done

oops,
    display(TTCHN, "Shell error or invalid shell: ", ash)

done,
    stop
end

```

The following program invokes the standard MS-DOS shell.

```

.define TTCHN,15
record
    dsh          ,d1                      ;Shell to invoke
    achar        ,a1

proc
    open(TTCHN, o, "TT:")
    dsh = 0                                ;Set to 0
    onerror ($ERR_MAXPRC) nope, oops      ;Trap any errors
    xcall shell(dsh)                      ;Invoke the shell
    offerror
    display(TTCHN, "[Press Return to continue] ")
    reads(TTCHN, achar)
    stop

nope,
    writes(TTCHN, "Too many processes")
    goto done

oops,
    display(TTCHN, "Shell error or invalid shell " +%string(dsh))

done,
    stop
end

```

^SIZE – Return the size of an expression

size = ^SIZE(*expression*)

or

xcall SIZE(*expression*, *size*)

Return value

size

The size of an expression. (**i**)

Arguments

expression

An expression whose size will be returned. (**a** or **n**)

Discussion

^SIZE returns the number of bytes in the data area referenced by the specified expression.

^SIZE is especially useful in subroutines that handle arguments with variable lengths, because you usually need to know the length of an argument to control how its contents are processed. For example, ^SIZE would be helpful in a subroutine that parses an alpha argument to decode a command line. Because it's evaluated at compile time, ^SIZE can be used on arguments referenced in the procedure division but not in the data division.

Subroutine arguments that have not been passed have a size of -1.

^SIZE of an array results in the size of one element of the array by default. You cannot use ^SIZE on arrays larger than 64K.

The following specifications, where *idx* and *ndx* are literals, are recognized as compile-time constants.

^SIZE(*dvar*[*ndx*])

^SIZE(*var*(*idx*))

^SIZE(*dvar*[*ndx*](*idx*))

If you compile with the /DECSCOPE or -s compiler option, specifying ^SIZE of a real array or a pseudo array returns the size of the whole array.

^SIZE is functionally identical to %LEN. We recommend that you use the ^SIZE format, which is more efficient than the SIZE subroutine.

Examples

Assume the following data division statements:

```
record
  aa      ,a10,  "abcdefghij"
  bb      ,d6,   102507
  cc      ,d1,   4
```

Here are some examples using ^SIZE:

Function	Length
^size(aa)	10
^size(aa(5,7))	3
^size(aa(6:-2))	2
^size(aa(bb(2,3),cc))	3

The following subroutine rotates the characters in **what** to the left. We used the ^SIZE function because there's no way of knowing in advance how long **what** will be.

```
subroutine rotat
  what      ,a
record
  first     ,a1

proc
  first = what(1, 1)
  what = what(2, ^size(what))
  what(^size(what):1) = first
  xreturn
end
```

SORT – Provide a callable interface to DBLSORT

VMS

```
xcall SORT(commands, output_file, dblsort_status[, vmssort_status])
```

Arguments

commands

A field, literal, or record that contains the DBLSORT commands. **(a)**

output_file

Returned with the full file specification of the output file that is created. **(a)**

dblsort_status

Returned with the status of the DBLSORT utility. **(n3)**

vmssort_status

(optional) Returned with the return status from OpenVMS SORT calls performed by DBLSORT. **(n)**

Discussion

The SORT subroutine provides a callable interface to the DBLSORT utility.

A description of the format specification of *commands* is also available with your Synergy Language distribution. Include this description in your program using the following statement:

```
.include "SORTPARMS" LIBRARY "SYS$SHARE:DBLTEXT"
```

The output file specification is moved to *output_file* according to the rules for moving alpha data.

If the *dblsort_status* return values are negative, they are documented in the file **sorterrs.dbl**, which is included in your Synergy/DE distribution. Otherwise, they are Synergy/DE error numbers, which are provided in the [“Error Messages”](#) chapter of *Synergy Language Tools*.

Vmssort_status should be large enough to contain an OpenVMS error code. The OpenVMS SORT status is moved to *vmssort_status* according to the rules for moving numeric data. *Vmssort_status* values are listed in the *OpenVMS System Messages: Companion Guide for Help Message Users*.

The SORT subroutine uses channels 101 through 108 for I/O operations. Programs that call the subroutine directly should not use these channels. If channels 101 through 108 are in use, the SORT subroutine uses alternative channels.

The SORT subroutine can also be called as a function.

Examples

Refer to the **sortparms.dbl** file included with your Synergy Language distribution for an example using the SORT subroutine. This file is located in SYNERGYDE\$ROOT:[DBL.DBLSTARLET] on OpenVMS and in SynergyDE\dbl\vaxdbl on Windows and UNIX.

SPAWN – Execute a command or program

```
xcall SPAWN(command [, mode])
```

Arguments

command

An expression that contains the command string to be processed. (**a**)

WIN

mode

(optional) One or more of the following, combined using .bor. or a vertical bar (|): (**n**)

D_DETACHED	Force a detached process when SPAWN is called from the db s service runtime running in a console window.
D_GAINFOCUS	Restore focus when the calling Synergy application resumes.
D_INHERIT	Allow the subprocess to inherit handles such as standard input and output.
D_MINIMIZED	Start the program minimized.
D_NOACTIVATE	Doesn't give the spawned program keyboard focus.
D_NOCONSOLE	Prevent new console from being created for new processes. A new process inherits the parent's console if possible.
D_NOWAIT	Let the program run asynchronously like RUNJB.
D_NOWINDOW	Prevent new console from being created for new processes. No DOS or command prompt window is displayed, not even on the task bar.

UNIX

mode

(optional) An expression that indicates whether **tty** will be reset. It can contain the following values: (**n**)

0	Do not reset tty .
1	Reset tty . (default)

Discussion

WIN

The SPAWN subroutine executes a program directly without using the command processor (**cmd.exe**) and returns control to the calling program upon exit. The SPAWN subroutine executes the *command* string as if the string were typed directly at the command prompt window. The calling Synergy Language program waits until the command is finished before it continues. The return value of the program that you run is set for the XSTAT subroutine.

The default SPAWN behavior is to start an interactive process with a window associated with it. The calling program is suspended until the requested task is complete. The mode **D_NOWAIT** specifies that no wait should occur.

The **D_NOCONSOLE** option prevents a new independent console from being created; it does not prohibit a DOS window from being displayed. To keep the DOS window from appearing, combine **D_NOCONSOLE** with **D_MINIMIZED**. For example:

```
xcall spawn("my_prog.exe", D_NOCONSOLE+D_MINIMIZED)
```

If you try to run more than the maximum number of processes, you'll get a "Too many processes" error (**\$ERR_MAXPRC**).

Command redirection requires you to use the command processor or the **SHELL** subroutine.

The SPAWN subroutine does not run Windows' internal commands such as **dir** or **copy** (since they are command processor functions); you must use the **SHELL** subroutine for these commands.

Command cannot contain an environment variable, as in "**DBLDIR:lpque.bat**". You must translate the environment variable using the **GETLOG** subroutine.

UNIX

The SPAWN subroutine executes a command in a child process. It executes the *command* string as if the string were typed directly at the terminal. The calling Synergy Language program waits until the command is finished before it continues.

If you try to run more than the maximum number of processes (which is determined by the configuration of your UNIX kernel, you'll get a "Too many processes" error (**\$ERR_MAXPRC**).

Because terminal (**tty**) settings are controlled by the runtime, they are reset by default when SPAWN is executed to allow "normal" terminal functionality. If you need to retain the Synergy Language terminal settings, specify a *mode* reset value of 0 or use the **STTY** subroutine.

Command cannot contain an environment variable, as in "**DBLDIR:lpque**". You must translate the environment variable using the **GETLOG** subroutine.

SPAWN handles only simple shell meta characters (such as input and output redirection). Use the **SHELL** subroutine if you want to make full use of the shell meta characters.

Due to shells of varying behavior, a shell alias or function contained in a command may not be recognized by the command execution shell. Instead, use command executables or executable shell scripts directly.

VMS

The SPAWN subroutine executes a program and returns control to the calling program when the program exits. It executes the *command* string as if the string were typed directly at the terminal. The calling Synergy Language program waits until the command is finished before it continues.

The SPAWN subroutine executes a DCL command string. It sends the command string to the Command Language Interpreter (CLI) as input.

The second argument, *mode*, is ignored.

If you try to run more than the maximum number of processes (which is determined by the process limit argument (PRCLM), you'll get a "Too many processes" error (\$ERR_MAXPRC).

Because SPAWN requires the command line interpreter, if you are using *x/ServerPlus*, you cannot XCALL this routine from within a shared image.

The return status from the executed command (if available) can be obtained from %XSTAT. If the value for %XSTAT is -1, the process creation failed and the creation system error code can be found in %SYSERR.

Examples

The following program (on Windows) checks the current drive.

```
proc
    open(15, o, "tt:")
    writes(15, "Disk check:")
    xcall spawn("chkdsk")           ;Execute chkdsk
    close(15)
    stop
end
```

The following example (on UNIX) suspends its own execution while the spawned program runs, and then it resumes with the next statement.

```
.define TTCHN,15
proc
    open(TTCHN, o, "tt:")
    writes(TTCHN, "List of who's using the system:")
    onerror ($ERR_MAXPRC)nope, oops
    xcall spawn("who")             ;Execute the "who" command
    offerror
    writes(TTCHN, "Now back to our program...")
```

```
        xcall menu                                ;We get here after SPAWN is done
        close(TTCHN)
        goto done

nope,
    writes(TTCHN, "Too many processes")
    goto done

oops,
    writes(TTCHN, "Error creating subprocess")

done,
    stop
end
```

%STRING – Convert a value to a string

string = %STRING (*value* [, *format*])

Return value

string

A string representation of *value*. (**a**)

Arguments

value

The value to convert. (**n**)

format

(optional) A format string. (**a**)

Discussion

If *format* is not specified, %STRING converts *value* to a nonblank string according to the rules for moving data to an alpha destination. (See “[Assignment Statements](#)” on page 3-26 for a description of these rules.) If *format* is passed, %STRING considers it to be a format string and converts *value* according to the rules for formatting data. If *value* is a compile-time expression, %STRING is also evaluated as a compile-time expression.



%STRING is a useful alternative to ^A for displaying integer values, because ^A doesn't perform a data conversion like %STRING does.

Examples

The following function reads in a price from an inventory file and displays it on the screen.

```
function show_price
    a_chn      ,n
    a_item     ,a
    .include "INV:invent.def" ;Data layout (record inven) for inventory file
    .define ERROR,-1

    proc
        read(a_chn, inven, a_item) [err=invalid]
        writes(TTCHN, "Price: " + %string(price, "$$$, $$$.XX"))
        freturn (%true)
    invalid,
        writes(TTCHN, "Invalid inventory item code")
        freturn (ERROR)
    endfunction
```


STTY – Control terminal settings

UNIX

`xcall STTY (setting_flag[, seconds])`

Arguments

setting_flag

One of the following terminal setting controls: (**n**)

- 0** Restore **stty** attributes to their original settings (before Synergy Language modified them).
- 1** Set the Synergy Language standard **stty** settings.
- 2** Don't restore terminal attributes on the SPAWN or SHELL subroutines at runtime exit.
- 3** Restore terminal attributes on the SPAWN or SHELL subroutines and at runtime exit.
- 4** Enable **stty** of a serial device used in the OPEN statement.
- 5** Disable **stty** of a serial device used in the OPEN statement.
- 6** Change device open time-out to *seconds*.
- 7** Restore the terminal attributes on the SPAWN or SHELL subroutines even if the runtime is set to not affect terminal attributes.

seconds

(optional) The number of seconds for the runtime to wait to open a device before returning a "Device in use" (\$ERR_DEVUSE) error.

Discussion

The STTY subroutine sets or unsets the terminal settings.

You can reset setting 7 by calling STTY with setting 3.



For settings 3 and 7, terminal attributes are only restored on SPAWN if SPAWN's second argument is passed.

%SUCCESS – Determine if low-order bit is on or off

VMS

status = %SUCCESS (*expression*)

Return value

status

0 if the low-order bit of *expression* is off, or **1** if the low-order bit is on. (**i**)

Arguments

expression

Any valid numeric expression. (**n**)

Discussion.



We recommend using a .BAND. test of bit 1 rather than %SUCCESS.

Examples

This example function calls an OpenVMS system function to determine whether or not a file exists. It calls %SUCCESS to check the status of the return value from this system function.

```
function file_exists

external function
    lib$find_file    , ^val ;OpenVMS system function

record
    context          , i4    ;Arguments for lib$find_file
    return_file_spec , i4
    ffile            , i4    ;For the return value

proc
    ffile = %lib$find_file(filename, return_file_spec, ^ref(context),,,, )
    xcall lib$find_file_end(^ref(context))    ; Release context block
    if (%success(ffile)) then
        freturn (%true)    ;File exists
    else
        freturn (%false)   ;File doesn't exist
endfunction
```

%SYN_ATEXIT – Register a routine to be called when a service shuts down

```
return = %SYN_ATEXIT (address)

or

xcall SYN_ATEXIT (address)
```

Return value

```
return

1.
```

Arguments

```
address

The address of the routine to register (in other words, the output of a call to %XADDR). (i)
```

Discussion

%SYN_ATEXIT registers a routine to be called under the following circumstances.

On	The specified routine is called when
Windows	The program running as a service with dbssvc is stopped by the stop service requested from the Services Control Panel (or NET STOP <i>service_name</i>).
UNIX	A process is stopped by SIGHUP or SIGTERM(-15).
OpenVMS	A process is stopped using the \$FORCEX system service.

Examples

```
status = %syn_atexit(%xaddr("MY_ROUTINE"))
```

%SYN_CHARTOSTR – Convert C-style character pointer to alpha

```
length = %SYN_CHARTOSTR (pointer, alpha)
```

or

```
xcall SYN_CHARTOSTR (pointer, alpha)
```

Return value

length

The length of the original null-terminated string, not counting the null. (**n**)

Arguments

pointer

The character pointer that points to the null-terminated string you want to convert. (**D_ADDR**)

alpha

Returned with the converted alpha string. (**a**)

Discussion

%SYN_CHARTOSTR converts a C-style character pointer (char *) to a Synergy Language alpha type. This routine can be useful for dealing with C routines that are in external DLLs or linked into the runtime. If the C routine returns a pointer to a null-terminated string, this routine can be used to copy the contents to a Synergy alpha variable. The contents of the null-terminated buffer whose address is *pointer* are copied into *alpha*. The null termination is removed from the copy, and any remaining space in *alpha* is blank-filled. *Length* may be larger than the size of the buffer into which the string was copied, although %SYN_CHARTOSTR only copies up to the size of the buffer.

Examples

```
main strex
.align
stack record
    kernel32 ,D_ADDR           ;Handle to kernel32.dll
    ptr      ,D_ADDR           ;Pointer to environment
    curr     ,D_ADDR           ;Pointer to current entry
    len      ,D_NATINT         ;Length of an entry
    buf      ,a256             ;Buffer for conversion

proc
    open(1,o,"TT:")
    kernel32 = %dll_open("kernel32")
```

```

;
; GetEnvironmentStrings returns a pointer to a block of memory containing
; the environment settings for the current process. This data is a series
; of null-terminated strings, with an extra null at the end.
;
    ptr = %dll_call(kernel32, DLL_TYPE_WINAPI, "GetEnvironmentStringsA")
    curr = ptr
    while (len = %syn_chartostr(curr, buf))      ;Zero length means the end
        begin
            writes(1, %atrim(buf))                ;Display the entry
            curr += len + 1                        ;Advance to next, skip null
        end
    xcall dll_call(kernel32, DLL_TYPE_WINAPI, "FreeEnvironmentStringsA",
&        ptr)
    xcall dll_close(kernel32)
    stop
endmain

```

%SYN_ESCAPE_HANDLE – Escape characters in a string in a handle

status = %SYN_ESCAPE_HANDLE (*handle*, *rules*, *length*)

Return value

status

One of the following values:

- 0** Successful
- E_INVHDL** A memory handle whose length is 0 was passed

Arguments

handle

The memory handle that contains the string to escape. (**D_HANDLE**)

rules

The escape rules to use:

- XML_RULES** Use XML escape rules
- HTTP_RULES** Use HTTP escape rules

length

The length of the string and, upon return, the length of the escaped text. (**n**)

Discussion

%SYN_ESCAPE_HANDLE enables you to escape certain characters in a string within a memory handle so that an XML or HTTP parser can process them properly.

The following table lists the characters that will be escaped in an XML string and their corresponding escape codes:

Character	Description	Escape code
&	Ampersand	&
<	Less than sign	<
>	Greater than sign	>
'	Apostrophe	'
"	Double quotation mark	"

The following table lists the characters that will be escaped in an HTTP URI and their corresponding escape codes:

Character	Description	Escape code (hex)
0-31 (decimal)	Unprintable characters	%00-%1F
127 (decimal)	Delete	%7F
128-255 (decimal)	Non-ASCII characters	%80-%FF
\$	Dollar sign	%24
+	Plus sign	%2B
,	Comma	%2C
' '	Space	%20
"	Double quotation mark	%22
<	Less than sign	%3C
>	Greater than sign	%3E
#	Pound sign	%23
%	Percent	%25
{	Left curly brace	%7B
}	Right curly brace	%7D
	Vertical bar	%7C
\	Backslash	%5C
^	Caret	%5E
~	Tilde	%7E
[	Left square bracket	%5B
]	Right square bracket	%5D
`	Accent	%60

See also

- ▶ [%SYN_UNESCAPE_HANDLE](#) on page 9-336.
- ▶ [HTTP_NOESCAPE](#) in the “Environment Variables” chapter of *Environment Variables and System Options*

Examples

```
.include "DBLDIR:synxml.def"

record
    h          ,D_ADDR
    len        ,i4
    i          ,i4

proc
    open(2,o,"tt:")
    h = %mem_proc(DM_ALLOC+DM_STATIC,25)
    ^m(h) = 'hey$+, \ "<>#%{|\\^~[]hey'
    len = %trimz(^m(h))
    i = %syn_escape_handle(h,HTTP_RULES,len)
    writes(2,^m(h))
    len = %trimz(^m(h))
    i = %syn_unescape_handle(h,HTTP_RULES,len)
    writes(2,^m(h))
    close(2)
```


%SYN_FREECHN – Get the next available channel

channel = %SYN_FREECHN([*start*][, *end*])

Return value

channel

The number of the next available channel, or 0 if no channel is available. (**n**)

Arguments

start

(optional) The first channel to consider. (**n**)

end

(optional) The last channel to consider. (**n**)

Discussion

%SYN_FREECHN checks the channels between *start* and *end*, if specified, and returns the first available channel.

Examples

The example below checks the channels between 101 and 199 and returns the first available channel encountered.

```
record
    chn          , i4

proc
    chn = %syn_freechn(101, 199)
    if (.not. chn)
        writes(1, "no channels available")
```

%SYN_GETDIR – Get the current directory path

path = %SYN_GETDIR

Return value

path

The path for the current directory. (a)

Discussion

%SYN_GETDIR returns the path for the current directory.

Examples

The example below sets **path** to the current directory path.

```
record
    path      ,a255

proc
    path = %syn_getdir
```

%SYN_GETSTATE – Retrieve a Synergy state option

status = %SYN_GETSTATE(*request*, [*channel*], *backup_mode*)

or

xcall SYN_GETSTATE(*request*, [*channel*], *backup_mode*)

Return value

status

One of the following values: (n)

0 Successful

1 Not successful

Arguments

request

The following value: (n)

0 Return the current backup mode

channel

(optional) The channel of a file opened via x/Server for which the current backup mode is being requested. (n)

backup_mode

Returned with one of the following backup modes: (n)

0 Off

1 On

2 Pending

3 Not Allowed

4 Disabled

Discussion

%SYN_GETSTATE enables you to determine the current backup mode. If *channel* is passed, it indicates that the caller is requesting the current backup mode for the server machine on which the specified channel is open.

If %SYN_GETSTATE is run from a Synergy/DE Client installation, *backup_mode* is returned with a value of 4 (disabled), regardless of the backup mode on the shared machine.

SYN_REPORTEVENT – Write entries to the event log

```
xcall syn_reportevent([type], text)
```

Arguments

WIN	
<i>type</i>	
One of the following Microsoft Windows event log types:	
D_EVENTLOG_SUCCESS	A successful operation.
D_EVENTLOG_ERROR_TYPE	A significant problem the user should know about (for example, a loss of functionality or data).
D_EVENTLOG_WARNING_TYPE	A problem that is not immediately significant but that may cause future complications (for example, resource consumption).
D_EVENTLOG_INFORMATION_TYPE	An infrequent but significant successful operation (for example, successful loading of a major server service).
D_EVENTLOG_AUDIT_SUCCESS	A security event that occurs when an audited access attempt is successful (for example, a successful log-on attempt).
D_EVENTLOG_AUDIT_FAILURE	A security event that occurs when an audited access attempt fails (for example, a failed attempt to open a file).

text

The text to be written to the event log. (a)

Discussion

The SYN_REPORTEVENT routine enables a Synergy Language program to report errors and other information to the system log (the event log on Windows, **syslog** on UNIX, or the operator console on OpenVMS). It is most beneficial when used by services or detached programs.

Type is optional on UNIX and OpenVMS. If *type* is passed on those systems, it is ignored.

Examples

```
xcall syn_reportevent(D_EVENTLOG_ERROR_TYPE, "My program had an error")
```

%SYN_SETDIR – Set the current directory path

status = %SYN_SETDIR(*path*)

Return value

status

One of the following values: (**n**)

1 Successful

0 Not successful

Arguments

path

The path for the directory you want to set as the current directory. (**a**)

Discussion

%SYN_SETDIR does not support relative paths (for example, ..).

WIN

Although changing a directory to a UNC path is valid, doing so restricts you from subsequently spawning any command that uses the command interpreter **cmd.exe**, because **cmd** does not support a default directory of anything other than a mapped or **subst** drive. For example, you would not be able to spawn (or do an open pipe) of a **dir** command or spawn a **.bat** command procedure.

If %SYN_SETDIR is not successful, the error code is the system error code from %SYSERR.

Examples

This example sets the current directory path to C:\mydir.

```
record
    sts          ,i4

proc
    sts = %syn_setdir("C:\mydir")
    if (.not. sts)
        writes(1, "cannot change to required directory")
```

%SYN_SETSTATE – Set a Synergy state option

status = %SYN_SETSTATE(*option*)

or

xcall SYN_SETSTATE(*option*)

Return value

status

One of the following values: (n)

0	Successful
1	Not successful

Arguments

option

One of the following values: (n)

0	Suspend application execution on any attempt to DELETE, STORE, or WRITE to a Synergy ISAM file when backup mode is On. (default)
1	Signal an error on any attempt to DELETE, STORE, or WRITE to a Synergy ISAM file when backup mode is On.

Discussion

%SYN_SETSTATE enables you to alter the behavior of your application when update I/O is frozen because a backup is occurring. If the backup mode is On and *option* is 0, the application is suspended until the backup finishes. If the backup mode is On and *option* is 1, the runtime signals a “Backup mode is On” error (\$ERR_BACKPEND) on any attempt to DELETE, STORE, or WRITE to a Synergy ISAM file.

WIN

If the **DBLDIR:synbackup.cfg** file doesn’t exist, *status* is 0.

UNIX

If the SYNBACKUP environment variable is not set, *status* is 0.



The backup mode feature is not supported on systems running multiple versions of Synergy.

See also

- ▶ [SYNBACKUP](#) in the “Environment Variables” chapter of *Environment Variables and System Options*.
- ▶ “[The synbackup Utility](#)” in the “General Utilities” chapter of *Synergy Language Tools*.

%SYN_SYSErrrTt – Get the formatted system error message

text = %SYN_SYSErrrTt([*error_num*])

Return value

text

The formatted operating system error message from the last system error or the specified error number. (a)

Arguments

error_num

(optional) The number of the system error for which you want to return message text. (n)

Discussion

%SYN_SYSErrrTt looks at the last system error or the specified error number and returns the message text for that error. The number of the last system error can be returned by calling %SYSErrr.

If you want the text for the last system error, %SYN_SYSErrrTt must be called immediately following the ONERROR label of the trapped error. No intervening Synergy statements should be executed, or random results may occur due to other low-level calls affecting the system error code.

Examples

```
onerror openfail
open(1, o, "cust.ddf")
.
.
.

openfail,
text = %syn_syseerrtxt
display(15, "ERROR: " + %string(%ernum) + " system error: " + text)
```


%SYN_UNAME – Get the current username

```
user = %SYN_UNAME
```

Return value

user

The username of the user executing the program. (**a**)

Discussion

%SYN_UNAME returns the username of the user executing the program. If a blank username is returned, check %SYSERR for the system error code.

%SYN_UNESCAPE_HANDLE – Unescape escape codes in a string in a handle

status = %SYN_UNESCAPE_HANDLE (*handle*, *rules*, *length*)

Return value

status

Always returns **0**.

Arguments

handle

The memory handle that contains the string to unescape. (**D_HANDLE**)

rules

The unescape rules to use:

XML_RULES Use XML unescape rules.

HTTP_RULES Use HTTP unescape rules.

length

The length of the string and, upon return, the length of the unescaped text. (**n**)

Discussion

%SYN_UNESCAPE_HANDLE enables you to unescape escape codes in a string within a memory handle. See the table on [pages 9-324](#) and [9-325](#) for a complete list of XML and HTTP escape codes and their corresponding characters.

%SYN_UNESCAPE_HANDLE can also unescape a character in XML decimal or hexadecimal escape code format to its corresponding ASCII character representation. For example, it will unescape ` ` (decimal) or ` ` (hexadecimal) to a space character.

Examples

See the example for [%SYN_ESCAPE_HANDLE](#) on [page 9-324](#).

%SYN_XNAME – Get a function name from its segment address

WIN, UNIX

name = %SYN_XNAME (*address*)

Return value

name

The name of the function.

Arguments

address

The address of the segment for which you want to return a name. This address may be returned from %XADDR.

Discussion

%SYN_XNAME returns a function name for the specified segment address.

One use for this routine is in debugging areas of your code in which you have stored routine addresses. The D_METHOD subfunction of %E_INFO returns the address of the routine registered for the method specified. You can use %SYN_XNAME to translate that address into a routine name for debugging or informational purposes.

Examples

```
record
    adr      ,i4
    name     ,a30

proc
    adr = %xaddr("DATE")
    name = %syn_xname(adr)
    writes(1, "system error: = "+%hex(%syserr))
```

%SYSERR – Return the status code from the last system call

```
status = %SYSERR
```

Return value

status

The status code from the last system call performed by the runtime. (i)

Discussion

%SYSERR is a partial alternative to the ERROR subroutine.

Note that if an error is trapped and your program performs I/O operations or calls an XCALL that requires system service *before* it calls %SYSERR, when %SYSERR is finally called, it contains the “success” status code from the I/O operation or XCALL, rather than the failure code from the last error.

On .NET, %SYSERR is only used for the system error code for Synergy-supplied subroutines, functions, and I/O statements.



When a system error is detected, we recommend that you use %SYSERR to capture the error immediately and then store the error in a local variable for later processing.

Examples

```
reads(chan, data) [err=ioerr]
return

ioerr,
  writes(1, "system error: " = %hex(%syserr))
```

%SYSID – Return the ID of the system where the source is being compiled

machine = %SYSID

Return value

machine

The variable to be loaded with the system code of the current machine: (**a** or **n**)

003	SCO OpenServer
004	IBM AIX 5L 32-bit
009	HP-UX 32-bit
020	Sun Solaris 32-bit
021	HP Tru64 UNIX
028	Linux 32-bit (Red Hat or SUSE)
101	Windows 32-bit
104	Windows 64-bit (x64)
200	HP OpenVMS Alpha
202	HP OpenVMS I64
304	IBM AIX 5L 64-bit
309	HP-UX 64-bit (PA-RISC)
320	Sun Solaris 64-bit (SPARC)
420	SunSolaris 64-bit (x64/x86)
428	Linux 64-bit (x64)
509	HP-UX 64-bit (Itanium)

Discussion

%SYSID is evaluated at compile time as a literal value corresponding to the system where the source code is compiled. This allows it to be used for conditional compilations. For example, if a program is compiled on 32-bit Windows and run on SCO OpenServer, the return value would be 101 for 32-bit Windows.

Examples

The example below writes “Program compiled for Alpha VMS” if the source code is being compiled on HP OpenVMS Alpha.

```
.if (%SYSID .eq. 200)
    writes(1, "Program compiled for Alpha VMS")
.endc
```

The following example writes “Program not compiled for this machine” if the machine on which the program is currently running does not have the same system ID as the machine on which the source code was compiled.

```
xcall envrn(system, machine)
if (machine .ne. %SYSID)
    writes(1, "Program not compiled for this machine")
```

TFLSH – Flush the terminal buffer

UNIX, VMS

```
xcall TFLSH([channel[, mode]])
```

Arguments

channel

(optional) A channel opened to a serial device. **(n)**

mode

(optional) A nonzero *mode* value indicates that any outstanding buffered output characters should be purged. **(n)**

Discussion

The TFLSH subroutine flushes the terminal or serial device buffer. It has different functions on UNIX and OpenVMS.

UNIX

Normally, terminal output is buffered, and the terminal buffer is automatically flushed on each DISPLAY or WRITE statement. However, by setting system option #12 with the DBLOPT environment variable, you can buffer terminal output across several statements. You can then use the TFLSH subroutine to explicitly flush the terminal buffer.

The terminal buffer is also flushed under the following conditions:

- ▶ The terminal buffer is full.
- ▶ One of the following operations has occurred: READS, GETS, ACCEPT, SLEEP, XCALL TTSTS, XCALL WAIT, or %TTSTS.
- ▶ Error conditions exist.
- ▶ The end of the program is reached.

You can customize the size of the terminal buffer by setting system option #12 and the environment variable TBUF. Refer to [TBUF](#) in the “Environment Variables” chapter and [#12](#) in the “System Options” chapter of *Environment Variables and System Options* for more information about the TBUF environment variable and system option #12, respectively.

You can also specify a channel open to a serial device. In this case, the TFLSH subroutine can be used to wait for I/O to be sent to a device or to purge pending output.

A *mode* value of one is equivalent to runtime PURGE functionality setting. A *mode* value of zero is equivalent to CLOSE functionality except that the channel is not closed.

VMS

Terminal I/O is asynchronous. The TFLSH subroutine ensures that the previous asynchronous terminal I/O has completed.

Examples

The following example is part of an application that uses buffered terminal output on a UNIX system. This code should only be used when system option #12 is set, or it won't do anything useful. Without option #12, output is only buffered for each individual statement.

```
display(TTCHN, $scr_att(blink, bold), "Working ...", $scr_att(clear))
xcall tflsh                                ;Make sure message is displayed
xcall long_job
```

TIME – Return the current system time

```
xcall TIME(time)
```

Arguments

time

The variable that will be loaded with the current time. (**d**)

Discussion

The TIME subroutine returns the current time, as maintained by the operating system.

Synergy Language returns the time in 24-hour notation in the format *hhmmss*, where *hh* is the number of hours elapsed since midnight, *mm* is the number of minutes elapsed since the last hour, and *ss* is the number of seconds elapsed since the last minute.

If the length of the *time* variable is less than 6, an “Argument specified with wrong size” error (\$ERR_ARGSIZ) is generated.

Examples

If the time is currently 12 seconds past 7:43 in the evening, the following program displays the time as 19:43:12.

```
.define TTCHN,1
record
    time      ,d6
    fmt       ,a8

proc
    open(TTCHN, i, "tt:")
    xcall time(time)
    fmt = time, "XX:XX:XX"
    display(TTCHN, "The time is ", fmt)
    forms(TTCHN, 1)
    stop
end
```

%TIMEZONE – Return the time zone of the current machine

offset = %timezone([*timezone*][, *daylight*])

Return value

offset

The number of minutes east of the Greenwich Meridian. (**n**)

Arguments

timezone

(optional) Returned with the name of the current time zone as defined on the local system: (**a**)

WIN

For example, Eastern Standard Time, Eastern Daylight Time, Pacific Standard Time, Pacific Daylight Time, GMT Daylight Time, and so forth.

UNIX, VMS

For example, EST, EDT, PST, PDT, GMT, and so forth.

daylight

(optional) Returned with one of the following daylight savings time flags: (**n**)

0 Daylight Saving Time is not in effect.

nonzero Daylight Saving Time is in effect.

Discussion

%TIMEZONE returns the number of minutes east of the Greenwich Meridian in which the current machine is located and optionally returns the name of the current time zone and/or whether Daylight Saving Time is in effect. For example, if the current time is 5 hours west of Greenwich, the *offset* value would be -300. If the current time is 3 hours east of Greenwich, *offset* would be 180.

You can use %TIMEZONE as an argument to %DATETIME.

See also

[%DATETIME on page 9-80.](#)

Examples

In the example below, `%TIMEZONE` will return the number of minutes east of Greenwich for the current time zone and will fill **zonename** with the name of the current time zone as defined on the current system.

```
offset = %timezone(zonename)
```

%TNMBR – Return the terminal number

```
terminal = %TNMBR([service_flag])
```

or

```
xcall TNMBR(terminal[, service_flag])
```

Return value

terminal

The number of the terminal that's running the current program. (i)

Arguments

WIN

service_flag

(optional) One of the following flags: (n)

- 0** The program is running with **dbf.exe**. (default)
 - 1** The program is running with **dbf.exe** and as a service, or as a scheduled task and the user is not logged in.
 - 2** The program is running with **dfs.exe** or **dfssvc.exe**.
-

Discussion

TNMBR returns the terminal number that corresponds to the program's TT: device. A terminal number can be up to eight digits long. Synergy Language's interpretation of the terminal number depends on the operating system on which your program is running.

On all operating systems, the TNMBR subroutine first looks at the TNMBR environment variable and uses its value if it is set. If the TNMBR environment variable is not set, Synergy Language uses the methods described below to determine the terminal number.

WIN

If the TNMBR environment variable is not set, the TNMBR routine returns a number as follows:

- 0** The process is a logged-in user session.
 - 1** The job is running as a service, as a scheduled task and the user is not logged in, or using **dfs.exe**, **dfssvc.exe**, or **dbf.exe -n**.
-

UNIX

If the TNMBR environment variable is not set, the terminal number is determined by

- ▶ looking in the file **/etc/ttys**.

If the terminal name has the form */dev/dev_type/unit_number*, TNMBR searches the **/etc/ttys** file for *dev_type/unit_number*.

- ▶ assigning a number based on the name of the terminal:

0 The process is a console application.

-1 The job is running detached, regardless of the terminal device specification.

positive All other instances

Synergy Language calculates a number from the device name that is unique if it was found in **/etc/ttys**. The device name may vary, depending on the flavor of the operating system and the type of terminal controller used. The terminal number is not necessarily compatible with other platforms or with DIBOL terminal numbers. No assumptions should be made as to the correspondence between terminal numbers and terminal names, other than the uniqueness of a terminal number for a local system. For example, using the number based on the name of the terminal, */dev/tty12* results in 13, */dev/tty0* results in 1, and both */dev/tty1a* and */dev/tty1b* result in 2. The “number based on filename” method is the default method if the **/etc/ttys** file is not present and the TNMBR environment variable is not set. If a unique value is desired, the **/etc/ttys** file should be used. See [“Terminal Numbers Used by Synergy Language”](#) in the “Terminal Characteristics on UNIX” chapter of the *Professional Series Portability Guide* for more information on **/etc/ttys**.

VMS

If the TNMBR logical is not set, the TNMBR subroutine assigns a number as follows:

0 OPA0:

-1 The job is running detached, regardless of the terminal device specification.

-2 The process is a network process.

-3 The process is a batch process.

positive All other instances

Synergy Language calculates a unique *number* from the device name. The device type (for example, TT) may vary, depending on the type of terminal controller used. The terminal number is not necessarily compatible with other platforms or with DIBOL terminal numbers, and it may vary from release to release. No assumptions should be made as to the correspondence between terminal numbers and terminal names, other than the uniqueness of a terminal number for a local system.



Do not rely on static terminal numbers, especially in a Telnet environment. We recommend using %JBNO to generate session-specific filenames.

See also

- ▶ [TNMBR](#) in the “Environment Variables” chapter of *Environment Variables and System Options*.
- ▶ “[Using dbr or dbs as a scheduled task](#)” in the “Building and Running Synergy Language Programs” chapter of *Synergy Language Tools* for more information about scheduled tasks.
- ▶ The “[Terminal Characteristics on UNIX](#)” and “[Other OpenVMS-Specific Information](#)” chapters of the *Professional Series Portability Guide* for information about terminal numbering.

Examples

The following example creates a unique filename.

```
subroutine bld_tmp
.define TMPCHN      ,2
record
    term            ,d3
    tmpnam          ,a11,"tmpfile.XXX"
    ext             ,a3 @tmpnam + 7

proc
    xcall tnmbtr(term)
    ext = term,'XXX'
    open(TMPCHN, o, tmpnam)
    xreturn
endsubroutine
```

TRACEBACK – Print the current traceback

```
xcall TRACEBACK([filename])
```

Arguments

filename

(optional) A file specification to which traceback output will be written instead of being displayed to the screen. **(a)**

Discussion

The TRACEBACK subroutine prints out a current traceback of the call tree in a Synergy Language program just as the debugger would. This subroutine is used by the Synergy debugger.

If *filename* is specified but is invalid, or if the specified file cannot be created, the traceback output will be written to the screen.

%TRIM – Return the length of a variable without trailing blanks

length = %TRIM(*variable*)

or

xcall TRIM(*variable*, *length*)

Return value

length

The length of *variable* minus the number of trailing blanks. If the variable is blank, %TRIM returns a value of one. If the variable is null, %TRIM returns a value of zero. In other words, the statement

```
%trim(" ")
```

returns a value of one, while the statement

```
%trim("")
```

returns a value of zero. (i)

Arguments

variable

A variable whose length will be returned. (a)

Discussion

%TRIM returns the length of a variable minus the number of trailing blanks.

The difference between %TRIM and %TRIMZ is that with blank strings, %TRIM returns a value of one, and %TRIMZ returns a value of zero.

Examples

The following subroutine displays a prompt, gets the input, and returns the length of the actual data that was input. If no data is input, the return length is one.

```
subroutine input
  a_data  ,a
  a_prompt ,a
  a_len    ,d

proc
  display(TTCHN, a_prompt, ": ")
  reads(TTCHN, a_data, done)
  a_len = %trim(a_data)
```



```
done,  
    xreturn  
endsubroutine
```

The example below checks for valid input.

```
subroutine chk_inp  
    a_valid      ,a  
    a_status     ,d  
    .define TTCHN      ,1  
    record  
        input      ,a60  
        len        ,d2  
  
    proc  
        reads(TTCHN, input)  
        xcall trim(input, len)  
        reads(TTCHN, input)  
        xcall trim(input, len)  
        if ((len.eq.1) .and. (input.ne.' ') .and.  
    &      (input(1:len) .eq. a_valid)) then  
            a_status = %true  
        else  
            a_status = %false  
        xreturn  
    endsubroutine
```

%TRIMZ – Return the length of a variable without trailing blanks

```
length = %TRIMZ(variable)
```

or

```
xcall TRIMZ(variable, length)
```

Return value

length

%TRIMZ returns the length of *variable* minus the number of trailing blanks. If the variable is blank, %TRIMZ return a value of 0. If the variable is null, %TRIMZ also returns a value of 0. In other words, both of the following statements return a value of 0. (i)

```
%trimz(" ")
```

and

```
%trimz("")
```

Arguments

variable

A variable whose length will be returned. (a)

Discussion

%TRIMZ returns the length of a variable minus the number of trailing blanks.

The difference between %TRIM and %TRIMZ is that with blank strings, %TRIM returns a value of one, and %TRIMZ returns a value of zero.

Examples

The following subroutine displays a prompt, gets the input, and returns the length of the actual data that was input. If no data is input, the return length is zero.

```
subroutine input
  a_data    ,a
  a_prompt  ,a
  a_len     ,d

proc
  display(TTCHN, a_prompt, ": ")
  reads(TTCHN, a_data, done)
  a_len = %trim(a_data)
```

```
done,  
    xreturn  
endsubroutine
```

The example below checks for valid input.

```
subroutine chk_inp  
    a_valid      ,a  
    a_status     ,d  
    .define TTCHN      ,1  
record  
    input        ,a60  
    len          ,d2  
  
proc  
    reads(TTCHN, input)  
    xcall trimz(input, len)  
    reads(TTCHN, input)  
    xcall trimz(input, len)  
    if (len) .and. (input(1:len) .eq. a_valid)) then  
        a_status = %true  
    else  
        a_status = %false  
    xreturn  
endsubroutine
```

%TRUE – Return a value of one

%TRUE

Return value

%TRUE returns a value of 1. (**n**)

Discussion

%TRUE is a compile time function that returns a value of 1.

The %TRUE function, like the %FALSE function, helps you clarify the intentions of certain statements. It's much easier to understand what a statement is doing if you make a reference to the value %TRUE instead of the decimal literal 1.



You cannot use %TRUE if you're including the file **WND:tools.def** (which contains the flag definitions and value-control arguments required by the UI Toolkit) in your Synergy Language programs. Instead, we recommend that you use the TRUE identifier. If you're using UI Toolkit, you don't have to define TRUE, because it's already defined in **WND:tools.def**.

Examples

```
if (%true) then
    open(chn, u, file) [err=nofile]
else
nofile,
    open(chn, o, file)
```

%TRUNC – Truncate a numeric expression

whole_num = %TRUNC (*expression*)

Return value

whole_num

The whole-number portion of *expression*. The result has the same numeric data type as *expression*. If *expression* is implied-decimal data type, the result has no fractional precision. **(n)**

Arguments

expression

A numeric expression to be truncated. **(n)**

Discussion

The difference between %TRUNC and %TRUNCATE is that with implied-decimal arguments, %TRUNC returns a result with no fractional precision and %TRUNCATE returns a result with the same fractional precision as *expression*.

Examples

The following intrinsic function returns the value 33.

```
dvar = %trunc(33.5)
```

%TRUNCATE – Truncate a numeric expression

whole_num = %TRUNCATE (*expression*)

Return value

whole_num

%TRUNCATE returns the whole-number portion of *expression*.

The result has the same numeric data type as *expression*. If *expression* is implied-decimal data type, the result has the same fractional precision as *expression*, and the fractional precision is filled with zeros. (n)

Arguments

expression

A numeric expression to be truncated. (n)

Discussion

The difference between %TRUNC and %TRUNCATE is that with implied-decimal arguments, %TRUNC returns a result with no fractional precision and %TRUNCATE returns a result with the same fractional precision as *expression*.

Examples

The following intrinsic function returns the value 33.0.

```
%truncate(33.5)
```

TT_NAME_TO_NUMBER – Convert a terminal name to its number

VMS

xcall TT_NAME_TO_NUMBER (*name*, *number*)

Arguments

name

A string that contains the OpenVMS terminal name. (**a**)

number

Returned with the terminal number. (**n**)

Discussion

The TT_NAME_TO_NUMBER subroutine converts an OpenVMS terminal name to its equivalent Synergy Language terminal number.

The conversion algorithm is the same as that used by the TNMBR subroutine to determine the initiating terminal number.

TT_NUMBER_TO_NAME – Convert a terminal number to its name

VMS

```
xcall TT_NUMBER_TO_NAME (number, name)
```

Arguments

number

An expression that contains the Synergy Language terminal number. (**n**)

name

Returned with the OpenVMS terminal name. (**a**)

Discussion

The TT_NUMBER_TO_NAME subroutine converts a Synergy Language terminal number to its equivalent OpenVMS terminal name.

The conversion algorithm used by TT_NUMBER_TO_NAME is the inverse of that used by the TNMBR and TT_NAME_TO_NUMBER subroutines. Therefore, it is possible to convert an OpenVMS terminal name to a Synergy Language terminal number using either of those subroutines, and then to convert back to the terminal name using the TT_NUMBER_TO_NAME subroutine. However, the terminal device name produced by this subroutine is not necessarily identical to the original device name, although the terminal device name uniquely identifies the same device.

Name must be at least 10 characters long, to hold the complete string returned by this subroutine.

TTBRDCST – Enable a program to trap broadcast messages

VMS

```
xcall TTBRDCST (channel, option[, ^XTRNL (ast_routine) ])
```

Arguments

channel

An expression that evaluates to a channel number specified in a previous OPEN statement. (**n**)

option

One of the following actions taken: (**n**)

- 0** Broadcast messages appear at the terminal. This option is equivalent to the DCL SET TERMINAL/BROADCAST command.
- 1** Broadcast messages do not appear at the terminal. This option is equivalent to the DCL SET TERMINAL/NOBROADCAST command.
- 2** Broadcast messages do not appear at the terminal, and *ast_routine* is executed whenever a message is broadcast to the terminal.

ast_routine

(optional) A subroutine to be executed as an AST (asynchronous trap). AST routines that don't use the SLEEP statement or any Synergy Language I/O statements or XCALLs are supported.

Discussion

The TTBRDCST subroutine enables a program to “trap” messages broadcast to a terminal.

The *ast_routine* is called with two arguments: the channel number of the terminal to which the message was broadcast and the message text.



When `^XTRNL()` is used in an argument list, as in the syntax for this subroutine, the return value is passed to the subroutine by value, rather than by descriptor. Therefore, if you use

```
int4_var = ^XTRNL(ast_routine)
```

or

```
int4_var = %xaddr("ast_routine")
```

to get the address of a routine before using TTBRDCST, you must then use the following:

```
xcall ttbrdcst(channel, option, ^VAL(int4_var))
```

Broadcast message trapping is implemented using temporary mailboxes and the BRDCSTMBX terminal attribute. If you change the broadcast and BRDCSTMBX terminal attributes through a call to TTBRDCST and you don't reset them before you exit your Synergy Language program, the runtime resets the attributes on exit.

If you use the SPAWN subroutine, you must disable broadcast message trapping and close the terminal. Otherwise, the SPAWN fails.

TTCHAR – Return type, lines, and width of a file

VMS

```
xcall TTCHAR(channel, [type], [lines], [width][, attributes])
```

Arguments

channel

The channel on which the terminal is open. (**n**)

type

(optional) A field containing a string that identifies the type of the terminal opened on *channel*. (**a**)

lines

(optional) The variable that will be loaded with the number of lines per screen page. (**n**)

width

(optional) The variable that will be loaded with the number of characters per line. (**n**)

attributes

(optional) The variable that will be loaded with the decimal codes of other terminal attributes. (**a51**)

Discussion

The TTCHAR subroutine returns the type, number of lines per page, number of characters per line, and other attributes of a file.

The terminal settings are obtained from the system when the channel is opened. If the terminal characteristics change while the channel is open, TTCHAR still returns the original information from when the channel was first opened. If the terminal characteristics are set improperly, TTCHAR returns incorrect information.

The following table indicates the position within the *attribute* argument of each code that TTCHAR returns. A value of one indicates the presence of the attribute. A value of zero indicates the attribute's absence. See your *OpenVMS I/O User's Reference Manual* for detailed information about the meaning of each attribute.

Attribute	Position	Attribute	Position
SCOPE	1	TTSYNCH	27
ANSICRT	2	HALFDUP	28
DECCRT	3	XON	29

System-Supplied Subroutines, Functions, and Classes

TTCHAR

Attribute	Position	Attribute	Position
AVO	4	<reserved>	30
BLOCK	5	NOBRDCST	31
EDIT	6	BRDCSTMBX	32
REGIS	7	MBXDSABLE	33
LOWER	8	NOTYPEAHD	34
WRAP	9	ALTYPEAHD	35
MECHFORM	10	SETSPEED	36
MECHTAB	11	EDITING	37
PASSALL	12	INSERT	38
EIGHTBIT	13	FALLBACK	39
CMA	14	DIALUP	40
NOECHO	15	SECURE	41
LOCALECHO	16	DISCONNECT	42
ESCAPE	17	PASTHRU	43
REMOTE	18	SIXEL	44
MODEM	19	DRCS	45
AUTOBAUD	20	PRINTER	46
HANGUP	21	APP_KEYPAD	47
MODHANGUP	22	SYSPWD	48
CRFILL	23	TT_DECCRT2	49
LFFIL24	24	TT_DECCRT3	50
HOSTSYNC	25	TT_DECCRT4	51
READSYNC	26		

Examples

The following example returns the type of terminal, along with the number of vertical lines and the width of each line on the terminal.

```

proc
    open(1, o, "tt:")
    xcall sho_term(1)
end

subroutine sho_term
    tt_ch          ,d          ;Terminal channel

record
    term           ,a30        ;Terminal type
    lines          ,d3         ;Vertical lines for terminal
    width          ,d3         ;Width of line
    buff           ,a80
    bufflen        ,d2

proc
    xcall ttchar(tt_ch, term, lines, width)
    xcall s_bld(buff, bufflen,
&          "Your terminal is a %a, with %d lines of " +
&          "%d characters", term, lines, width)
    writes(tt_ch, buff(1:bufflen))
    xreturn
endsubroutine

```

TTFLGS – Set terminal-oriented flags

UNIX, VMS

`xcall TTFLGS (flags [, control])`

Arguments

flags

The terminal-oriented flags. Each flag is a single digit, and the digit positions are assigned ordinal numbers from right to left. Thus, the low-order digit is digit number one.

Synergy Language examines the flags value digit by digit to determine which flags are set, and hence, which functions are to be performed. For optimal performance, flags should be a zoned decimal field or literal. The Discussion section below contains a detailed description of the flags and their meanings. **(n)**

control

(optional) A value that controls how the individual digit positions in the flag specification value are interpreted. If *control* is not specified, each digit in *flags* is interpreted as follows: **(n)**

- 0** Reset the associated flag.
- 1** Set the associated flag.
- n* Leave the associated flag unmodified, where *n* is any other decimal number.

We recommend that you use the control value when setting and resetting your flags values. If *control* is present, its value is interpreted as follows:

- 0** Reset all flags whose corresponding digit position contains a nonzero number. Leave all other flags unmodified.
- 1** Set all flags whose corresponding digit position contains a nonzero number. Leave all other flags unmodified.
- 2** Set the *flags* variable to a value that reflects the current settings of zero if the associated flag is currently reset or to one if the flag is set. If the length of the variable is less than four characters, the settings are returned as in a decimal-to-decimal assignment. (See [“Moving decimal/packed data to a decimal/packed destination” on page 3-33](#) for more information about decimal-to-decimal assignments.)

Discussion

The TTFGLS subroutine sets terminal-oriented flags.

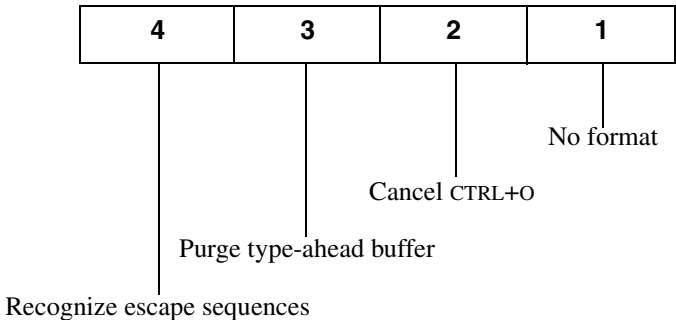
When program execution begins, all program-controlled flags are normally reset, and the program can alter them as needed. The flag settings are not preserved across program chaining. In effect, the STOP statement resets all of the flags as its last action, even if another program is being started. The flag settings in one program do not affect the settings in a program that is running concurrently from another terminal.

Synergy Language interprets the *flags* argument as if it were storing it to a **d4** variable. All rules that apply when storing to a decimal variable apply here. For example, if you specify

```
xcall ttflgs(01)
```

Synergy Language interprets your argument as 0001. (All flags corresponding to these zeros are cleared.)

Remember that the rightmost digit in the flag specification string corresponds to flag number one, as shown in the figure below. You don't have to specify a flag if that flag and all flags to its left have a value of zero.



A more detailed description of the flags is provided below.

Flag	Description
1	(OpenVMS) When flag 1 is set, the terminal driver does not perform data interpretation or formatting on data output by a DISPLAY, WRITES, or FORMS statement to the terminal. When flag 1 is reset, data interpretation and formatting do occur.
2	(OpenVMS) When flag 2 is set, CTRL+O (if it is in effect) is disabled before a DISPLAY, WRITES, or FORMS statement to the terminal is performed. When flag 2 is reset, CTRL+O is enabled.

Flag	Description
3	(OpenVMS) When flag 3 is set, the type-ahead buffer is purged before a READS or ACCEPT statement from the terminal is performed. When flag 3 is reset, the type-ahead buffer is not purged.
4	When flag 4 is set, valid ANSI escape sequences are recognized as terminators for READS, and they return the corresponding sequence code. (The escape character is not considered a valid terminator for READS.) When flag 4 is reset, ANSI escape sequences are not recognized. Flag 4 also understands the SS3 (143) and CSI (155) form of escape keys.

The table below specifies the return code number corresponding to each key on a VT220 keyboard.

Key group	Key name	Code	VT200	VT100
Keypad keys	PF1	256	PF1	PF1
	PF2	257	PF2	PF2
	PF3	258	PF3	PF3
	PF4	259	PF4	PF4
	KP0 (0)	260	0	0
	KP1 (1)	261	1	1
	KP2 (2)	262	2	2
	KP3 (3)	263	3	3
	KP4 (4)	264	4	4
	KP5 (5)	265	5	5
	KP6 (6)	266	6	6
	KP7 (7)	267	7	7
	KP8 (8)	268	8	8
	KP9 (9)	269	9	9
	ENTER (RETURN)	270	ENTER	ENTER
	MINUS (–)	271	–	–
	COMMA (,)	272	,	
	PERIOD (.)	273	.	.

Key group	Key name	Code	VT200	VT100
Cursor positioning keys	UP	274	UP ARROW	UP ARROW
	DOWN	275	DOWN ARROW	DOWN ARROW
	LEFT	276	LEFT ARROW	LEFT ARROW
	RIGHT	277	RIGHT ARROW	RIGHT ARROW
Function keys	F6	286	F6	
	F7	287	F7	
	F8	288	F8	
	F9	289	F9	
	F10	290	F10	
	F11	291	F11	
	F12	292	F12	
	F13	293	F13	
	F14	294	F14	
	HELP	295	HELP	
	DO	296	DO	
	F17	297	F17	
	F18	298	F18	
	F19	299	F19	
	F20	300	F20	
Editing keys	HOME/FIND	311	FIND	
	INSERT_HERE	312	INSERT HERE	
	REMOVE	313	REMOVE	
	END/SELECT	314	SELECT	
	PREV_SCREEN	315	PREV SCREEN	
	NEXT_SCREEN	316	NEXT SCREEN	
Other	UNKNOWN	511		

VMS

Terminal I/O is implemented using the OpenVMS \$QIO System Service. You can find additional information on the use of the flags set by TTFLGS in your *OpenVMS I/O User's Reference Manual*. The set flags modify the I/O function code passed to the \$QIO service as follows:

- ▶ Flag 4 sets the IO\$_ESCAPE modifier for Read QIOs, which instructs the terminal driver to validate incoming escape sequences.
 - ▶ Flag 3 sets the IO\$_PURGE modifier for Read QIOs, which allows the program to throw away any characters that may have been typed ahead but not yet read by the program.
 - ▶ Flag 2 sets the IO\$_CANCTRLO modifier for Write QIOs, which allows the program to force data to be displayed, even though the user may have previously pressed CTRL+O to turn output off.
 - ▶ Flag 1 sets the IO\$_NOFORMAT modifier for Write QIOs, which allows the program to bypass the formatting normally performed by the terminal driver, such as line wrapping. Setting this flag may allow performance gains when using the DMA transfer mode supported by certain terminal line controllers, such as DMF-32.
-

Examples

In the following example, the ACCEPT statement accepts either a normal key or a function key. For example, if the user presses the A key, **code** will contain a decimal 65 (the ASCII code for "A"). If the user presses the UP ARROW key, **code** will contain a decimal 275 (the code for an UP ARROW).

```
record
    code      ,d3

proc
    open(1, i, "TT:")
    xcall ttflgs(1000)      ;Turn on esc seq processing
    accept(1, code)
```

In the example below, the ACCEPT statement accepts either a normal key or a function key. However, since **achar** cannot hold the code for a function key, an escape character is returned to **achar**. The RSTATD subroutine is then called to retrieve the code for the function key. A single escape character is not a valid escape sequence and is not accepted when escape sequence processing is enabled.

```
record
    size      ,d1
    code      ,d3
    achar     ,a1
```

```
proc
  open(1, i, "TT:")
  xcall ttflgs(1000)      ;Turn on esc seq processing
  xcall rstatd(size, code)
```

In the following example, READS accepts a string of characters (up to the size of buffer). The string can be terminated by one of the normal terminator keys (return, line feed, form feed, or vertical tab), by filling the buffer (if the appropriate FLAGS digit is set), or by a function key. The RSTATD subroutine is then called to retrieve the code for the terminator. **Code** contains either the decimal code for one of the normal terminator keys or a function key code.

```
record
  size      ,d2
  code      ,d3
  buffer    ,a20

proc
  open(1, i, "TT:")
  xcall ttflgs(1000)
  reads(1, buffer)      ;Turn on esc seq processing
  xcall rstatd(size, code)
```

TTMBX – Associate a mailbox with a channel opened to a terminal device

VMS

`xcall TTMBX (status, channel, option[, mailbox])`

Arguments

status

Returned with a value that indicates the status of the operation: **(n)**

0 The operation failed.

n The system channel on which the mailbox was opened.

channel

A Synergy Language channel opened to a terminal. **(n)**

option

Specifies whether the mailbox will be assigned to or deassigned from a channel: **(n)**

0 Deassign a mailbox from a channel.

1 Assign a mailbox to a channel.

mailbox

(optional) Returned with the name of the mailbox created by the TTMBX subroutine. **(a)**

Discussion

The TTMBX subroutine associates a mailbox with a channel opened to a terminal type device.

You cannot spawn DCL commands when a mailbox is assigned to the controlling terminal.

%TTNAME – Return the terminal name

```
name = %TTNAME
or
xcall TTNAME(name)
```

Return value

name

The name of the terminal at which the current process is running. (a)

Discussion

We recommend that you use terminal names for unique process distinctions rather than using the TNMBR environment variable.

UNIX

If the terminal name has the form below, TTNAME returns *device_type/unit_number*:

/dev/device_type/unit_number

VMS

If the program is running as a batch job, the TTNAME subroutine returns the “null” device specification: “_NLAO.”

Examples

```
subroutine bld_tmp
  .define TMPCHN      ,2
  record
    term              ,d3
    tmpnam             ,a10, "tmpfile.XXX"
    ext               ,a3 @tmpnam + 7

  proc
    xcall ttname(term)
    ext = term
    open(TMPCHN, o, tmpnam)
    xreturn
  endsubroutine
```

%TTSTS – Determine whether input is pending

```
status = %TTSTS([channel])
```

or

```
xcall TTSTS(status[, channel])
```

Return value

status

%TTSTS returns a value of 0 under the following circumstances:

- ▶ No characters are currently pending on the referenced terminal or serial device.
- ▶ *Channel* is not specified and no channel is opened to the terminal or serial device.

If one or more characters are pending on the referenced terminal, %TTSTS returns either a value of 1 or the number of pending characters, depending on what's available on your system. (i)

Arguments

channel

(optional) The number of a channel opened to a terminal or serial device. (n)

Discussion

%TTSTS determines whether at least one input character is still pending on either the current terminal or another character device or serial device.

If *channel* is not specified, Synergy Language looks for the channel opened to the terminal (TT:).

You can use %TTSTS to inspect a terminal during a period of complicated and time-consuming processing. If %TTSTS finds that input is pending, your program can receive it and process it before continuing. By specifying the channel argument, you can check a terminal channel other than TT:. If *channel* is not a terminal or serial device, %TTSTS returns a value of 1.

When performing an input operation, program execution stops until the operation is completed and the information is transferred. In applications such as unsolicited user status facilities, the program should continue processing until terminal input (the status request) is available. When this input becomes available, the program can receive it and respond to it. If you don't use %TTSTS, the only way your program can test for user input is to issue an input request, which suspends program processing until the user responds.



On some UNIX systems, and for the keyboard on Windows, the status value is 1 or 0.



Each call to the TTSTS routine generates an operating system request, which consumes processing time. To avoid wasting processing time, we recommend that you use the TTSTS routine sparingly. If your program loops on the TTSTS routine, the required processing time affects the efficiency of your system.

Examples

The following subroutine prints a large report and checks after each page prints to see if the user typed a “Q” to quit printing.

```
subroutine prnt_rpt
  a_file      ,a
  a_data      ,a
  .define TTCHN      ,1
  external function
    get_pg      ,d
  record
    page      ,d3, 1
    input      ,a1

  proc
    while (%get_pg(page, a_data)) do      ;User function that returns 0
      begin                               ; when there is no more data
        xcall prnt_pg(a_data)
        if (%ttsts)
          accept(TTCHN, input)
          if (input .eq. 'Q')
            exitloop
          incr page
        end
      xreturn
    endsubroutine
```

UNPAK – Decompress numeric data

```
xcall UNPAK (record, field[, ...])
```

Arguments

record

The record that contains the compressed fields. (**a**)

field

One or more decimal fields to be restored. These fields must exist in the record and must be listed in ascending order as they appear in the record. (**d**)

Discussion

The UNPAK subroutine restores numeric data that has been compressed by the PAK subroutine.

Each packed field has the following length:

$(\text{number of characters}/2) + 1$

Examples

See the example for [PAK on page 9-236](#).

%UNSIGNED – Return the unsigned binary value of a numeric expression

value = %UNSIGNED(*expression*)

Return value

value

The unsigned binary value of *expression*. (**n**)

Arguments

expression

A numeric expression. (**n**)

Discussion

%UNSIGNED returns an unsigned value for the specified expression. For noninteger variables, the routine operates the same as %ABS. Literals are treated as integers. Negative 4-byte integers on 32-bit machines are returned as positive **i8** variables. Negative **i8** integers always generate an “Arithmetic operand exceeds maximum size” error (\$ERR_BIGNUM).

Examples

In the following example, **jvar** is set to 254.

```
record
    ivar      ,i1      ,-2
    jvar      ,i4

proc
    jvar = %unsigned(ivar)
    .
    .
    .
```

^VAL – Pass an argument by value or declare a value type function

```

^VAL (expression)
or
^VAL (argument)          ,i
or
external function
    funcname ,^VAL
    .
    .
    .
function funcname      ,^VAL[i)]

```

Arguments

expression

An argument to pass by value. (**n**)

argument

An argument of type **i** passed by value. (**i**)

funcname

The name of the function being declared. (**a**)

Discussion

^VAL is used to pass an argument by value, to declare an argument as type value, or to declare a function as returning an int by value.

^VAL can be used to pass arguments to C routines. If you use ^VAL when passing arguments to Synergy Language routines, the corresponding parameters in the Synergy Language routines must be declared as type ^VAL, with a data type of integer. When an argument is passed by value, the binary representation of the result of *expression* is passed.

Functions declared as type ^VAL are “true” functions with integer return values. If the value specified in the function’s FRETURN statement is not integer data type, it is converted to integer.

VMS

Any routine that passes floating-point arguments by value will not work on either the Alpha or the I64. This type of routine requires a C wrapper routine.

See also

- ▶ [“Compiler options”](#) in the “Building and Running Synergy Language Programs” chapter of *Synergy Language Tools* for information about the -X compiler option.
- ▶ [“^VAL functions”](#) on page 7-6.
- ▶ [“Parameter definitions”](#) on page 2-33.

Examples

This example shows a call to a ^VAL function in which some arguments are passed by value.

```
.include "$DVIDEF" library "DBLSTARLET"

external function
    sys$getdviw      , ^val

record
    stat              , D_NATINT
    group itmlst      , [4]i
        len           , i2
        code          , i2
        bufaddr       , i4
        retlen        , i4
    endgroup
    mbx_syschan       , i2
    mbx_name          , a16

proc
    open(1, 0, "tt:")
    xcall crembx(2, "message_mbx", 0, 132, 1320,,, 1) ;Create mailbox
    itmlst[1].len = ^size(mbx_name)                  ;Set up item list
    itmlst[1].code = DVI$_DEVNAM
    itmlst[1].bufaddr = ^addr(mbx_name)
    itmlst[1].retlen = 0
    clear itmlst[2].len, itmlst[2].code
    xcall getfa(2, "SCH", mbx_syschan)                ;Get mailbox channel
    stat = %sys$getdviw(, ^val(mbx_syschan),, ^ref(itmlst),,,,)
                                                    ;Pass arguments by value
                                                    ; and reference

    if (.not. (stat .band. 1))
        xcall lib$stop(^val(stat))
    writes(1, "Mailbox name: " + mbx_name)            ;Get the device name of the
                                                    ; mailbox directly into the
                                                    ; send message

end
```

^VARIANT – Return the value of an internal compiler variable

^VARIANT

Discussion

^VARIANT returns the value of an internal compiler variable set by the variant compiler option (**/VARIANT** on OpenVMS and **-qvariant** on all other systems).

The value set by the variant compiler option must be a positive number. For example, if you compile your program with the following commands, **^VARIANT** returns a value of 2:

```
dbl -qvariant=2 filename    (on Windows and UNIX)
```

```
dbl filename /variant=2     (on OpenVMS)
```

If the variant compiler option is not present, **^VARIANT** returns a value of 0.

Examples

This example determines whether or not code is compiled based on the value specified on the command line.

```
.if ^variant .eq. 2
.
.
.
;This code will be compiled
.else
.
.
.
;This code will not be compiled
.
.endc
```

%VERSN – Return the Synergy Language version number and operating system

```
version = %VERSN
```

or

```
xcall VERSN(version)
```

Return value

version

%VERSN returns the operating system and current version number of the Synergy runtime in one of the following formats:

- ▶ DBL/MS-WINDOWS Version *v.r.m*
- ▶ DBL/UNIX Version *v.r.m*
- ▶ DBL/OpenVMS Version *v.r.m*

where *v* is the version number, *r* is the major revision number, and *m* is the minor revision number. (a)

Discussion

%VERSN returns a textual string that identifies the Synergy Language environment in which the program is running.

%VERSN is functionally similar to the ENVRN subroutine, except that %VERSN returns a text string while ENVRN returns a decimal code, and %VERSN also provides the release level of the current Synergy Language runtime.

Version is stored left-justified over blanks. Each *r* and *m* component can only be a single digit.

Examples

The following example displays the operating system and Synergy Language runtime version on which the program is running.

```
.define TTCHN, 1
record
    ident      , a50

proc
    open(TTCHN, o, "tt:")
    ident = %versn
    writes(TTCHN, "Synergy system is: " + %atrim(ident))
    stop
end
```

VMCMD – Execute a DCL command

VMS

`xcall VMCMD (command, status, logflag)`

Arguments

command

The DCL command to execute. The command must not include the DCL command prompt (\$) and must not be more than 132 characters in length. (a)

status

A variable that will be loaded with one of the following values to indicate if the command was successfully executed: (n)

- 0** The command was successful.
- 1** The command was not an acceptable OpenVMS command or the command execution generated an error.

logflag

A variable or literal that controls whether or not the command execution is traced on the user's terminal: (n)

- 0** Disable logging.

nonzero Enable logging.

Discussion

The VMCMD subroutine enables a Synergy Language program to execute any acceptable DCL command, including a request to execute an entire command file.

The VMCMD subroutine is the same as the SPAWN subroutine, except that it also returns a status and controls whether or not command execution is traced on the terminal.

The DCL command is executed in the context of a subprocess. Therefore, the requesting process must be permitted to use at least one subprocess (which requires the PRCLM resource parameter). The use of a subprocess is mandatory under OpenVMS because a process cannot execute two images simultaneously (in this case, those of the requestor and his or her command).

The subprocess is constrained to operate within the resource limits of the parent. However, note that resources consumed by the subprocess reduce those available to the parent.



Because VMCMD requires the command line interpreter, if you are using x/ServerPlus, you cannot XCALL this routine from within a shared image.

The VMCMD subroutine uses the LIB\$SPAWN library routine to invoke the subprocess. (Refer to the *OpenVMS RTL Library (LIB\$) Manual* for more information about LIB\$SPAWN.) Upon return from LIB\$SPAWN, the exit status is made available to the XSTAT subroutine. If an error is indicated by status, the error can be further qualified by calling XSTAT.

See also

- ▶ [SPAWN on page 9-314.](#)
- ▶ [XSTAT on page 9-394.](#)

Examples

Assuming that “WORK:” is a logical name set up to reference the intended directory and device, the subroutine below has the same effect as typing the text of **cmd** on a logged-in terminal. The Synergy Language program pauses while the DCL interpreter executes the command. If successful, a file named **tmp.dta** is created. The **tmp.dta** file contains a listing of the most recent versions of all files with the extension **.ddf**.

```
record
    cmd      ,a*,    "dir/output=tmp.dta/col=1 WORK:*.ddf;0"
    stat     ,d4
    log      ,d1,    0

proc
.
.
.
xcall vmcmd(cmd, stat, log)
.
.
.
```

VMMSG – Get the text of an OpenVMS system message

VMS

```
xcall VMMSG([error_code], error_text)
```

Arguments

error_code

(optional) An expression whose value equals an OpenVMS system message code. **(n)**

error_text

The variable that will be loaded with the text associated with the desired error code. **(a)**

Discussion

The VMMSG subroutine obtains the text associated with a given OpenVMS error code. You will use it primarily within Synergy Language programs to display text after an OpenVMS system service (or a Synergy Language routine that directly interfaces to system services) has returned an error status.

If you omit the *error_code* argument, Synergy Language uses the status of the last non-Synergy Language subroutine. This status is the one that would be obtained if you issued a call to the XSTAT subroutine.

The VMMSG subroutine returns the full error code expansion, including any facility, severity level, and identifier emissions that are currently set. Additionally, any FAO arguments that may exist within the text of the message are not expanded.

For a more detailed discussion of the structure of OpenVMS messages, refer to your OpenVMS manual.



VMMSG has been superseded by the portable function %SYN_SYSErrTXT.

See also

[XSTAT on page 9-394.](#)

Examples

The example below uses the system screen library procedure to display text in a terminal-independent manner. If an error on the call to get screen library is detected, a branch to **label1** is taken, and the text for the detected error is returned in **errtxt**.

```
record
    message      ,a*,    "This is the message"
    row          ,d2,    12
    column       ,d2,    8
    flags        ,d2,    15
    error        ,d1

proc
    .
    .
    .
    xcall scr$put_screen(message, %val(row), %val(column), %val(flags))
    xcall xstat(status, error, ,,,)
    if (error .eq. 0) then goto label1
    .
    .
    .
label1,
    .
    .
    .
    xcall vmmsg(, errtxt)
    .
    .
    .
```

%VMS – Indicate if a program is being compiled on OpenVMS

status = %VMS

Return value

status

One of the following values: (i)

- 1** The program is being compiled on the OpenVMS operating system.
- 0** The program is not being compiled on the OpenVMS operating system.

Discussion

%VMS is evaluated at compile time.

%WAIT – Suspend program execution

```
event = %WAIT(parameters[, seconds])
```

or

```
xcall WAIT([seconds], parameters[, event])
```

Return value

event

%WAIT returns a numeric specification of the event that caused program execution to resume. (**i**)

Arguments

parameters

An expression that specifies the event(s) for which to wait. Each event is indicated by a nonzero number in a specific position, as follows: (**n**)

Position 4 Wait for a message.

Position 3, 2 Wait for terminal input at any of the keyboards opened by the program.

Position 1 Wait for the amount of time specified by the *seconds* argument to expire.

seconds

(optional) An expression that specifies the number of seconds for which program execution should be suspended. (**n**)

Discussion

%WAIT suspends program execution until a specified event occurs and indicates the event that caused execution to resume.

The *parameters* digits are numbered from right to left, as shown in [figure 9-4](#).

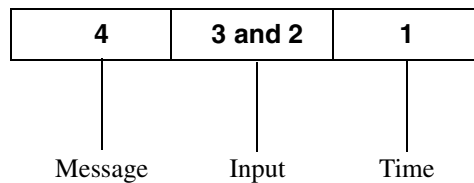


Figure 9-4. %WAIT parameters.

If *parameters* contains a value of 0001, for example, %WAIT suspends program execution for the specified number of seconds. If more than one event is specified (in other words, more than one digit in the *parameters* argument is nonzero), the first event to occur causes the program to resume execution.

Positions 2 and 3 perform the same function. If position 2 or 3 is set, the WAIT subroutine waits for terminal input on one or more channels opened to terminal or serial devices. (These settings perform identical functions.) The first five channels (in numerical channel order) opened to such devices are checked.

Except on some older UNIX systems, a WAIT subroutine that waits for time and keyboard input (xcall wait(1, 321)) tests for input every 1/10 second.

VMS

The WAIT subroutine has always performed a test for time and input every 1/10 second; however, this shows up as DIO in the process statistics for each interval.

The *seconds* argument is only required when digit position 1 in the *parameters* argument is nonzero.

Examples

The following example sets wait flag 4. The routine continues after the message manager receives a message for the suspended program.

```
xcall wait(, 4000)
```

The following example causes the program to sleep for five seconds before continuing execution.

```
xcall wait(5, 1)
```

The following example displays a “Press RETURN to continue” message, then waits for the user to press a record terminator.

```
open(1, o, "tt:")
display(1, "Press RETURN to continue")
xcall wait(, 300)
display(1, $scr_clr(line))
```

%WKDAY – Return the day of the week

```
day = %WKDAY ([date])
```

or

```
xcall WKDAY (day [, date])
```

Return value

day

One of the following values that represents the weekday: (**n**)

1	Sunday
2	Monday
3	Tuesday
4	Wednesday
5	Thursday
6	Friday
7	Saturday

Arguments

date

(optional) The date for which you want to determine the day of the week. This argument may be specified as either a numeric date or an alphabetic date. The default is the current date.

(**a** or **n**)

Discussion

%WKDAY returns a numeric value that identifies the day of the week either for the current day or for the date passed.

If *date* is passed as a numeric value, the format must be YYYYMMDD. If the date is passed as an alpha value, the format can be

```
[DD][-/][MMM][-/][YYYY][BC]
```

or

```
+|-n
```

The hyphen delimiters in the first format are optional, but they must be specified if leading components are omitted. Any omitted component defaults to the current value.

The second format indicates plus or minus *n* days from the current date (for example, +5 or -10).

See [%JPERIOD on page 9-198](#) for examples of valid dates.

If an invalid date is passed, an “Invalid date” error (\$ERR_INVDATE) is generated.

Examples

```
.define TTCHN,1
record
    today      ,d1
    day        ,7a9, "Sunday", "Monday", "Tuesday", "Wednesday",
    &          "Thursday", "Friday", "Saturday"

proc
    open(TTCHN, o, "tt:")
    xcall wkday(today)           ;Get day
    writes(TTCHN, "Today is" + day(today)) ;Display it
    close TTCHN
    xreturn
end
```

^X – Interpret a string as a hexadecimal number

hex = ^X(*expression* [, *size*])

Return value

hex

The integer value of an alpha *expression*, evaluated as a hexadecimal number. If *expression* is null, ^X returns a value of 0. (**i**)

Arguments

expression

The expression to interpret. (**a**)

size

(optional) A value that specifies the size of the resulting integer. Valid values are

- 1** The result is returned in an **i1** value.
- 2** The result is returned in an **i2** value.
- 4** The result is returned in an **i4** value.
- 8** The result is returned in an **i8** value.

Discussion

You can specify unquoted arguments with ^X. For example:

^x(F4)

If *expression* does not contain only valid hexadecimal digits (0 through 9 and A through F), a “Hexadecimal digits expected in argument (*string*)” error (\$ERR_HDIGXP) occurs.

By default, ^X produces an **i4** value if the return value fits in 32 bits or an **i8** value if the return value is larger than 32 bits. If *size* is specified, the value is returned in an integer of that size. If the value exceeds the specified size, it is truncated to fit.

An ^X value that fits in 32 bits but has the high bit set will be negative. If you want a positive value when a negative **i4** would normally be returned, you can specify a value of 8 in the optional *size* argument.



^X only returns the low 32 bits of the converted expression.

Examples

This example uses alpha input as data for a hexadecimal arithmetic computation. If the alpha variable contains any nonnumeric data other than a blank, a decimal point, or a sign (+ or –), a “Bad digit encountered” error (\$ERR_DIGIT) is generated.

```
subroutine math
  a_avar      ,a
record
  val1        ,d5
  val2        ,d5
  result      ,d10

proc
  result = (val1*val2) / ^x(a_avar)
  .
  .
  .
  xreturn
endsubroutine
```


%XADDR – Return the address of a routine

address = %XADDR (*subroutine*, [*elb*][, *notfound*])

Return value

address

The address of a function or external subroutine that can be used as the first argument to the XSUBR subroutine. The result is integer data type. Variables used for the return value should be declared as type D_ADDR, which is defined by the compiler as either **i4** or **i8**, depending on the system. (**i**)

Arguments

subroutine

The name of the subroutine whose address will be returned. (**a**)

elb

(optional) The shared image or ELB that contains the routine. (**a**)

notfound

(optional) If nonzero, a value that causes the return value to be 0 if the subroutine is not found. (**n**)

Discussion

%XADDR returns an integer value that can be used to access the specified subroutine. It is intended to be used in conjunction with the XSUBR subroutine when you need to invoke the same subroutine more than once.



You cannot use %XADDR to return the address of a class method.

WIN, UNIX

If *elb* is not already open, it is opened, along with any ELBs that it is linked against. You can use an environment variable to specify the location of the *elb*. For example you can pass “ELBDIR:*elb_name*” to %XADDR.

VMS

Elb is the name of an OpenVMS shared image. Only the filename section is used; all other elements are ignored. The name can be either the filename of a shared image located in SYSS\$SHARE, or a logical name that refers to a shared image, which may reside anywhere. A routine in a shared image will only be found if the shared image is explicitly specified or an OPENELB has been done on the shared image.

Normally, if the routine is not found, a “Cannot access external routine” error (\$ERR_RTNNF) is signaled. If *notfound* is passed as a nonzero value and the routine is not found, *address* is returned as 0, and no error is signaled. If the specified ELB or any linked ELB is not found, the runtime generates a “File not found” error (\$ERR_FNF).

See also

[XSUBR on page 9-397.](#)

Examples

```
record
    adr          , D_ADDR

proc
    adr = %xaddr("U_START", "WND:tklib.elb")
    if (adr)
        xcall xsubr(adr)                ;xcall U_START
```

XARGS – Return information about arguments passed to external routine

```
xcall XARGS ([arguments][, passed])
```

Arguments

arguments

(optional) Returned with the number of arguments passed to the calling routine. (**n**)

passed

(optional) An array of single-digit numeric fields that is returned with an indication of whether each defined argument was passed (1) or not passed (0). (**a**)

Discussion

The XARGS subroutine returns information about arguments passed when called within a Synergy Language external subroutine.

A digit within the passed argument is set to one if the corresponding argument of the external subroutine from which XARGS is called has been passed. Otherwise, the digit is set to zero. Therefore, *passed* can be redefined and subscripted such that passed(5), for example, indicates whether or not the fifth argument was passed.

Example

```
subroutine sub
record
  numargs  ,i4
  args     ,a5
  passed   ,5a1 @args

proc
  xcall xargs(numargs, args)
  if (numargs .and. passed(3))
    writes(TTCHN, "argument #3 was passed")
```

XSTAT – Get return value from last routine called

```
xcall XSTAT([return_value], [bit0], [severity], [message], [facility][, control])
```

Arguments

return_value

(optional) The variable that will be loaded with the return value of the last function or subroutine called. On OpenVMS, this argument is optional and corresponds to the full 32-bit return status value. It must be 10 digits wide to avoid overflow. **(n)**

bit0

(optional) A field that corresponds to the value of the low bit of the return value. **(n)**

severity

(optional) A field that corresponds to the value of bits 0 through 2 of the return value. **(n)**

message

(optional) A field (five digits wide to avoid overflow) that corresponds to the value of bits 3 through 15 of the return value. **(n)**

facility

(optional) A field (five digits wide to avoid overflow) that corresponds to the value of bits 16 through 27 of the return value. **(n)**

control

(optional) A field that corresponds to the value of bits 28 through 31 of the return value. **(n)**

Discussion

The XSTAT subroutine gets the return value from any of the following:

- ▶ The last non-Synergy Language subroutine or function (including C functions, OpenVMS library routines, and so forth)
- ▶ The last Synergy Language function of type ^VAL
- ▶ The last Synergy Language subroutine that XRETURNed a return value

The return value from a user-defined subroutine that does not XRETURN a return value is one.

The return value from a Synergy Language system-supplied subroutine or non-^VAL function is zero. The SPAWN and SHELL subroutines are exceptions to this rule. On Windows, SPAWN returns the exit status from the system call and SHELL returns the status of **cmd.exe**, not the program executed by **cmd.exe**. On UNIX and OpenVMS, both SPAWN and SHELL return the exit status from the system call, which is the return value from the shell.

VMS

Any of the arguments to the XSTAT subroutine can be passed as null arguments.

Examples

The following example verifies an ISAM file and stops the program if the ISAM file is corrupt.

```

subroutine chkism
    a_file      ,a
record vfycomd
    ,a*, "isutl -v "
    vfile      ,a40
record
    rtnval     ,d1

proc
    vfile = a_file
    xcall spawn(vfycomd)
    xcall xstat(rtnval)
    if (rtnval)
        begin
            writes(TTCHN, "ISAM file " + vfile + " corrupt")
            xcall cleanup
            stop
        end
    xreturn
endsubroutine

```

The following example illustrates how to use XSTAT to retrieve the ^VAL return value from an external subroutine. This return value is an integer value of the natural integer size of the machine. (That is, for a 32-bit machine, it is an **i4**; for a 64-bit machine, it is an **i8**.) See [“Writing Subroutines and Functions” on page 7-2](#) for additional information.

```

main
record
    retval     ,D_NATINT

proc
    xcall mult(3, 4)          ;Call mult subroutine
    xcall xstat(retval)      ;Retrieve the ^VAL return value
endmain

subroutine mult
    a_arg1     ,n
    a_arg2     ,n

```

System-Supplied Subroutines, Functions, and Classes

XSTAT

```
record
    result      ,D_NATINT

proc
    result = a_arg1 * a_arg2
    xreturn result      ;Return the result as a ^VAL value
endsubroutine
```

XSUBR – Invoke an external routine with the specified arguments

```
xcall XSUBR (subroutine, arguments[, ...])
```

or

```
value = %XSUBR (valfunction, arguments[, ...])
```

or

```
xcall xsubr (nonvalfunction, return, [arguments, ...])
```

Return value

value

The return value of the specified external function. (^VAL)

Arguments

subroutine

The name of the external routine to invoke or the value returned from %XADDR. (**a** or **n**)

arguments

The arguments to the specified external routine. (**a** or **n**)

valfunction

The name of the external function to invoke or the value returned from %XADDR. This must be a ^VAL function. (**a**)

nonvalfunction

The name of the external function to invoke or the value returned from %XADDR. This cannot be a ^VAL function. (**a**)

return

The return value of the specified external function. (**a** or **n**)

Discussion

The XSUBR subroutine invokes the specified external routine with the specified arguments. The first syntax calls a subroutine, the second calls a ^VAL function, and the third calls a non-^VAL function.



You cannot use XSUBR to access a class method.

If the routine is not present, the runtime generates a “Cannot access external routine *routine_name*” error (\$ERR_RTNNF).

VMS

Synergy/DE for OpenVMS uses shared executable images to implement an ELB. Only Synergy Language routines in the main program can be invoked by XSUBR. If the target routine is not found in the main program image, LIB\$FIND_IMAGE_SYMBOL is used to search the shared image list created by calls to OPENELB.

The order of shared image searches is in reverse order of calls to OPENELB: the first image searched will be the one referenced in the most recent call to OPENELB. Also, OPENELB must be called to access routines in a shared image; XSUBR does not have direct access to the shared image linked to the program image by the OpenVMS linker.

See also

[%XADDR on page 9-391](#).

Examples

The example below calls the **rname** external subroutine.

```
record
    rname          ,a30                      ;External routine name
    arg1           ,a5
    arg2           ,d2
    arg3           ,d1
    quit_wanted    ,d1

proc
    repeat
        begin
            xcall get_mnu(rname, quit_wanted)    ;Wait for menu selection
            if (quit_wanted .eq. 1)
                exitloop
            xcall xsubr(rname, arg1, arg2, arg3)
        end
    xreturn
end
```


The following example calls the **blue** ^VAL function. The return value for **blue** is returned in the **results** variable.

```
main
record
    arg1      ,d1      ,4
    arg2      ,d2      ,74
    results   ,d2

external function
    blue      ,i

proc
    open(1, o, "tt:")
    results = %xsubr('blue', arg1, arg2)
    writes(1, "r_arg = " + %string(results))
end

function blue, ^VAL
    a_arg1    ,d
    a_arg2    ,d
record
    results   ,i4

proc
    results = a_arg1 + a_arg2
    freturn results
end
```

^XTRNL – Provides the value of a link-time symbol

VMS

`^XTRNL (symbol)`

Arguments

symbol

A symbol that represents a 32-bit value to be resolved at link time.

Discussion

^XTRNL provides the value of a link-time global symbol. The resulting value is resolved and built into the code as a literal at link time.

If ^XTRNL is used to pass a value to a subroutine, it passes the value of the symbol *by value* as follows:

`^VAL (value)`

^XTRNL is primarily used to pass arguments to system routines.

Examples

The example below returns the username of the current OpenVMS process.

```
function get_username
record
    usrn timer      ,a15
    usrn timer      ,i2
    d_usrn timer     ,d2

record itmlst
    buflen          ,i2      ,^size(usrn timer)
    itm timer        ,i2
    buf timer        ,i4
    ret timer        ,i4
    terminator timer ,i4      ,0

proc
    itm timer = ^xtrnl(JPI$_USERNAME)
    buf timer = ^addr(usrn timer)
    ret timer = ^addr(usrn timer)
    clear usrn timer
```

```
xcall sys$getjpiw( , , , %ref(itmlst), , , )  
  
freturn(%atrm(usrn timer))  
endfunction
```

See the example for [%FSTAT](#) on page 9-134 for another example of ^XTRNL.

%ZONED – Convert expression to decimal in zoned numeric representation

zoned_numeric = %ZONED (*expression*)

Return value

zoned_numeric

The zoned numeric representation of *expression*. (**d**)

Arguments

expression

The expression to convert to decimal. (**a** or **n**)

Discussion

%ZONED returns the zoned numeric representation of the specified expression.

The conversion follows the rules for moving data to a decimal destination. (See [“Assignment Statements” on page 3-26](#) for a description of these rules.)

On OpenVMS and 64-bit systems, the size of the decimal field returned by %ZONED depends on the contents of the expression. On all other systems, %ZONED always returns a **d19**.

Examples

The following example uses alpha input as data for a zoned numeric arithmetic computation. Note that if the alpha variable contains any nonnumeric data other than a blank, a decimal point, or a sign (+ or –), a “Bad digit encountered” error (\$ERR_DIGIT) is generated.

```
subroutine math
    a_avar    ,a
record
    val1      ,d5
    val2      ,d5
    result    ,d10

proc
    result = (val1 * val2) / %zoned(a_avar)
    .
    .
    .
    xreturn
endsubroutine
```

System-Supplied Classes

Synergy Language includes several system-supplied classes. They are as follows:

- ▶ `System.String` (or `String`)
- ▶ `System.Object` (or `Object`)
- ▶ `System.Exception` (or `Exception`)
- ▶ `System.Array` (or `Array`)
- ▶ `Synergex.SynergyDE.SynException` (or `SynException`)
- ▶ `System.Collections.ArrayList` (or `ArrayList`)
- ▶ `Synergex.SynergyDE.Collections.ArrayList` (or `ArrayList`)
- ▶ `Synergex.SynergyDE.Select.Select` (or `Select`)
- ▶ `Synergex.SynergyDE.Select.From` (or `From`)
- ▶ `Synergex.SynergyDE.Select.Where` (or `Where`)
- ▶ `Synergex.SynergyDE.Select.OrderBy` (or `OrderBy`)
- ▶ `Synergex.SynergyDE.Select.Event` (or `Event`)
- ▶ `Synergex.SynergyDE.Select.AlphaEnumerator` and `RestrictedAlphaEnumerator`



The fully abbreviated forms of these classes can only be used where they are unique within all imported namespaces. For example, the `ArrayList` abbreviation requires the import of either `System.Collections` or `Synergex.SynergyDE.Collections`, but not both.

System.String

The String class looks like this:

```
namespace System
public sealed class String
    public method String
        in parm1, a
    public override method Equals, boolean
        in obj8, @*
    end
    public property Length, int
        method get
    endproperty
    public static method Concat, string
        in s1, string
        in s2, string
    end
    public override method ToString, string
    end
    public method ToUpper, string
    end
    public method ToLower, string
    end
    public method Clone, @*
    end
    public method Contains, boolean
        in s1, string
    end
    public method StartsWith, boolean
        in s1, string
    end
    public method EndsWith, boolean
        in s1, string
    end
    public static method op_Equality, boolean
        in s1, string
        in s2, string
    end
    public static method op_Inequality, boolean
        in s1, string
        in s2, string
    end
    public static method op_Addition, string
        in s1, string
        in s2, string
    end
end
```

```

        public static method op_True, boolean
            in s1, string
        end
    endclass
endnamespace

```

It consists of the following members:

String member	Description
String() constructor	Default constructor
Equals(objB) method	Returns true if objB is a string and its contents are the same as this instance; otherwise returns false
Length property	Returns the length of the string
Concat(s1,s2) method	Returns a new instance of the strings concatenated together
ToString() method	Returns the instance of this string
ToUpper() method	Returns a copy of the string in uppercase
ToLower() method	Returns a copy of the string in lowercase
Clone method	Returns an exact copy of the string
Contains(s1) method	Returns true if s1 is contained within the instance string
StartsWith(s1) method	Returns true if the instance string starts with s1
EndsWith(s1) method	Returns true if the instance string ends with s1
op_Equality(s1,s2) method	Operator override for =. Compares the string contents
op_Inequality(s1,s2) method	Operator override for !=. Compares the string contents
op_Addition(s1,s2) method	Operator for +. Concatenates two strings
op_True(s1)	Truth operator. Allows the string to function just like "if (alpha)"

All string routines are case sensitive. Operators override their symbolic forms and cannot be called directly.

System.Object

System.Object is the same as @*. The Object class looks like this:

```
namespace System
public class Object
    public method Object
    end
    method ~Object
    end
    public virtual method Equals, boolean
        in objB, @*
    end
    public static method Equals, boolean
        in objA, @*
        in objB, @*
    end
    public static method ReferenceEquals, boolean
        in objA, @*
        in objB, @*
    end
    public virtual method ToString, string
    end
    protected virtual method MemberwiseClone, @*
    end
endclass
endnamespace
```

It consists of the following members:

Object member	Description
Object() constructor	Default constructor
Equals(objB) method	Returns true if objB is equal to the instance object; otherwise returns false. Note that a derived type might override the Equals method to implement value equality
Equals(objA,objB) method	Returns true if objA is the same instance as objB, or if both are ^NULL, or if objA.Equals(objB) returns true; otherwise returns false
ReferenceEquals(objA,objB) method	Returns true if objA is the same instance as objB or if both are ^NULL; otherwise returns false

Object member	Description
ToString() method	Returns the string representation of the current instance unless a derived type overrides ToString. Note that the default implementation for System.Object.ToString() gives the name of the class, so if you create another class (for example, class1) and don't override ToString(), calling ToString() will give a string that contains the fully qualified name of the class (for example ns1.class1).
MemberwiseClone method	Creates a shallow copy of the object

Creating a dynamic array of classes using the Object class

When you use the Object class (@*), the compiler doesn't know what type of object the variable is at compile time. Therefore, you must use casting when accessing a class in an array declared with the @* type. For example,

```

namespace nsTheNameSpace
  public class clTheClass
    public file ,a15
    public method clTheClass
      proc
        text = "ABC DEF GHI JKL"
      endmethod
    endclass
endnamespace

.define NUM_OBJS 10

main
record named
  chn      ,i4
  counter  ,i4
  class    ,[#]@*      ;Declare a dynamic array of any type classes

proc
  open(chn=%syn_freechn,0,"TT:")
  class = new clTheClass[NUM_OBJS]

  for counter from 1 thru NUM_OBJS
    begin
      class[counter] = new clTheClass()
      ;Line below compiles when object is cast
      writes(chn,((clTheClass)class[counter]).file)
      ;Line below does not compile, because class declared with *
      ;writes(chn,class[counter].file)
    end
  end
end

```

System.Exception

The Exception class is the type of an exception, used for structured exception handling. The Exception class looks like this:

```
namespace System
public class Exception
    public method Exception
        end
    public method Exception
        in message, string
        end
    public method Exception
        in message, string
        in inner, @Exception
        end
    public property Source, string
        method get
        method set
    endproperty
    public property StackTrace, string
        method get
    endproperty
    public property Message, string
        method get
    endproperty
    public property InnerException, @Exception
        method get
    endproperty
    public override method ToString, string
    end
endclass
endnamespace
```

It consists of the following members:

Exception member	Description
Exception() constructor	Default constructor
Exception(s) constructor	Passes in the error message
Exception(s, Exception) constructor	Passes in the error message and the inner exception
Source property	Gets or sets the application that caused error

Exception member	Description
StackTrace property	Gets a text representation of the call stack at the time the exception was thrown
Message property	Gets the text error message describing the exception
InnerException property	Gets the exception instance that caused the current exception
ToString() method	Returns the string representation of the current exception. This includes the following components, separated by the system line terminator: ▶ The name of the exception class that was thrown ▶ The Message if one exists ▶ The result of calling ToString on the InnerException if one exists ▶ The StackTrace if one exists

System.Array

The Array class is equivalent to *array@class*. (See “[*#*,*#*]*@class*” on page 2-14 for more information.) An instance of this class will only be created via the Synergy array syntax. It represents a 1-based array. The Array class looks like this:

```
namespace System
public sealed class Array
    public property Length, int
        method get
    endproperty
    public property Rank, int
        method get
    endproperty
    public static method Clear, void
        in arr, @Array
        in index, int
        in count, int
    end
    public method SetValue, void
        in value, @*
        in index, int
    end
    public method GetValue, @*
        in index, int
    end
    public int GetLength, int
        in dim, int
    end
    public static method IndexOf, int
        in arr, @Array
        in value, @*
        in startIdx, int
        in count, int
    end
    public static method LastIndexOf, int
        in arr, @Array
        in value, @*
        in startIdx, int
        in count, int
    end
    public static method Copy, void
        in source, @Array
        in srcIdx, int
        in dest, @Array
        in destIdx, int
        in count, int
```

```

end
endclass
endnamespace

```

It contains the following members:

Array member	Description
Length property	Gets total number of all elements in all dimensions of array
Rank property	Gets the number of dimensions
Clear method	Sets elements in array to 0, false, or null depending on element type. When clearing a real array, the array behaves like one pseudo array whose rows are laid end to end. For example, StartIdx of 6 in a [5,5] array will start clearing from index [2,1]
Copy method	Copies section of one array to another array. When copying between real arrays, each array behaves like one pseudo array whose rows are laid end to end. For example, StartIdx of 6 in a [5,5] array will start copying from index [2,1]
GetValue method	Gets value at specified index in a single-dimension array
GetLength	Gets the size of the given dimension
IndexOf method	Gets first index with specified value in a single-dimension array. This method determines equality by calling the Equals method of the Array element type
LastIndexOf method	Gets last index with specified value in a single-dimension array. This method determines equality by calling the Equals method of the Array element type
SetValue method	Sets value at specified index in a single-dimension array

Synergex.SynergyDE.SynException

The SynException class is used for structured exception handling and extends the System.Exception parent class by adding more error information. All Synergy/DE exceptions derive from this class.

SynException looks like this:

```
namespace Synergex.SynergyDE
public class SynException extends System.Exception
    public method SynException
end
public method SynException
    in message, string
end
public method SynException
    in message, s
    in inner, @Exception
end
    public record
        Errno, int
        SourceFileno, int
        Lineno, int
        SysErrno, int
        InfoErr, string
endclass
endnamespace
```

It consists of the following members, as well as the members of the exception class from which it inherits:

SynException member	Description
SynException() constructor	Default constructor
SynException(s) constructor	Passes in error message
SynException(s, Exception) constructor	Passes in error message and inner exception
Errno field	The error number of the exception
SourceFileno field	The integer source file number

SynException member	Description
Lineno field	The line number
SysErrno field	The system error number
InfoErr field	One or more information messages separated by EOL if error information is available.

System.Collections.ArrayList

The `System.Collections.ArrayList` class is a dynamic collection class with 0-based indexing (which means the index of the first element is 0). It looks like this:

```
namespace System.Collections
public class ArrayList
    public method ArrayList
        end
    public method ArrayList
        in capacity, int
    end
    method ~ArrayList
    end
    public virtual property Capacity, int
        method get
        method set
    endproperty
    public virtual property Count, int
        method get
    endproperty
    public virtual property Indexer, @*
        in index, int
        method get
        method set
    endproperty
    public virtual method Add, int
        in value, @*
    end
    public virtual method RemoveAt, void
        in index, int
    end
    public virtual method IndexOf, int
        in value, @*
    end
    public virtual method IndexOf, int
        in value, @*
        in idx, int
    end
    public virtual method IndexOf, int
        in value, @*
        in idx, int
        in cnt, int
    end
    public virtual method Clear, void
    end
```



```

    public virtual method Insert, void
        in idx, int
        in value, @*
    end
    public virtual method LastIndexOf, int
        in value, @*
    end
    public virtual method LastIndexOf, int
        in value, @*
        in idx, int
    end
    public virtual method LastIndexOf, int
        in value, @*
        in idx, int
        in count, int
    end
endclass
endnamespace

```

The System.Collections.ArrayList class contains the following members:

ArrayList member	Description
ArrayList() constructor	Constructor to create a list of default capacity 16
ArrayList(i4) constructor	Constructor to create a list of specified capacity
~ArrayList() destructor	Destructor to release references to elements in the array list
Capacity property	Get or set the array list capacity
Count property	Number of elements actually in the array list
Indexer property	Access an element in the array by a 0-based index
Add method	Add an element to the end of the array list. Returns the index at which the element was added
Clear method	Clear all elements from the array list
IndexOf method	Get the first 0-based index position for the specified value. This method determines equality by calling the Equals method of the ArrayList element type

System-Supplied Subroutines, Functions, and Classes

System.Collections.ArrayList

ArrayList member	Description
Insert method	Insert an element at the specified 0-based index
LastIndexOf method	Get the last 0-based index position for the specified value. This method determines equality by calling the Equals method of the ArrayList element type
RemoveAt method	Remove the element at the specified 0-based index. The elements that follow are moved up to occupy the vacant spot

Synergex.SynergyDE.Collections.ArrayList

The Synergex.SynergyDE.Collections.ArrayList class is a dynamic collection class with 1-based indexing (which means the index of the first element is 1). It looks like this:

```
namespace Synergex.SynergyDE.Collections
public class ArrayList
    public method ArrayList
    end
    public method ArrayList
        in capacity, int
    end
    method ~ArrayList
    end
    public virtual property Capacity, int
        method get
        method set
    endproperty
    public virtual property Count, int
        method get
    endproperty
    public virtual property Indexer, @*
        in index, int
        method get
        method set
    endproperty
    public virtual method Add, int
        in value, @*
    end
    public virtual method RemoveAt, void
        in index, int
    end
    public virtual method IndexOf, int
        in value, @*
    end
    public virtual method IndexOf, int
        in value, @*
        in idx, int
    end
    public virtual method IndexOf, int
        in value, @*
        in idx, int
        in cnt, int
    end
    public virtual method Clear, void
    end
```

```
        public virtual method Insert, void
            in idx, int
            in value, @*
        end
        public virtual method LastIndexOf, int
            in value, @*
        end
        public virtual method LastIndexOf, int
            in value, @*
            in idx, int
        end
        public virtual method LastIndexOf, int
            in value, @*
            in idx, int
            in count, int
        end
    endclass
endnamespace
```

The Synergex.SynergyDE.Collections.ArrayList class contains the following members:

ArrayList member	Description
ArrayList() constructor	Constructor to create a list of default capacity 16
ArrayList(i4) constructor	Constructor to create a list of specified capacity
~ArrayList() destructor	Destructor to release references to elements in the array list
Capacity property	Get or set the array list capacity
Count property	Number of elements actually in the array list
Indexer property	Access an element in the array by a 1-based index
Add method	Add an element to the end of the array list. Returns the index at which the element was added
Clear method	Clear all elements from the array list
IndexOf method	Get the first 1-based index position for the specified value. This method determines equality by calling the Equals method of the ArrayList element type

ArrayList member	Description
Insert method	Insert an element at the specified 1-based index
LastIndexOf method	Get the last 1-based index position for the specified value. This method determines equality by calling the Equals method of the ArrayList element type
RemoveAt method	Remove the element at the specified 1-based index. The elements that follow are moved up to occupy the vacant spot

Synergex.SynergyDE.Select.Select

The Select class is a group of classes and class members that support SQL-like data selection. Its targeted use is as an enumerator accessed via the FOREACH statement, but individual access is also allowed.

The Select class object requires a From object, which describes the file to use and the record layout. It can also contain optional Where and OrderBy class objects. If a Where class object is specified, it defines the selection criteria. If it is not specified or has no record scrutiny, all records will be selected. If an OrderBy class object is specified, it defines the order by which the records will be presented upon retrieval.

The Select class object that is created represents the complete selection criteria to use when elements are enumerated. It does not retrieve or perform I/O on the selected file.

The Select class looks like this:

```
namespace Synergex.SynergyDE.Select
class Select
    public method Select
        in from, @From
    proc
    endmethod

    public method Select
        in from, @From
        in where, @Where
    proc
    endmethod

    public method Select
        in from, @From
        in where, @Where
        in orderby, @OrderBy
    proc
    endmethod

    public method Select
        in from, @From
        in orderby, @OrderBy
    proc
    endmethod

    public method GetEnumerator, @AlphaEnumerator
    proc
    endmethod
```

```

public static method GetEnumerator, @RestrictedAlphaEnumerator
proc
endmethod

public static method GetEnumerator, @RestrictedAlphaEnumerator
    in rec, a
proc
endmethod

public varargs method SparseRecord, @Select
    in arg1, any
proc
endmethod

public property QueryString, string
    method get
    proc
    endmethod
endproperty

public property isOptimized, int
    method get
    proc
    endmethod
endproperty

public property isOrderedBy, int
    method get
    proc
    endmethod
endproperty

public method RegisterEvent, @Select
    in event, @Event
proc
endmethod

endclass
endnamespace

```

System-Supplied Subroutines, Functions, and Classes

Synergex.SynergyDE.Select.Select

The Synergex.SynergyDE.Select.Select class contains the following members:

Select member	Description
Select(from) constructor	Describe the file to use and the record layout, with all records selected (see Synergex.SynergyDE.Select.From on page 9-425)
Select(from, where) constructor	Describe the file to use and the record layout, with the specified selection criteria (see Synergex.SynergyDE.Select.From on page 9-425 and Synergex.SynergyDE.Select.Where on page 9-427)
Select(from, where, orderby) constructor	Describe the file to use and the record layout, with the specified selection criteria and with the records presented in the specified order upon retrieval (see Synergex.SynergyDE.Select.From on page 9-425 , Synergex.SynergyDE.Select.Where on page 9-427 , and Synergex.SynergyDE.Select.OrderBy on page 9-437)
Select(from, orderby) constructor	Describe the file to use and the record layout, with all records selected and with the records presented in the specified order upon retrieval (see Synergex.SynergyDE.Select.From on page 9-425 and Synergex.SynergyDE.Select.OrderBy on page 9-437)
GetEnumerator method	Return an AlphaEnumerator object (see Synergex.SynergyDE.Select.AlphaEnumerator and RestrictedAlphaEnumerator on page 9-439)
GetEnumerator method	Retrieve a RestrictedAlphaEnumerator from within the nearest FOREACH context (see Synergex.SynergyDE.Select.AlphaEnumerator and RestrictedAlphaEnumerator on page 9-439)
GetEnumerator(rec) method	Retrieve a RestrictedAlphaEnumerator from within the first FOREACH context using the specified record
SparseRecord(arg1,...) method	Define the parts of the record to be returned by the enumerator
QueryString property	Return a string representing the resulting query

Select member	Description
isOptimized property	Implement a get method that returns one of the following values: 0 Selection is not optimized 1 Selection employs head optimization 2 Selection employs tail optimization 3 Selection employs both head and tail optimization
isOrderedBy property	Implement a get method that returns one of the following: -1 Selection will be ordered by file order 0 Selection will be ordered by the primary key <i>numbered_key</i> Selection will be ordered by an alternate key
RegisterEvent(event) method	Register a specific event that may be activated due to some condition during the select process. An event is considered a user-defined class that extends the Event class (see Synergex.SynergyDE.Select.Event on page 9-438)



The FOREACH statement hides the complexity of enumerators. Therefore, we recommend using FOREACH rather than directly manipulating the enumerator.

After the GetEnumerator method is called, the selection context is established on the specified channel. The AlphaEnumerator methods and properties will then operate in that context. A READ, GET, FIND, relative WRITE, or another call to GetEnumerator will break out of any selection context established on the specified channel. Performing any of the following operations on a channel with an established selection context cause an “Invalid operation: Select context is active” error (\$ERR_INVOPER): READS, GETS, ACCEPT, DELETE, or ISAM WRITE. Doing a STORE of a record on a channel with a selection context will store the new record without changing the selection context.

Only one enumerator is allowed on a channel at one time. GetEnumerator (called explicitly or implicitly through FOREACH) generates a new enumerator and invalidates any prior enumerator generated on the same Select object or on another Select object using the same From channel.

For the GetEnumerator method, a ^NULL handle is returned if an appropriate FOREACH context is not found.

The SparseRecord method is so named because it specifies a partial, or sparse, record. The method can accept one or more field specifications from a file opened either in INPUT mode or with LOCK:Q_NO_LOCK set. All field specifications must be part of the record specified in the From class object.

When a sparse record is a subset of the record, the entire record will be read, but only the specified fields will be returned; all other fields will be blanked out. Using the SparseRecord method is most effective when you're using *x/Server* and only parts of the record are needed. Only those fields specified are transferred across the network.

Examples

```
foreach rec in new Select(new From(ch, record), (Where) fld1.ge.10)
begin
    writes(TTCHN, rec)
    rfa = Select.GetEnum(rec).CurrentRFA
end
or
fobj = new From(ch, Q_NOLOCK, Q_NOWAIT, rec)
wobj = Where.Between((where) custno, 1, 1000)
sobj = new Select(fobj, wobj)
enuma = select.GetEnumerator()
while (enuma.MoveNext)
begin
    writes(TTCHN, enuma.Current)
    rfa = enuma.CurrentRFA
end
```

Synergex.SynergyDE.Select.From

The From class defines from which file a selection will be made. It also defines the input record size and provides a template for determining field to record offsets. In addition, the locking behavior can be altered from the locking behavior of the file open.



When changing locking behavior, the new behavior must be supported by the open mode of the file. Adding locking (Q_AUTO_LOCK or Q_MANUAL_LOCK) is ignored if the file is opened for input mode

The From class looks like this:

```
namespace Synergex.SynergyDE.Select
class From
    public method From
        in ch, int
        in record, a
    proc
    endmethod

    public method From
        in ch, int
        in lock, int
        in wait, int
        in record, a
    proc
    endmethod
endclass
```

The Synergex.SynergyDE.Select.From class contains the following members:

From member	Description
From(ch, record) constructor	Specify the file from which the selection will be taken and the name of a record, group, or structure that represents the beginning offset and maximum record size of the record to be read
From(ch, lock, wait, record) constructor	Specify the file from which the selection will be taken and the name of a record, group, or structure that represent the beginning offset and maximum size of the record to be read, and set up locking behavior that overrides the behavior set up for the channel by the Open statement

System-Supplied Subroutines, Functions, and Classes

Synergex.SynergyDE.Select.From

The fields used by the other Where class methods will be fields declared within this buffer. The values passed to lock and wait are the same values used by the OPEN and READ statements. For example,

```
lock=Q_AUTO_LOCK  
wait=-1
```

If the channel is not open, a “Channel has not been opened” error (\$ERR_NOOPEN) will occur.

Synergex.SynergyDE.Select.Where

The Where class object is the fundamental building block for a selection expression. Essentially a collection of logical and relational operators (including their symbolic forms), it provides a means to create intuitive, procedure-like expressions for record selection.

A Where expression is processed from left to right (according to implicit or explicit expression precedence). The first operation that references a defined ISAM key is established as an explicit selection hint. If no operation references an ISAM key, the primary key is established as an implicit selection hint.

A Where expression requires at least one operand of each expression to be explicitly or implicitly cast. Using the op_Explicit method is an explicit cast. For example,

```
(Where) fld1.eq.1.and.(Where) fld2.gt.20
```

An implicit cast is applied to all Where class operators within a parenthesized expression when an explicit (Where) cast is used. For example, with implied Where expression casting,

```
(Where) fld1.eq.1.and.(Where) fld2.gt.20
```

can be simplified to

```
(Where) (fld1.eq.1.and.fld2.gt.20)
```

The Where class looks like this:

```
namespace Synergex.SynergyDE.Select
class Where
  public static method op_Explicit, @Where
    in parm, any
  proc
  endmethod

  public static method op_LogicalAnd, @Where
    in left, @Where
    in right, @Where
  proc
  endmethod

  public static method op_LogicalNot, @Where
    in parm, @Where
  proc
  endmethod

  public static method op_LogicalOr, @Where
    in left, @Where
    in right, @Where
  proc
  endmethod
```

System-Supplied Subroutines, Functions, and Classes

Synergex.SynergyDE.Select.Where

```
public static method op_Equality, @Where
    in left, @Where
    in right, any
proc
endmethod

public static method op_Equality, @Where
    in left, any
    in right, @Where
proc
endmethod

public static method op_EQS, @Where
    in left, @Where
    in right, any
proc
endmethod

public static method op_EQS, @Where
    in left, any
    in right, @Where
proc
endmethod

public static method op_Inequality, @Where
    in left, @Where
    in right, any
proc
endmethod

public static method op_Inequality, @Where
    in left, any
    in right, @Where
proc
endmethod

public static method op_NES, @Where
    in left, @Where
    in right, any
proc
endmethod

public static method op_NES, @Where
    in left, any
    in right, @Where
proc
endmethod
```

```
public static method op_GreaterThan, @Where
    in left, @Where
    in right, any
proc
endmethod

public static method op_GreaterThan, @Where
    in left, any
    in right, @Where
proc
endmethod

public static method op_GTS, @Where
    in left, @Where
    in right, any
proc
endmethod

public static method op_GTS, @Where
    in left, any
    in right, @Where
proc
endmethod

public static method op_LessThan, @Where
    in left, @Where
    in right, any
proc
endmethod

public static method op_LessThan, @Where
    in left, any
    in right, @Where
proc
endmethod

public static method op_LTS, @Where
    in left, @Where
    in right, any
proc
endmethod

public static method op_LTS, @Where
    in left, any
    in right, @Where
proc
endmethod
```

System-Supplied Subroutines, Functions, and Classes

Synergex.SynergyDE.Select.Where

```
public static method op_GreaterThanOrEqual, @Where
    in left, @Where
    in right, any
proc
endmethod

public static method op_GreaterThanOrEqual, @Where
    in left, any
    in right, @Where
proc
endmethod

public static method op_GES, @Where
    in left, @Where
    in right, any
proc
endmethod

public static method op_GES, @Where
    in left, any
    in right, @Where
proc
endmethod

public static method op_LessThanOrEqual, @Where
    in left, @Where
    in right, any
proc
endmethod

public static method op_LessThanOrEqual, @Where
    in left, any
    in right, @Where
proc
endmethod

public static method op_LES, @Where
    in left, @Where
    in right, any
proc
endmethod

public static method op_LES, @Where
    in left, any
    in right, @Where
proc
endmethod
```



```
public static method Keynum, @Where
    in krf, int
proc
endmethod

public static method Contains, @Where
    in item, any
    in value, a
proc
endmethod

public static method Contains, @Where
    in fld, any
    in expr, @Where
proc
endmethod

public static method Between, @Where
    in fld, any
    in low, any
    in high, any
proc
endmethod

public static varargs method In, @Where
    in fld, any
    in value1, any
    in value2, any
proc
endmethod

public static varargs method Portable, @Where
    in ifld, int
proc
endmethod

endclass
```

System-Supplied Subroutines, Functions, and Classes

Synergex.SynergyDE.Select.Where

The Synergex.SynergyDE.Select.Where class has the following members.

Where member	Description
op_Explicit(parm) method	Implement the (Where) cast
op_LogicalAnd(left, right) method	Implement the .and. binary operator
op_LogicalNot(parm) method	Implement the .not. unary operator
op_LogicalOr(left, right) method	Implement the .or. binary operator
op_Equality(left, right) method	Implement the .eq. binary operator
op_EQS(left, right) method	Implement the .EQS. string relational operator
op_Inequality(left, right) method	Implement the .ne. binary operator
op_NES(left, right) method	Implement the .NES. string relational operator
op_GreaterThan(left, right) method	Implement the .gt. binary operator
op_GTS(left, right) method	Implement the .GTS. string relational operator
op_LessThan(left, right) method	Implement the .lt. binary operator
op_LTS(left, right) method	Implement the .LTS. string relational operator
op_GreaterThanOrEqual(left, right) method	Implement the .ge. binary operator
op_GES(left, right) method	Implement the .GES. string relational operator
op_LessThanOrEqual(left, right) method	Implement the .le. binary operator
op_LES(left, right) method	Implement the .LES. string relational operator
Keynum(krf) method	Set the explicit key of reference for optimization and selection ordering purposes
Contains(item, value) method	Implement an SQL-like Contains method, which checks if a value contains a certain item
Contains(fld, expr) method	Implement an SQL-like Contains method, which checks if a field contains an alpha value

Where member	Description
Between(field, low, high) method	Return a Where class object and take as its arguments a single field specification, the low value, and the high value, respectively. At runtime, this method will be transformed to the appropriate expression in the form field.ge.low.and.field.le.high
In(field, value1, value2, ...) method	Return a Where class object and take as its arguments a single field specification followed by two or more value arguments. At runtime, this method will be transformed to the appropriate expression in the form field.eq.value1.or.field.eq.value2.or.field.eq.value3
Portable(ifld) method	Identify integer fields in a record that are converted to portable integers when stored

These methods are equivalent to their Synergy counterparts; see [Chapter 3, “Expressions”](#) for additional information about each operator.

When the Where class implements an explicit type conversion, for example,

(Where) descriptor_type

the result of the operator is a Where class object. If *descriptor_type* is not a Synergy descriptor type or Where class type, an “Invalid argument:” error (\$ERR_INVARG) occurs.

The string relational operators (EQS, GTS, LTS, etc.) and their symbolic relational forms (==, >, <, etc.) always result in a string comparison, compared for both of their lengths. Note that in a Where expression, the symbolic operators (==, >=, etc.) are treated a little differently than they are in the Language:

Symbolic operator expression	Language expression	Where expression
(avar1 == avar2)	avar1 .eq. avar2	avar1 .eqs. avar2
(avar1 == string)	avar1 .eqs. string	avar1 .eqs. string

SQL symbolic comparisons such as ==, >, <, etc., behave like string comparisons. For example,

(avar .eqs. "abc") is equivalent to the SQL statement (WHERE avar = 'abc')

and

(avar .eq. "abc") is equivalent to the SQL statement (WHERE avar LIKE 'abc%')

The alpha relational operators (EQ, GT, LT, etc.) always result in an alpha comparison, compared for the length of the shortest. This is similar to the SQL LIKE operator with an implied trailing “*” wildcard.

The Keynum method sets the explicit key of reference for optimization and selection ordering. If Keynum is not specified in a Where expression, the primary key is used for optimization and ordering. See [“Optimization and ordering” on page 9-435](#) for more information.

If Keynum is specified, it must be the first (leftmost) operator in the Where expression (unless Portable is also specified, in which case Portable can be the first operator and Keynum can be the second operator). Only one Keynum can be specified in a Where expression, and it must be separated from the rest of the Where expression by the .AND. operator. For example,

```
Where.Keynum( 1 ) .and. Where.Contains(field, value)
```

If the value passed to Keynum does not match an existing KEYNUM for the file specified by the From object, an “Illegal key specified” error (\$ERR_BADKEY) will be generated at runtime.

The Contains method returns a Where class object. Its first argument is either a single field specification within the buffer specified in the From class object or a Where class object, and its second argument is a single alpha data reference. For example,

```
Where.Contains(field, avar)
```

The Contains method can alternatively take a field specification as its first argument and a Where class object as its second argument. For example,

```
Where.Contains(field, (Where)avar1 .or. avar2)
```

The Where object must be an alpha expression constructed with one or more of the following binary operators (or their symbolic forms): AND, OR, NOT. The comparison is case-sensitive.

A positive selection occurs when the alpha value is contained within the specified field. The specified field must be a field within the buffer specified in the From class object; otherwise an “Invalid argument” error (\$ERR_INVARG) will be generated at runtime.

Typically an application that shares files between systems with a different endian byte order will convert non-key integers to portable format before storing and after reading records from the shared files. The Portable method is provided for handling records read from files of this type. (Portable only affects functions of Select; it does not affect STORE.)

If Portable is specified, it must be the first (leftmost) operator in the Where expression (unless Keynum is also specified, in which case Keynum can be the first operator and Portable can be the second operator). Only one Portable can be specified in a Where expression, and it must be separated from the rest of the Where expression by the .AND. operator. For example,

```
Where.Portable(ifld) .and. Where.Contains(field, value)
```

ISAM

Optimization and ordering

Explicit and implicit ordering

The ordering of retrieval can be specified explicitly or implicitly. The order of precedence is as follows:

- ▶ Where.Keynum(n) and/or OrderBy.AscendingKey(n) or OrderBy.DescendingKey(n) explicitly sets the key of reference to the specified key, which determines the retrieval order.
- ▶ If neither Where.Keynum or OrderBy(key) are specified, an optimized key determines the retrieval order.
- ▶ If no optimization occurs, the primary key determines the retrieval order.

Implicit key selection via optimization

If the key of reference is not explicitly specified using Where.Keynum or Orderby(key), key optimization will implicitly set the key that determines the retrieval order. If key optimization cannot be achieved, the primary key becomes the key of reference. The following key selection is presented in order of precedence:

- ▶ In the Where clause, starting from left to right, the first equality operator (EQ) whose field specification matches a key (or partial key starting from the beginning of the key) or key segment (beginning with the first segment) determines the optimized key.
- ▶ In the Where clause, starting from left to right, the first relational operator narrowing pair (greater-than GT or GE with matching less-than LT or LE) whose field specification matches a key (or partial key starting from the beginning of the key) or key segment (beginning with the first segment) determines the optimized key.
- ▶ In the Where clause, starting from left to right, the first relational operator (LT, LE, GT, or GE) whose field specification matches a key (or partial key starting from the beginning of the key) or key segment (beginning with the first segment) determines the optimized key.



To be eligible for optimization, the Where clause must not contain any .OR. or .NOT. operators.

Optimization types

There are currently two types of Where clause optimization: head optimization and tail optimization. Head optimization is a single direct positioning (e.g., FIND) to the beginning of the selection with little or no record I/O. In the absence of head optimization, sequential record I/O occurs starting from the beginning of the file. Tail optimization is a known exit condition that terminates the sequential record I/O when it is known that past a certain point there will be no further matches to the selection criteria. In the absence of tail optimization, sequential record I/O

continues until the end of the file. A key or segment contained in an EQ operator expression will always generate both head and tail optimization. A key or segment contained in a relational operator expression will generate either a head or a tail optimization, depending on the key or segment order. A key or segment contained in a pair of narrowing relational operator expressions will generate both head and tail optimization. (A narrowing condition is defined as two relational operator expressions on the same key or key segment where one is a GT or GE relational operator and the other is the complement relational operator LT or LE.)

Optimizing key segments

To optimize key segments, the following should be considered:

- ▶ All key segments that will be optimized must be contiguous segments starting from segment 1.
- ▶ All segments other than the first must be preceded by a prior segment contained in an EQ operator expression to be optimized.

A narrowing condition involving a segmented key is possible when two comparisons using the same key segment are complementary relational operators, provided the above two rules are true.

Ordering of segmented keys is determined by the following:

- ▶ If individual segment order is not specified during file creation (ORDER=), the retrieved order is based on the entire key.
- ▶ If the Where clause is not eligible for optimization, the retrieved order is based on the order of the first segment.
- ▶ When optimized, the segment order of the first key segment with a relational operator following a prior segment or the last key segment containing an equality operator determines the retrieval order. For example, the order in the following expression is based on segment 2 (seg2):

```
(Where) (seg1.eq.500.and.seq2.gt.10.and.seq3.lt.40)
```

Using key optimization (ISAM files)

To make use of Where clause optimization for best performance, consider the following:

- ▶ When more than one record match is necessary (using an OR condition), consider using two separate Selects rather than using an OR. Use GE or LE instead of EQ .OR. GT, etc.
 - ▶ Avoid using .NOT. when possible. Use GE instead of .NOT. LT, etc.
 - ▶ Place fields that coincide with the most unique key leftmost in the Where clause to force the best optimization.
-

Synergex.SynergyDE.Select.OrderBy

The OrderBy class is a collection of static methods for ordering a selection. These methods specify the order by which a selected key is traversed, thereby affecting the order of the final output.

OrderBy specifies the key of reference for the Where object selection if the key of reference has not been explicitly specified by Where.Keynum. If both keyed OrderBy and Where.Keynum are specified, they must specify the same key of reference, or an “Invalid option” error (\$ERR_OPTINV) will be generated. If the specified key of reference is a segmented key, and the segments are individually ordered differently, the order is determined by the key optimization level. If no optimization occurs, the order is determined by the first key segment. (See [“Optimization and ordering” on page 9-435](#) for more detailed information.)

The OrderBy class looks like this:

```
namespace Synergex.SynergyDE.Select
class OrderBy
    public static method AscendingKey, @OrderBy
        in left, int
    proc
    endmethod

    public static method DescendingKey, @OrderBy
        in left, int
    proc
    endmethod
endclass
```

The Synergex.SynergyDE.Select.OrderBy class contains the following members:

OrderBy member	Description
AscendingKey(left)	Specify that Where object selection occurs in ascending order
DescendingKey(left)	Specify that Where object selection occurs in descending order

Synergex.SynergyDE.Select.Event

The Event class object includes event override methods.

The Event class looks like this:

```
namespace Synergex.SynergyDE.Select
class Event
    public virtual method onLock, boolean
        inout lock, int
        inout wait, int
        in rec, a
        in rfa, a
    proc
        mreturn false
    endmethod
```

The Synergex.SynergyDE.Select.Event class contains the following member:

Event member	Description
onLock(lock, wait, rec, rfa) method	Change locking behavior, if possible, when a “Record is locked” error (LOCKED) is generated during an iteration. onLock is called with the current lock and wait settings, as well as a copy of the record that was locked and its RFA.

Changing the lock parameter to Q_NOLOCK before returning true will cause the lock wait behavior to be changed for the retry operation, and then reset on the next operation.

If the returned RFA is to be used to re-establish the record after the Select has completed, we recommend that the file be configured with static RFAs.

Synergex.SynergyDE.Select.AlphaEnumerator and RestrictedAlphaEnumerator

The AlphaEnumerator class is an enumerator class that is used to access elements of a selection and their attributes. An AlphaEnumerator class object is required to retrieve the data from a selection. Initially, following the creation of the enumerator, it is positioned before the first element in the selection. The MoveNext method is used to position to the next element and retrieve the data, one element per call.



Using a FOREACH statement with a Select object creates and maintains the enumerator class object internally and is hidden.

Although the AlphaEnumerator class object and members are not available from within a FOREACH statement, you can access the RestrictedAlphaEnumerator class object from the FOREACH statement controls by using the Select class GetEnum method. The RestrictedAlphaEnumerator class allows access to the current record but not the enumerator itself.

The AlphaEnumerator and RestrictedAlphaEnumerator classes look like this:

```
namespace Synergex.SynergyDE.Select
class RestrictedAlphaEnumerator
    public property Current, a
        method get
        proc
        endmethod

        method set
        proc
        endmethod
    endproperty

    public method DeleteCurrent, void
    proc
    endmethod

    public method CurrentRFA, void
        out rfa, a
    proc
    endmethod

    public property CurrentRDLEN, int
        method get
        proc
        endmethod
    endproperty
```

System-Supplied Subroutines, Functions, and Classes

Synergex.SynergyDE.Select.AlphaEnumerator and RestrictedAlphaEnumerator

```
    public property CurrentReadCount, int
        method get
        proc
        endmethod
    endproperty

    public property TotalReadCount, int
        method get
        proc
        endmethod
    endproperty

    public property isLocked, int
        method get
        proc
        endmethod
    endproperty
endclass

class AlphaEnumerator extends RestrictedAlphaEnumerator
    public method MoveNext, boolean
    proc
    endmethod

    public property Count, int
        method get
        proc
        endmethod
    endproperty

    public method Reset, void
    proc
    endmethod

    public property GetEnumerator, @AlphaEnumerator
        method get
        proc
        endmethod
    endproperty
endclass
```

The Synergex.SynergyDE.Select.AlphaEnumerator and RestrictedAlphaEnumerator classes contain the following member:

AlphaEnumerator or RestrictedAlphaEnumerator member	Description
Current property	Implement a get method that returns the current record and a set method that updates the current record by effectively doing a WRITE to the file, which unlocks the record on completion
DeleteCurrent method	Delete the current record. If the current record is not locked, a "No current record" error (E_NOCURR) is generated
CurrentRFA(rfa) method	Return the RFA or GRFA of the current record, depending on whether rfa is 6 or 10, respectively
CurrentRDLEN property	Implement a get method that returns the %RDLEN of the Current record
CurrentReadCount property	Implement a get method that returns a value representing the number of read operations made to retrieve the current record
TotalReadCount property	Implement a get method that returns a value representing the total number of read operations made to retrieve all records up to the current record. If TotalReadCount is used after the MoveNext method completes (and returns false), the value returned represents the total number of read operations used to retrieve the entire selection
isLocked property	Implement a get method that returns a non-zero value if the current record was acquired with a lock and is currently locked and a 0 value if it is not currently locked
MoveNext method	Advance to the next record and return true if successful and false if there are no more records. (This method performs actual I/O based on the selection criteria)

System-Supplied Subroutines, Functions, and Classes

Synergex.SynergyDE.Select.AlphaEnumerator and RestrictedAlphaEnumerator

AlphaEnumerator or RestrictedAlphaEnumerator member	Description
Count property	Implement a get method that returns the number of elements currently in the file (table) that match the selection criteria. Note: This property must iterate through the entire file (which may cause extensive I/O local to the file) and finish by calling the Reset property.
Reset method	Set the enumerator to its initial position before the first element in the selection
GetEnumerator property	Implement a get method that returns the this object for the current AlphaEnumerator

The get method of the Current property returns the same record until either MoveNext or Reset is called. The set method requires the file to be opened for update and the current record to be locked, or else an “Invalid operation for file type” error (\$ERR_FILOPT) and “No current record” error (\$ERR_NOCURR) will be generated.

Both get and set methods generate a “No current record” error if the enumerator is in its initial position (prior to calling MoveNext or after calling Reset) or in its final position (after MoveNext returns FALSE, indicating EOF).

If automatic record locking is enabled, the MoveNext method causes the obtained record to be locked and any prior selected record to be unlocked. If the next record that matches the selection criteria is locked, one of the following occurs:

- ▶ If locking has not been changed by the From class object, standard record locked behavior will occur: An \$ERR_LOCKED error is generated or execution is suspended if DFLAG 3 (wait on lock) is set. (See [“Synergex.SynergyDE.Select.From” on page 9-425.](#))
- ▶ The Event.onLock method is called, and if it is overridden and returns TRUE, another attempt is made to read the record. (See [“Synergex.SynergyDE.Select.Event” on page 9-438.](#))

Records not matching the selection criteria that are locked are always skipped, so handling a lock in that condition is not necessary.”

The CurrentRFA method returns the same RFA or GRFA until either MoveNext or Reset is called. If the enumerator is in its initial position (prior to calling MoveNext or after calling Reset) or in its final position (after MoveNext returns FALSE), a “No current record” error (\$ERR_NOCURR) will occur.

You can gather performance statistics by using the Statistics method to query the effectiveness of certain selections. Using optimization correctly can improve performance, and analyzing these statistics and tuning the Where clause appropriately can lead to better optimization.

```
record
    enum, @RestrictedAlphaEnumerator    ;Must be declared outside FOREACH
    .                                    ; to access after exiting
    .                                    ; compound statement
    .
    foreach rec in new Select(from, where)
        begin
            .
            .
            .
            enum = Select.GetEnum()
            enum.Statistics(n)            ;n returned with # I/Os used to
                                           ; retrieve the current record
        end
    enum.Statistics(n, m)                ;n returned with # I/Os used to
                                           ; determine selection complete,
                                           ; m returned with total # I/Os
```

Because the enumerator is a snapshot in history, when following the Count property, MoveNext may encounter more or fewer elements than returned by Count if updates are allowed to occur to the file.

10

Synergy ActiveX API

The Synergy ActiveX API is not available on Synergy .NET.

Introduction to the Synergy ActiveX API 10-2

Discusses requirements for using the ActiveX API, callback routines, properties supported by the ActiveX API, ActiveX data types, the ActiveX debug display, and more. A sample program is also included.

Synergy ActiveX API Routines 10-12

This API includes the following routines:

AX_BIND – Bind a callback subroutine to an ActiveX event	10-13
%AX_CALL – Call a method within an ActiveX control	10-15
%AX_CREATE – Create an ActiveX container	10-16
%AX_DELETE – Delete an ActiveX object.....	10-18
AX_GET – Get a property value for an ActiveX object	10-19
%AX_GETINT – Get an integer property value for an ActiveX object	10-20
AX_HIDE – Hide an ActiveX container	10-21
%AX_LOAD – Load an ActiveX control into a container	10-22
AX_NEWVAL – Modify an ActiveX callback routine’s parameter	10-24
AX_SET – Set a property in an ActiveX object.....	10-26
AX_SHOW – Show an ActiveX container	10-28
AX_TIMEOUT – Set the time-out value for an ActiveX control.....	10-29
%WIN_PROC – Process system-level Windows messages.....	10-30
%WIN_STOP – Suspend Windows system processing	10-31

Introduction to the Synergy ActiveX API

The Synergy™ ActiveX API provides ActiveX® Automation support to Synergy applications. ActiveX is a Microsoft technology that stems from OLE and COM (Component Object Model). ActiveX Automation is the process of controlling the functionality of an ActiveX component's methods from another application or development tool. By adding ActiveX components to your application, you can extend its capabilities with a wide variety of interactive and multimedia effects.

Requirements

The file **DBLDIR:activex.def** defines the macros and constants required by the ActiveX API. You must include this file in your program using the **.INCLUDE** compiler directive.

All programs that use the ActiveX API must be built using the compiler options **-T** (which trims trailing null arguments from XCALLs and function references) and **-X** (which automatically declares all undefined function references as external ^VAL functions).

You must also link with **DBLDIR:axlib.elb**.

The *Migrating Your Application to Windows* document, available on the Online Manuals CD, contains an overview for developing with ActiveX components and describes how to register an ActiveX control.

Callback routines

A *callback routine* is a Synergy Language routine that is called whenever a specific event occurs within an ActiveX control. A callback routine must be bound to the event using the **AX_BIND** routine. (See [AX_BIND on page 10-13](#) for syntax.) Alternatively, you can omit the **AX_BIND** step if you've specified the *autobind_prefix* argument to **%AX_CREATE** and **%AX_LOAD**. An *autobind_prefix* is a routine prefix that causes automatic binding to occur. If this argument is present, each event is automatically bound to any existing routine whose name is the specified prefix followed by the event's name.

The first argument of every callback routine is the ID of the ActiveX control that calls it (that is, the *id* passed to **AX_BIND**). Passing this control ID enables a routine to interact with an ActiveX control without requiring global data. In fact, if there are no global data references, the same routine can be used as a callback for more than one loaded ActiveX control or event.

ActiveX controls can also have parameters associated with their events. The parameters are passed as arguments to the bound callback routine, beginning with the callback routine's second argument. In other words, the first parameter of the ActiveX event is passed as the second argument to the callback routine, the second parameter is passed as the callback routine's third argument, and so on.

Every parameter passed is automatically promoted or converted to a data type that is consistent within Synergy Language. For example, if two parameters are defined as an integer and a string, they are converted to an **i4** and an **ann** field, respectively. From the standpoint of the callback routine, it is the same as processing a standard external subroutine from a Synergy Language routine. Passing an array argument converts the array to a BSTR (a variable-length string).

All parameter passed to a callback routine are read-only. If you attempt to modify one directly, a runtime “Writing to a literal” error is generated. If you need to modify a callback routine parameter, use AX_NEWVAL. (See [AX_NEWVAL](#) on page 10-24 for more information.)

Be aware of synchronicity with your callback routines. An ActiveX control may fire many simultaneous events as the user moves the mouse, for instance. While your program is in a callback routine, it can be interrupted and the same callback called recursively. Ensure all callback routines are re-entrant, and don’t use writable global data. See [%M_SIGNAL](#) in the “Menu Routines” chapter of the *UI Toolkit Reference Manual* concerning the use of [%M_SIGNAL](#) within an ActiveX event.

Workbench support

Professional Series Workbench facilitates the use of ActiveX controls in your applications. First, it enables you to add ActiveX controls to your Workbench projects in the same way you would add any other file to a project. As soon as you add an ActiveX control, Workbench loads the classes specified in the control, along with their methods and fields, into the project’s tag database.

Workbench also enables you to browse an ActiveX control in its class browser so you can see what functionality and objects the control provides.

UI Toolkit support

To integrate ActiveX controls into UI Toolkit, you must use a special Toolkit container window. A Toolkit container window has the basic characteristics of a Toolkit window (character-based increments, placement order, maintained environment, and title manipulation, for example), yet it has the display and operational characteristics of an ActiveX container (load controls, get/set property values, call methods, bind callback routines, and so forth).

For more information, see the “[ActiveX Routines](#)” chapter of the *UI Toolkit Reference Manual*.

Microsoft Windows support

The ActiveX API includes two functions that provide the Synergy Language developer with an interface to the Windows system. Refer to the specified pages for syntax and additional information:

- ▶ [%WIN_PROC](#) [page 10-30](#)
- ▶ [%WIN_STOP](#) [page 10-31](#)

Properties

A *property* is a named attribute associated with an ActiveX object. Property names can be up to 63 characters long and are not case sensitive. All properties are manipulated using the AX_GET (or %AX_GETINT) and AX_SET routines. The following basic kinds of properties are supported:

- ▶ Control
- ▶ Extended
- ▶ Ambient
- ▶ Local

Control properties

Control properties are defined by each ActiveX object, both in name and in data type. There are no requirements or conventions regarding what control properties should or should not be present; it is entirely up to the ActiveX control's designer. The ActiveX API supports the following control properties for containers:

- ▶ Caption Container's caption.
- ▶ ClipChildren WS_CLIPCHILDREN window style
- ▶ Icon Container's icon.

The value of Caption is a string that specifies the caption, or title, of the window.

ClipChildren can be set to true (the default) or false. True makes the ActiveX container have the WS_CLIPCHILDREN window style, which reduces background repainting (flicker) and some repaint anomalies. You can turn this setting off if the control expects the background to be erased.

Icon is set using an icon filename string and an optional integer value. If present, the value is the one-based index in the icon file for the icon to reference. If not specified, a value of 1 is assumed.

Some controls support “indexed” properties, which are stored as arrays in a control and have one or more indices. When an indexed property is accessed, the index values of the array element to reference must be added as arguments to the end of the AX_GET, %AX_GETINT, or AX_SET call. The user documentation for a given control should specify when a property is indexed.

Extended properties

An ActiveX container can “extend” a set of properties to each control that is loaded in that container. The following extended properties are supported:

- ▶ Ext_hWnd Object's window handle. (read-only)
- ▶ Ext_Rect Object's position and size. (read/write)
- ▶ Ext_Visible Object's visible state. (read/write)
- ▶ Ext_Focus Whether or not the object has focus. (read/write)

For the read/write properties, changing the property using AX_SET changes the object's visual representation if necessary.

The value of Ext_hWnd is a standard numeric window handle for the window associated with the object.

The value of Ext_Rect is a single-dimensioned numeric array whose first four elements are the pixel values for the left, top, right, and bottom of the object's window. (These values are screen coordinates for an ActiveX container but container coordinates for an ActiveX control.)

The value of Ext_Visible is the numeric value 1 (true) if the object's window is visible or 0 (false) if the window is hidden. Although the initial state of Ext_Visible is true for loaded controls, it is false for created containers. Thus, you must use AX_SET to set a created container's Ext_Visible value to true before the container can be viewed on the screen. The routines AX_SHOW(*id*) and AX_HIDE(*id*) manipulate the visible state.

The value of Ext_Focus can only be true. When Ext_Focus is set for a control, the overall system focus is set to that control. You don't have to clear the focus from the previous control; it is cleared automatically. An AX_GET is true (-1) if the related object currently has the system focus and false (0) if not. An AX_SET of a nonzero value sets the system focus to the related object, while an AX_SET of zero is ignored. Set Ext_Focus for the control on which you want to focus rather than on the control's container.

Ambient properties

Each ActiveX container has a set of properties that provides default values for control properties. These properties are known as *ambient properties*, and one set is associated with each container. Setting an ambient property for a container sets it for all controls within that container.

The ambient properties provided by the Synergy ActiveX API are

- ▶ Amb_BackColor Background color.
- ▶ Amb_ForeColor Foreground (text) color.
- ▶ Amb_LCID The language used. (read only)
- ▶ Amb_UserMode The user or design mode state. (read/write)

The value of Amb_BackColor and Amb_ForeColor is a numeric RGB (red, green, blue) value that has the hexadecimal form

`^X(00bbggrr)`

where *bb*, *gg*, and *rr* are intensity values in the decimal range 0–255 for the blue, green, and red color components, respectively. (For convenience, the macro RGB_VALUE(*rr,gg,bb*), which constructs a numeric RGB value, is defined in **DBLDIR:activex.def**. The RED_PART(*rgb_value*), GREEN_PART(*rgb_value*), and BLUE_PART(*rgb_value*) macros are also defined.)

The value of Amb_LCID is a numeric "Locale ID" as defined within the Windows system.

The returned value of `Amb_UserMode` is either true (-1) if the container is currently running in user mode or false (0) if it is running in design mode. When setting `Amb_UserMode`, a zero value sets the container to design mode, while a nonzero value sets it to user mode. Note that a container should be set to user mode before any processing of loaded controls occurs.

Local properties

You may sometimes want to associate user-definable data with an ActiveX object. *Local properties* enable you to define, access, and modify named values that are associated with any given ActiveX object (container or control). Local properties are manipulated using the standard `AX_GET`, `AX_SET`, and `%AX_GETINT` mechanisms and have the following characteristics:

- ▶ All local property names begin with a tilde (~).
- ▶ The data associated with a local property is a signed, 32-bit value.
- ▶ Local properties are created with a value of 0 on their first reference.



If a single 32-bit value is not enough information for a given local property, you can always use the dynamic memory support (the `%MEM_PROC` routine and the `STRUCTURE` statement) and store the memory handle as the property's value.

ActiveX data types

The following ActiveX data types are supported for `AX_GET` and arguments passed to a routine bound to a control's event.

- ▶ `VT_BOOL`. Represented in Synergy Language as an integer field whose valid values are 0 (false) and -1 (true).
- ▶ `VT_R4`. A 4-byte (32-bit) “real” value that is represented in Synergy Language as a **d28.15**, with seven decimal digits of significance.
- ▶ `VT_R8`. An 8-byte (64-bit) “real” value that is represented in Synergy Language as a **d28.15**, with fifteen decimal digits of significance.
- ▶ `VT_CY`. A currency type, which is a 64-bit integer whose low-order four decimal digits are to the right of the decimal point. The representation in Synergy Language is **d19.4**.
- ▶ `VT_DATE`. A date type that is represented in Synergy Language as an **a14** in the same form as `%DATETIME (YYYYMMDDHHMMSS)`.
- ▶ `VT_BSTR`. A variable-length string represented in Synergy Language as an alpha field.
- ▶ `VT_I1`, `VT_I2`, `VT_I4`, `VT_I8`, `VT_INT`, `VT_ERROR`, `VT_HRESULT`, `VT_UI1`, `VT_UI2`, `VT_UI4`, `VT_UI8`, `VT_UINT`. Represented in Synergy Language as an integer field.
- ▶ `VT_VARIANT`. Allows any of the listed data types. This is similar to using “,n” in a Synergy Language subroutine, and in fact, using “,n” is correct for numeric `VT_VARIANT`s in a bound routine.

- ▶ VT_USERDEFINED. Represented in Synergy Language as an integer. You (the developer) define what it means.
- ▶ Any other data types return an **a4** containing the string “????”.

For AX_SET, AX_NEWVAL, and %AX_CALL, in addition to the above data types, VT_NULL and VT_EMPTY are also supported. If the ActiveX data type is not supported for conversion from a Synergy data type, an error is generated. When dealing with VT_VARIANT, the data type of the passed Synergy Language argument determines the data type passed to the ActiveX control. We translate the Synergy data to the nearest VT_ data type. It is up to the ActiveX control to be able to support the data type passed.



An ActiveX method that contains optional arguments generates an “Argument missing” error if the argument is specified as data type VT_BSTR. This error does not occur if the data type for the optional argument is specified as VT_VARIANT.

ActiveX Diagnostic utility

The ActiveX Diagnostic utility (**axutl.exe**) provides basic analysis of an ActiveX control. It tests to make sure a given control can be accessed on your system, and it can also be used to register controls. If you cannot get an ActiveX control to load, this utility should be your first diagnostic tool. See “[The ActiveX Diagnostic Utility](#)” in the “General Utilities” chapter of *Synergy Language Tools* for more information.

Licensing considerations

Many ActiveX controls require a license. If your site is licensed to use a particular ActiveX control, you will usually have an **.lic** file for that control on your system. An ActiveX control can also be run at an unlicensed site (in other words, one that does not have an **.lic** file) as long as you pass a license string to the control using %AX_LOAD. This means that if you want to redistribute an ActiveX control to your end-users, you must either redistribute the **.lic** file (if it is legally permitted by the control’s owner) or pass a license string to %AX_LOAD in order for the control to load on an end-user’s machine. If you are not permitted to redistribute the **.lic** file, passing a license string to %AX_LOAD is your only legal option.

The ActiveX Diagnostic Utility can return licensing information for a control. See “[Testing an ActiveX control](#)” in the “General Utilities” chapter of *Synergy Language Tools* for details.

Debugging

The ActiveX API supports a debugging system to enable you to analyze problems in your code. Set the AXDEBUG environment variable as follows to use the debugging system:

To	Set AXDEBUG to
Enable the debug display system	YES
Enable the debugging system and log events to a file	FILE= <i>filename</i>
Enable debugging system and send debug information to an attached Windows debugger	EXTERNAL

On 32-bit Windows, when the debugging system is enabled by setting AXDEBUG to YES, the creation of the first ActiveX container creates a debug display, which consists of a tree-view representation of the Synergy Language/ActiveX system and a log of the related operations. (See [figure 10-1](#).)

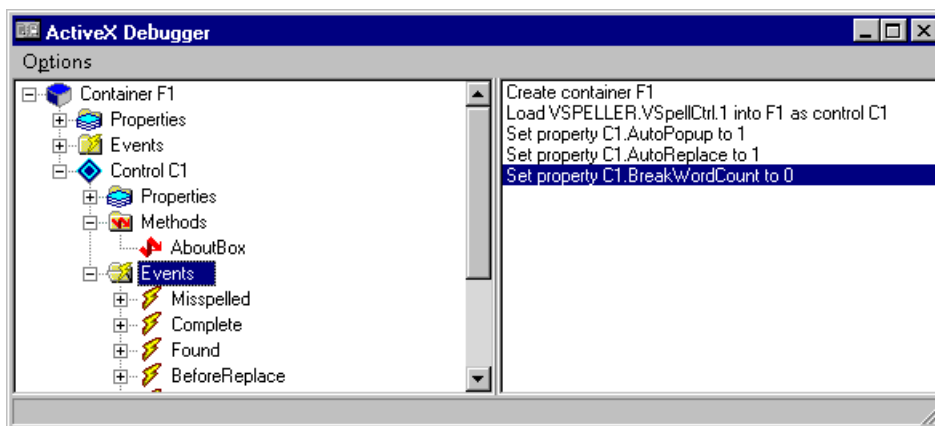


Figure 10-1. The ActiveX debug display.

The tree view is in a standard layout:

- ▶ The outer nodes of the tree are the containers.
- ▶ Nodes beneath a container are the ActiveX controls loaded into that container.
- ▶ Each control has method, property, and event “folder” nodes beneath it.
- ▶ The individual methods have nodes within the method folder, and argument nodes are beneath each method.

A log window is displayed in conjunction with the tree view. As Synergy Language/ActiveX operations occur, a description of each operation is written to the log. You can scroll through the log, as well as specify the maximum number of log entries. See [AXDEBUG](#) in the “Environment Variables and System Options” chapter of *Synergy Language Tools* for more information.

You can also turn on debug mode programmatically if no containers have been created yet. To do so, make an empty call to `AX_CREATE` and specify `-1` as the parent window, as shown below:

```
xcall ax_create(,,, -1)
```

This call does not create a container; instead, it serves the same purpose as setting `AXDEBUG` to `YES` before the program is run. When you call `%AX_CREATE` a second time to actually generate the container, it starts up in debug mode.

Sample program

Other samples are included in the `dbl\examples` directory of your Synergy Language distribution.

```

;=====
; Module:      axmusic.dbl
;
; Facility:    Synergy Language ActiveX demo
;
; Description: Synergy Language program demonstrating ActiveX support
;
; $Revision:   1.0   $
;
; $Date:      Jun 04 1997 13:45:42   $
;=====
main axmusic
.include "DBLDIR:ActiveX.def"
.define AX_XPOS      ,200
.define AX_YPOS      ,200
.define AX_HEIGHT    ,500
.define AX_WIDTH     ,740

record
    axwid      ,i4                ;ID of ActiveX container object
    axid       ,i4                ;ID of ActiveX control object

proc
    xcall flags(7000000)          ;Disable stop msg
    axwid = %ax_create(AX_XPOS, AX_YPOS, AX_WIDTH, AX_HEIGHT)
                                ;Create the container
    axid = %ax_load(axwid, "AXDemoMusic.AXDemo2",
&                                0, 0, AX_WIDTH, AX_HEIGHT)      ;Load the control
    xcall ax_bind(axid, "OnLoading", "OnLoading")                ;Bind events
    xcall ax_bind(axid, "OnGetDetail", "OnGetDetail")

```

```
        xcall ax_bind(axid, "OnClose", "OnClose")
        xcall ax_show(axwid)           ;Show the container
        xcall win_proc(1)              ;Process Activex controls
        stop
endmain

subroutine OnLoading
;
; Description: Load from file event
;
        axid      ,n              ;ID of ActiveX control
        FileName ,a              ;Filename to open

record
        music_rec,a127

proc
        open(10, u:i, FileName)[err=openerr]           ;Open the demo file

        repeat
                begin
                        reads(10, music_rec, eof)
                        xcall ax_call(axid, "AddRecord", music_rec) ; and pass to
                                                                    ; control
                end

eof,
        xcall ax_set(axid, "FindBtnEnabled", 1)         ;Enable the find button
        close 10

openerr,
        xreturn
endsubroutine

subroutine OnGetDetail
;
; Description: ActiveX requests a detail record
;
        axid      ,n              ;ID of ActiveX control
        KeyString ,a              ;Key of detail record

record
        filchn    ,i4
        detail_rec,a333
```



```
proc
    open(11, u:i, "detail")[err=notfound]          ;Open the demo file
    read(11, detail_rec, KeyString)[eof=notfound,key=notfound]
    xcall ax_call(axid, "AddDetail", detail_rec)    ; and pass to
                                                    ; control
    close 11

notfound,
    xreturn
endsubroutine

subroutine OnClose
;
; Description: Closing down
;

proc
    xcall win_stop(1);Terminate ActiveX processing
    xreturn
endsubroutine
```

Synergy ActiveX API Routines

The majority of the ActiveX API routines begin with %AX_ and are documented on the pages that follow. Toolkit routines are documented in the “[ActiveX Routines](#)” chapter of the *UI Toolkit Reference Manual*.

AX_BIND – Bind a callback subroutine to an ActiveX event

AX_BIND

WIN

```
xcall AX_BIND(id, callback_event[, function])
```

or

```
return = %AX_BIND(id, callback_event[, function])
```

Return value

return

%AX_BIND always returns a value of 1. (i)

Arguments

id

The ActiveX control ID returned by an %AX_LOAD call or the ActiveX container ID returned by an %AX_CREATE call. (**D_ADDR**)

callback_event

A string that contains the name of a callback event for the ActiveX control if *id* is a control ID. If *id* is a container ID, one of the following events: (**a**)

OnFocus	The window has focus; keyboard events go to the window. (Note that a container rarely receives focus.)
OnUnfocus	The window lost focus.
OnClick	The window was clicked.
OnDblClick	The window was double-clicked.
OnResize	The window was resized.
OnShow	The window's visible flag was set so the window is no longer hidden.
OnHide	The window was hidden.
OnClose	The window was closed.
OnDestroy	The window was rendered useless and is no longer available.
OnActivate	A component of the window has focus, which has activated the window.
OnDeactivate	No component of the window has focus.
OnOriginChange	The origin (upper-left cell) of an ActiveX list was changed.

function

(optional) A string that contains the name of the function to call when the event occurs. (**a**)

Discussion

Once a callback routine has been bound to an ActiveX event, that routine is called whenever the associated event occurs within the control. In other words, `%AX_BIND` establishes a Synergy Language routine as the supporting method for an ActiveX event. A callback routine should not perform any extended processing or processing that requires user input. Doing so may cause the application to lock up. For instance, rather than calling a Toolkit input routine within the callback, you should use `M_SIGNAL` to transfer the event to the outer input loop.

Be aware that when an `OnFocus` event is triggered while you're debugging an `OnFocus` event method, the debug window receives focus.



If you're familiar with Windows programming, don't confuse a Windows event with an ActiveX event. A Windows event is directly associated with low-level windows "message traffic," whereas an ActiveX event is entirely defined by a control's designer. Confusion can arise because ActiveX events that are driven by and have the same name as Windows events are often created. (`OnClick` and `OnClose` are common examples.)

The `OnClick` and `OnDbtClick` events pass a `VT_I4` bit flag argument that indicates which mouse button and what other keys are present. The `OnResize` event passes a `VT_I4` flag that indicates what kind of resizing occurred. The **activex.def** file contains definitions for these button and resize flags.

The `OnClose` and `OnDestroy` events always occur as a pair. `OnClose` always occurs first.

The `OnOriginChange` event is for ActiveX lists only. See the "[List Routines](#)" chapter of the *UI Toolkit Reference Manual* for details.

Whether or not the return value for the specified *function* is used depends on what return value (if any) is expected by the ActiveX control. If the *function* argument is not passed, the bind is cleared.

We recommend that you `XCALL AX_BIND` instead of accessing it as a function.

Note that you can also bind a routine automatically by specifying the *autobind_prefix* argument to `%AX_CREATE` or `%AX_LOAD`. See [%AX_CREATE on page 10-16](#) or [%AX_LOAD on page 10-22](#).

Examples

See the example that begins on [page 10-9](#).

%AX_CALL – Call a method within an ActiveX control

WIN, DLL
WIN

value = %AX_CALL (*id*, *method_name*[, *arguments*, ...])

or

xcall AX_CALL (*id*, *method_name*[, *arguments*, ...])

Return value

value

The value returned by the method. If this value cannot be successfully converted to the natural integer for the platform (**i8** on 64-bit platforms or **i4** on 32-bit platforms), %AX_CALL returns 0, and the value returned by the method is ignored. (**i**)

Arguments

id

The ID of the ActiveX container or control to reference. (**D_ADDR**)

method_name

A string that contains the name of the method to call. (**a**)

arguments

(optional) One or more arguments to be passed to the specified method.

Discussion

%AX_CALL attempts to convert all optional arguments into the type of data that the method expects. It is your responsibility to ensure that the type of the data passed has meaning within the context of the targeted conversion. Failure to do so will yield undefined results.

Any string arguments that are used by a “set” call (for example, SetHeaderCellFont) may need to be trimmed. %AX_CALL differentiates between strings with and without trailing spaces.

Examples

See the example that begins on [page 10-9](#).

%AX_CREATE – Create an ActiveX container

%AX_CREATE
WIN

container_id = %AX_CREATE ([*xpos*], [*ypos*], [*xsize*], [*ysize*], [*parent*][, *autobind_prefix*])

Return value

container_id

A positive, nonzero ID for the ActiveX container. The default string is “Container_*nn*”, where *nn* may be 1, 2, 3, and so forth in the order of containers created. (**D_ADDR**)

Arguments

xpos

(optional) The X pixel screen position of the upper-left corner of the ActiveX container. (**n**)

ypos

(optional) The Y pixel screen position of the upper-left corner of the ActiveX container. (**n**)

xsize

(optional) The pixel width of the ActiveX container. (**n**)

ysize

(optional) The pixel height of the ActiveX container. (**n**)

parent

(optional) The handle of a parent window for the ActiveX container. (**n**)

autobind_prefix

(optional) A routine prefix that causes automatic binding to occur. If this argument is present, each event is automatically bound to any existing routine whose name is the specified prefix followed by the event’s name. (**a**)

Discussion

The default values for *xpos*, *ypos*, *xsize*, and *ysize* are 200, 150, 400, and 300, respectively. These values together form a centered, half-screen window in an 800-by-600 resolution environment.

Specifying a parent window enables you to make the container you’re creating a “slave” to another window. The value that you pass in *parent* is a standard system window handle. If the parent window is visible, the container is visible on creation. Likewise, if the parent window is hidden, the container is hidden.

Specifying the *autobind_prefix* argument eliminates the need to manually bind each callback subroutine using AX_BIND.

Other than memory, there's no limit to the number of containers that can be created simultaneously.

Examples

See the example that begins on [page 10-9](#).

%AX_DELETE – Delete an ActiveX object

%AX_DELETE

WIN

return = %AX_DELETE(*id*)

or

xcall AX_DELETE(*id*)

Return value

return

%AX_DELETE always returns 0. (i)

Arguments

id

One of the following: (**D_ADDR**)

container_id The ID of an ActiveX container.

control_id The ID of a loaded ActiveX control.

D_AX_ALL All containers.

Discussion

If *id* is D_AX_ALL, all containers are deleted in the reverse order in which they were created.

When a container is deleted by this operation, it is first hidden, and then all controls in the container are deleted in the reverse order in which they were loaded.

AX_GET – Get a property value for an ActiveX object

AX_GET
WIN

`xcall AX_GET (id, property, current_value[, parameters])`
 or
`return = %AX_GET (id, property, current_value[, parameters])`

Return value

return
 %AX_GET always returns a value of 1. **(i)**

Arguments

id
 The ID of the container or control for which you're getting a property. **(D_ADDR)**

property
 A string that contains the name of the property to access. See [“Properties” on page 10-4](#) for information about supported properties. **(a)**

current_value
 A field that will be loaded with the current value for the property. **(a)**

parameters
 (optional) Up to eight additional parameters required by a property. **(n)**

Discussion

It is the caller's responsibility to ensure that the data type of *current_value* has meaning within the context of the specified property. If it does not, the results are undefined.

We recommend that you XCALL AX_GET instead of accessing it as a function.

Some controls support indexed properties, which are stored as arrays in a control and have one or more indices. If the property is an indexed property, *parameters* must be the index values of the array element to reference. The user documentation for a given control should specify when a property is indexed.

The *parameters* argument can only pass integer parameters. It cannot be used to pass additional parameters for a property that expects alpha parameters.

%AX_GETINT – Get an integer property value for an ActiveX object

%AX_GETINT**WIN**

value = %AX_GETINT (*id*, *property*[, *parameters*])

Return value

value

The current value for the property. (**i**)

Arguments

id

The ID of the container or control for which you're getting a property. (**D_ADDR**)

property

A string that contains the name of the property to access. (**a**)

parameters

(optional) Up to eight additional parameters required by a property. (**n**)

Discussion

Property should reference a property whose value has an integer representation. If this is not the case, the return value is undefined.

Some controls support indexed properties, which are stored as arrays in a control and have one or more indices. If the property is an indexed property, *parameters* must be the index values of the array element to reference. The user documentation for a given control should specify when a property is indexed.

The *parameters* argument can only pass integer parameters. It cannot be used to pass additional parameters for a property that expects alpha parameters.

AX_HIDE – Hide an ActiveX container

```
AX_HIDE
WIN
xcall AX_HIDE(id)

or

return = %AX_HIDE(id)
```

Return value

return
%AX_HIDE always returns a value of 1. (i)

Arguments

id
The ID of the container to hide. (**D_ADDR**)

Discussion

The AX_HIDE routine is equivalent to calling AX_SET(*id*, "EXT_VISIBLE", 0).
We recommend that you XCALL AX_HIDE instead of accessing it as a function.

%AX_LOAD – Load an ActiveX control into a container

%AX_LOAD**WIN**

```
control_id = %AX_LOAD(container_id, control_name, [xpos], [ypos], [xsize], [ysize], [license]  
& [ , autobind_prefix])
```

Return value

control_id

The positive, nonzero ID of the loaded ActiveX control. (**D_ADDR**)

Arguments

container_id

The ActiveX container ID returned by an %AX_CREATE call. (**D_ADDR**)

control_name

A string that contains the identifying name of the ActiveX control. (**a**)

xpos

(optional) The X pixel position within the container of the upper-left corner of the ActiveX control. (**n**)

ypos

(optional) The Y pixel position within the container of the upper-left corner of the ActiveX control. (**n**)

xsize

(optional) The pixel width of the ActiveX control. (**n**)

ysize

(optional) The pixel height of the ActiveX control. (**n**)

license

(optional) The license string required to run the ActiveX control at an unlicensed site. (**a**)

autobind_prefix

(optional) A routine prefix that causes automatic binding to occur. If this argument is present, each event is automatically bound to any existing routine whose name is the specified prefix followed by the event's name. (**a**)

Discussion

The default value for *xpos* and *ypos* is zero, which positions the ActiveX control at the upper-left corner of the container. If *xsize* or *ysize* is not specified, the default is the respective edge of the container's display area. (The display area is the total area of the container minus all borders, headers, and so forth.)

The *control_name* and *license* strings can be determined using the ActiveX Diagnostic Utility. See [“Testing an ActiveX control”](#) in the “General Utilities” chapter of *Synergy Language Tools* for more information.

Specifying the *autobind_prefix* argument eliminates the need to manually bind each callback subroutine using `AX_BIND`.

Examples

See the example that begins on [page 10-9](#).

AX_NEWVAL – Modify an ActiveX callback routine’s parameter

AX_NEWVAL**WIN**

```
xcall AX_NEWVAL (argument, new_value)
```

or

```
return = %AX_NEWVAL (argument, new_value)
```

Return value

return

%AX_NEWVAL always returns a value of 1. (i)

Arguments

argument

The argument of a callback routine. (This is not a string that contains the name of the argument but the argument itself.) (**a** or **n**)

new_value

The new value to be assigned to that argument. (**a** or **n**)

Discussion

All arguments that are passed to a bound callback routine are read-only and cannot be modified directly. Usually this situation presents no problems, because most callbacks that are defined by ActiveX controls either have no parameters or, if they do have parameters, the parameters are for information access only. Some ActiveX controls, however, require that information be returned to them by one or more callback parameters. In this case, you can use AX_NEWVAL.

Note that *argument* differs from the other name parameters in the ActiveX API, which specify the name as a string. For example, suppose you have the numeric argument **level** defined for a callback routine. To change the value to 5, you might use the following call:

```
xcall ax_newval(level, 5)
```

but *not*

```
xcall ax_newval("level", 5)
```

Any parameter modifications made by AX_NEWVAL are not returned to the calling ActiveX control until the callback routine returns.

If you call AX_NEWVAL for something that passes a VT_BSTR argument, the length of the string changes to the length of *new_value*. AX_NEWVAL may also change the data type of an argument if the ActiveX control defines the argument as VT_VARIANT. Variant data types must be passed by reference.

ADVANCED

When developing ActiveX controls in C++, Delphi, or other native languages, be sure to pass modifiable arguments as variants that have the VT_BYREF flag set. If the reference is to another variant, that variant must also have VT_BYREF, and so on. In other words, the argument cannot resolve to a value-type variant at any level. The ultimate dereference must resolve to a primitive type. Thus, a variant of type VT_BYREF|VT_I4 will work, but one of type VT_BYREF|VT_VARIANT will generate a “Writing to a literal” error unless the variant referred to is also a VT_BYREF variant.

We recommend that you XCALL AX_NEWVAL instead of accessing it as a function.

AX_SET – Set a property in an ActiveX object

AX_SET**WIN**

```
xcall AX_SET(id, property, new_value[, parameters])
```

or

```
return = %AX_SET(id, property, new_value[, parameters])
```

Return value

return

%AX_SET always returns a value of 1. (**i**)

Arguments

id

The ID of the container or control for which you're setting a property. (**D_ADDR**)

property

A string that contains the name of the property to access. See [“Properties” on page 10-4](#) for information about supported properties. (**a**)

new_value

The new value for the property. (**a** or **n**)

parameters

(optional) Up to eight additional parameters required by a property. (**n**)

Discussion

It is the caller's responsibility to ensure that the data type of *new_value* has meaning within the context of the specified property. If it does not (for example, if you specify an alpha string for a numeric property), the results are undefined.

We recommend that you XCALL AX_SET instead of accessing it as a function.

Some controls support indexed properties, which are stored as arrays in a control and have one or more indices. If the property is an indexed property, *parameters* must be the index values of the array element to reference. The user documentation for a given control should specify when a property is indexed.

The *parameters* argument can only pass integer parameters. It cannot be used to pass additional parameters for a property that expects alpha parameters.

Examples

See the example that begins on [page 10-9](#).

AX_SHOW – Show an ActiveX container

AX_SHOW**WIN**

`xcall AX_SHOW(id)`

or

`return = %AX_SHOW(id)`

Return value

return

%AX_SHOW always returns a value of 1. (**i**)

Arguments

id

The ID of the container to show. (**ID_ADDR**)

Discussion

The AX_SHOW subroutine is equivalent to calling AX_SET(*id*, "EXT_VISIBLE", 1).

We recommend that you XCALL AX_SHOW instead of accessing it as a function.

Examples

See the example that begins on [page 10-9](#).

AX_TIMEOUT – Set the time-out value for an ActiveX control

AX_TIMEOUT

WIN

```
xcall AX_TIMEOUT(wait_time)
```

Arguments

wait_time

The input time-out (or “wait”) in seconds for I/O processing to be completed before returning to the calling routine. (**n**)

D_IMMEDIATE or **0** Time out immediately.

D_FOREVER or **-1** Never time out.

D_GLOBAL or **-2** Use the global value. (default)

#seconds Wait up to *#seconds* for I/O processing to be complete.

Discussion

The AX_TIMEOUT routine changes the time-out value for the current invocation of AX_INPUT. Calling this routine resets any existing timer to the time-out specified, effective immediately.

See also

[AX_INPUT](#) in the “ActiveX Routines” chapter of the *UI Toolkit Reference Manual* for setting the time-out value when performing input to an ActiveX control.

%WIN_PROC – Process system-level Windows messages

%WIN_PROC**WIN**

```
exit = %WIN_PROC ([wait_flag])
```

Return value

exit

The exit value of the function. (**n**)

Arguments

wait_flag

(optional) The “wait for completion” flag. (**n**)

Discussion

If *wait_flag* is not specified or has a value of 0, any currently pending Windows messages are processed. Once all messages are processed, %WIN_PROC returns with a value of 0.

If *wait_flag* is specified and has a nonzero value, the program enters a “wait for Windows messages” mode. All pending Windows messages are processed, and any further messages are waited for and processed. This continues until a %WIN_STOP function is executed. (See [%WIN_STOP on page 10-31](#) for more information.) At this time, the value specified by the %WIN_STOP request is returned.

Wait modes cannot be nested. If a wait mode is already in effect from a currently active %WIN_PROC call, a fatal error is generated and the running Synergy Language program is stopped.

%WIN_PROC is used as part of the ActiveX API, but you can use it in other contexts as well. For example a call to %WIN_PROC with a *wait_flag* of 0 is the same as doing an EM_YIELD.

%WIN_PROC requires **axlib.elb**.

%WIN_STOP – Suspend Windows system processing

%WIN_STOP

WIN

status = %WIN_STOP (*proc_value*)

Return value

status

The code D_ERR_SUCCESS. (**n**)

Arguments

proc_value

An expression that specifies the value to return for the currently active %WIN_PROC call. (**n**)

Discussion

%WIN_STOP suspends the processing of any Windows system messages.

11

Synergy DLL API

Introduction to the Synergy DLL API 11-2

Describes the Synergy DLL API, which enables you to integrate DLLs into your application.

Synergy DLL API Routines 11-3

This API includes the following routines:

%DLL_CALL – Call a DLL routine	11-4
%DLL_CLOSE – Close a Dynamic Link Library	11-9
%DLL_OPEN – Open a Dynamic Link Library	11-10

Introduction to the Synergy DLL API

The Synergy DLL API enables you to integrate DLLs into your application. A DLL (dynamic link library) is a library of executable functions or data that your Windows-based application can call during execution.

You can either write your own DLL from scratch or acquire one of the many existing DLLs from an external source.



DLLs created with Visual Basic are not accessible by Synergy. Synergy can only access DLLs written in a language that can create standard (non-ActiveX) DLLs.

WIN

DLL files usually have one of the following extensions: **.dll**, **.exe**, **.drv**, or **.fon**.

UNIX

DLL files always have the extension **.so**.

To integrate a DLL into your application,

1. Create or otherwise acquire a DLL. Refer to your operating system documentation for more information about creating a DLL.
2. If you've created a new DLL, explicitly export the routines that your Synergy program will call. How you export the routines depends on which tool you used to create it. Refer to the documentation for your development tool for instructions.
3. Write a routine that uses the %DLL_ functions listed above to access the routines in the DLL.

Let's say we want to add a splash screen to our application. Our routine might look like this:

```
record
    splash_dll      ,D_ADDR
    splash_form     ,D_NATLNG

proc
    splash_dll = %dll_open("SYNBIN:synsw")      ;Open the splash DLL
; This DLL routine creates and displays the splash screen
    splash_form = %dll_call(obj.splash_dll,, "Create_SW")
; Do whatever you want here while the splash screen is displayed
; This DLL routine removes and deletes the splash screen
    xcall dll_call(obj.splash_dll,, "Close_SW", splash_form)
    xcall dll_close(splash_dll)                ;Close the splash DLL
end
```


Synergy DLL API Routines

The DLL API consists of the following functions, which are described on the pages that follow:

- ▶ [%DLL_CALL](#)
- ▶ [%DLL_CLOSE](#)
- ▶ [%DLL_OPEN](#)

%DLL_CALL – Call a DLL routine

WIN, UNIX

value = %DLL_CALL (*handle*, [*convention*], *func_name*[, *arg*, ...])

Return value

value

The value of the DLL function as an integer data type. (**D_NATLNG**)

Arguments

handle

The handle to the DLL, as returned from %DLL_OPEN. (**i**)

convention

One of the following calling conventions: (**n**)

DLL_TYPE_C Use C calling convention.

DLL_TYPE_DBLCALL Use Synergy runtime-compatible calling convention (C).

DLL_TYPE_FASTCALL Use __fastcall calling convention.

DLL_TYPE_STDCALL Use __stdcall calling convention.

DLL_TYPE_WINAPI Use WINAPI calling convention.

func_name

The name of the function to be called within the DLL. (**a**)

arg

(optional) One or more variables used to pass alpha or integer data types to the function in the DLL. Arguments of integer data type can be one, two, or four bytes long (or eight bytes on 64-bit systems).

Discussion

The %DLL_CALL function calls a subroutine or function in a DLL.

Before invoking %DLL_CALL, check the documentation for your DLL to find out how the function is declared, which data type the function will return, which calling convention to use, and which arguments are necessary. If you call a DLL subroutine with improper arguments, the result is undefined. The tables on [pages 11-5](#) and [11-6](#) provide additional information about the return value of the DLL function and the arguments expected by the DLL function.

DLL_TYPE_C and **DLL_TYPE_DBLCALL** are the only calling conventions available on UNIX. The default calling convention on UNIX is **DLL_TYPE_C**. The default calling convention on Windows is **DLL_TYPE_STDCALL**.



DLL_TYPE_STDCALL and **DLL_TYPE_WINAPI** are interchangeable.

A DLL may require an underscore as a prefix to the function name. In such a case, *func_name* must have an underscore as its first character as well. For example:

`_example`

You can pass up to 20 *arg* variables. The number and type of *arg* variables depends on the number and type that the DLL function expects. These *arg* variables are automatically converted from Synergy Language descriptors into C data types of four-byte integers, pointers, or null-terminated strings for use by the DLL. %DLL_CALL only passes the number of arguments specified.



Arg cannot be passed as a numeric literal. If passed as a numeric literal, it is handled as a decimal, which is not supported. To pass a literal value, use %integer(*value*, 4). See the example on [page 11-6](#).

Use ^ADDR as an *arg* variable to create a pointer to obtain the address of a Synergy Language variable. (See ^ADDR on [page 9-17](#) for more information.)

If the DLL function modifies an alpha argument, the modified string overwrites the unmodified string. If the modified string is longer than the unmodified string, the modified string is truncated.

The following table lists how a variable should be declared in Synergy Language based on the argument expected by the DLL function. The “Argument expected by the DLL function” column lists the argument types. The “Variable declared in Synergy” column shows which Synergy Language data types are allowed for the argument types.

Argument expected by the DLL function	Variable declared in Synergy
int, long, long int, unsigned long, dword	i1, i2, i4, ^ADDR()
int *, void *, any other pointers (except pointers to null-terminated character strings)	^ADDR()
char *, null-terminated character string	a
short, short int, unsigned short, word	i1, i2
char, unsigned char, byte	i1
int **, void **, char ** (pointer to a pointer)	D_ADDR

The table below lists how a variable should be declared in Synergy Language based upon the return value of the DLL function. The “Return value of the DLL function” column lists the return value types. The “Variable declared in Synergy” column shows which Synergy Language data type to declare for the return value.

Return value of the DLL function	Variable declared in Synergy
int, int *, char *, long int, any other pointers (including null-terminated character strings)	i4 (long int is i8 on 64-bit machines)
unsigned long	i4 (or i8 on 64-bit machines) – Accessed in Synergy Language as %UNSIGNED
short int	i2
unsigned short	i2 – Accessed in Synergy Language as %UNSIGNED
char	i1
unsigned char	i1 – Accessed in Synergy Language as %UNSIGNED

WIN

%SYSERR is updated from the GetLastError value after every %DLL_CALL. You cannot call GetLastError directly, as the value is overwritten in making the call.

Examples

The following example uses %integer(0,4) to pass an integer 0:

```
record
    handle      ,i4
    ret         ,i4
    text        ,a40    ,"This is the text"
    title       ,a40    ,"This is the title"

proc
    handle = %dll_open("user32.dll")      ;Open the 32-bit DLL. It is part
                                           ; of Win32s
    ret = %dll_call(handle, DLL_TYPE_WINAPI, "MessageBoxA",
    &                                     %integer(0, 4), text, title, %integer(0, 4))
                                           ;For details on "MessageBoxA"
                                           ; check your Windows API doc
                                           ;
```

```

                                ;Using XCALL
    xcall dll_call(handle, DLL_TYPE_WINAPI, "MessageBoxA", %integer(0, 4),
&    text, title, %integer(1, 4))
    ret = %dll_close(handle)      ;Close the DLL
end

```

Pointers as return values

Some DLL functions return a pointer to a variable. In order to access that variable, the Synergy Language program has to de-reference the pointer. De-referencing is the inverse operation to ^ADDR and can be accomplished using Synergy Language's dynamic memory management routines. Only pointers that are returned by DLL functions can be de-referenced and used by a Synergy Language program.

Examples

The following example illustrates how to de-reference the pointer:

```

struct data {
    long int num1;
    char text[100]; }           //The struct may contain any datatype.
                                // When calling 16-bit DLLs, pointers are
                                // not allowed as structure members.

struct data *examplefunc5(long arg1);
struct data **examplefunc6(long arg1);

```

Calling a DLL from Synergy Language:

```

structure datastruct           ;Describes layout of data record.
    num1           ,i4
    text           ,a100
record
    mem_hand       ,i4
    dll_hand       ,i4
    pointer        ,D_ADDR
    arg1           ,i4
    retsts         ,i4
record datarec
    number         ,i4
    str            ,a100

;The DLL function returns a pointer to the struct called data.
pointer = %dll_call(dll_hand,, "examplefunc5", arg1)

;Alternative: The DLL function returns a pointer to a pointer.
retsts = %dll_call(dll_hand,, "examplefunc6", arg1)

```

```
;First step: DM_REG converts the returned pointer to a Synergy Language
; memory handle.
    mem_hand = %mem_proc(DM_REG, 104, pointer)
                                ;104 is size of struct data in bytes.

;Second step: ^m() converts the memory handle to a usable Synergy Language
; record.
    datarec = ^m(datastruct, mem_hand)
                                ;Datastruct tells ^m() about expected
                                ; data layout.
    writes(1, %string(datarec.number))
    writes(1, datarec.str)
```

Also see the CodeExchange at <http://www.synergex.com> for another example.

Passing structured data types

Structured data types can be passed to a DLL function by reference. To do so, a corresponding record must be declared in the Synergy Language application. In the function call %DLL_CALL, ^ADDR is used to obtain a reference to a record.

Examples

The following example shows how to pass a pointer to a structured data record as an argument to a DLL function:

```
struct data {
    int num1;
    char text[100]; }      //The struct may contain any data types
long int CDECL examplefunc4(struct data *arg1);

record datarec
    num1      ,i2
    text      ,a100

retlong = %dll_call(handle,, "examplefunc4", ^addr(datarec))
                                ;retlong declared as i4
```

%DLL_CLOSE – Close a Dynamic Link Library

WIN, UNIX

status = %DLL_CLOSE (*handle*)

Return value

status

An integer value of 1 if %DLL_CLOSE is successful. If unsuccessful, the error \$ERR_DLLCLSERR is generated. (i)

Arguments

handle

The handle to the DLL, as returned from %DLL_OPEN. (i4)

Discussion

The %DLL_CLOSE function closes a DLL that was opened with %DLL_OPEN.

Make sure you close each DLL only once for each DLL_OPEN. If you close a DLL twice in one part of an application, the rest of the application may assume the DLL is still open and available but can no longer access it. This situation will leave the rest of the application in an unstable state. Pay particular attention whenever a system DLL (for example, **user32.dll** on Windows) is accessed, because the runtime also uses these DLLs.

Each DLL opened by a Synergy application should be closed with %DLL_CLOSE before the Synergy application is exited. If %DLL_CLOSE is not called and the customer does a program chain, the resources are not freed. Calls across program chains can use the same DLL handle.

When the runtime ends, whether normally or abnormally, it automatically closes all open DLLs.

Examples

For an example of %DLL_CLOSE, see the examples for %DLL_CALL, which begin on [page 11-6](#).

%DLL_OPEN – Open a Dynamic Link Library

WIN, UNIX ---

handle = %DLL_OPEN(*filename*[, *cinterface*])

Return value

handle

The handle to the DLL if %DLL_OPEN is successful. If unsuccessful, the error \$ERR_DLLOPNERR is generated. (**D_ADDR**)

Arguments

filename

The full name and path of the DLL to be opened. (**a**)

cinterface

(optional) A flag that tells %DLL_OPEN to initialize the Synergy Language C interface in the DLL being opened. Any value is valid. (**a** or **n**) (Synergy standard only)

Discussion

The %DLL_OPEN function opens a DLL. Multiple DLLs can be opened simultaneously; the number you can open is limited only by your system's available memory.

If the call to %DLL_OPEN fails, you can use %SYSERR to get the system error code.



DLLs created with Visual Basic are not accessible by Synergy. Synergy can only access DLLs written in a language that can create standard (non-ActiveX) DLLs.

WIN ---

If no path is specified within *filename*, the system searches for the DLL in the following directories in the following order:

- ▶ DBLDIR\bin directory
 - ▶ Current directory
 - ▶ Windows system directory (usually \windows\system)
 - ▶ Windows directory (usually \windows)
 - ▶ Directories listed in the PATH environment variable
-

UNIX

If no path is specified within *filename*, the system searches the path specified by the LD_LIBRARY_PATH environment variable (which is called LIBPATH on AIX and SHLIB_PATH on HP-UX 32-bit). This environment variable was automatically set if you used the **setsde** script to set up your Synergy environment.

Specifying the *cinterface* argument tells %DLL_OPEN to call a setup routine with the correct arguments to initialize the Synergy Language C interface in the DLL. This is to assist with user-created DLLs written in C that use the C interface. See [“Building a Dynamic Library with C Routines” on page 21-27](#) for more details.

Each DLL opened by a Synergy application should be closed with %DLL_CLOSE prior to exiting the Synergy application.

Examples

For an example of %DLL_OPEN, see the examples for %DLL_CALL which begin on [page 11-6](#).

12

Synergy Floating Point API

The Synergy floating point API is not available on Synergy .NET.

Introduction to the Synergy Floating Point API 12-2

Describes the steps you must follow to perform floating point calculations and provides an example program.

Synergy Floating Point API Routines 12-4

This API includes the following routines:

FP_ADD – Add floating-point numbers.....	12-5
FP_ARCCOS – Return the arc cosine of a floating-point number.....	12-6
FP_ARCSIN – Return the arc sine of a floating-point number.....	12-7
FP_ARCTAN – Return the arc tangent of a floating-point number.....	12-8
FP_CMP – Compare floating-point numbers	12-9
FP_COS – Return the cosine of a floating-point number.....	12-10
FP_DIV – Divide floating-point numbers	12-11
FP_EXPE – Exponentiate a floating-point number.....	12-12
FP_FROM_NUM – Convert to floating-point from implied-decimal	12-13
FP_LOG10 – Return the base 10 logarithm of a floating-point number	12-14
FP_LOGE – Return the logarithm of a floating-point number.....	12-15
FP_MUL – Multiply floating-point numbers	12-16
FP_POW – Raise a floating-point number to a floating-point power	12-17
FP_SIN – Return the sine of a floating-point number	12-18
FP_SQRT – Get the square root of a floating-point number.....	12-19
FP_SUB – Subtract floating-point numbers	12-20
FP_TAN – Return the tangent of a floating-point number.....	12-21
FP_TO_NUM – Convert from floating-point to implied-decimal	12-22

Introduction to the Synergy Floating Point API

The floating point API consists of single- and double-operand routines that enable you to perform floating-point calculations. Because DIBOL has no native floating-point data type, we have implemented this support as routines, and data (the binary representation of the **double** result) is stored in **a8** fields.

The floating-point routines require that you follow these steps to perform floating-point calculations:

1. Initialize your floating-point **a8** fields using the `FP_FROM_NUM` subroutine.
2. Perform the calculation using the appropriate `FP_` routine.
3. Return the floating-point **a8** field to usable data using the `FP_TO_NUM` subroutine.

Sample program

The following example contains several of the `FP_` routines.

```
subroutine calc_stats
;Returns mean and standard deviation for a set of numbers. This routine
;requires that the set's sum and sum of squares have been calculated and
;are passed as arguments. This eliminates the need for the caller to
;build an array of the set -- handy for report writers etc.

    a_size   ,n           ;Number of entries
    a_sum    ,n           ;Sum of the entries
    a_sqs    ,n           ;Sum of the squares
    a_mean   ,n           ;Returned mean
    a_stdd   ,n           ;Returned standard deviation
    a_sample ,n           ;(Optional) flag. If TRUE, requests an
                           ; unbiased standard deviation -- use this
                           ; if estimating for the population based on
                           ; the sample provided.

record
    fval     ,a8
    fn_1     ,a8           ;Needed for nonbiased calculation
    fmean    ,a8
    fsum     ,a8
    fsqs     ,a8
    fstdd    ,a8
```

```

proc
  if (a_size .lt. 2)
    begin
      if (a_size .eq. 1) then
        a_mean = a_sum
      else
        a_mean = 0
        a_stdd = 0
      xreturn
    end
    ;Get count, sum and sum of squares into floating-point
    ; variables for the standard deviation calculation:
    xcall fp_from_num(fval, %implied(a_size))
    xcall fp_from_num(fsum, %implied(a_sum))
    xcall fp_from_num(fsqs, %implied(a_sqs))
    if (^passed(a_sample) .and. a_sample) then
      begin
        ;Does user want a sample-based estimate?
        xcall fp_from_num(fn_1, %implied(a_size - 1))
        xcall fp_div(fsqs, fsqs, fn_1) ;Divide by n-1 instead of n
        xcall fp_div(fmean, fsum, fval) ; for samples (instead of
        xcall fp_mul(fsum, fsum, fsum) ; the whole population).
        xcall fp_mul(fval, fval, fn_1) ; The n-1 denominator yields
        xcall fp_div(fsum, fsum, fval) ; a larger estimate for small
      end
      ; samples.
    else
      begin
        ;This is the whole "population"
        xcall fp_div(fsqs, fsqs, fval)
        xcall fp_div(fsum, fsum, fval)
        fmean = fsum
        xcall fp_mul(fsum, fsum, fsum)
      end
      xcall fp_to_num(fmean, a_mean)
      xcall fp_sub(fval, fsqs, fsum)
      xcall fp_sqrt(fval, fval)
      xcall fp_to_num(fval, a_stdd)
      xreturn
    endsubroutine

```

Synergy Floating Point API Routines

The floating point API routines all begin with FP_ and are documented on the pages that follow.

FP_ADD – Add floating-point numbers

```
xcall FP_ADD(result, operand1, operand2)
```

Arguments

result

Returned with the result of adding *operand1* and *operand2*. (**a8**)

operand1

The first operand to be added. (**a8**)

operand2

The second operand to be added. (**a8**)

Discussion

The FP_ADD subroutine adds two floating-point numbers together.

The following equation is performed:

$$result = operand1 + operand2$$

Result, *operand1*, and *operand2* are native floating-point numbers represented as **a8** variables.

FP_ARCCOS – Return the arc cosine of a floating-point number

```
xcall FP_ARCCOS (result, operand)
```

Arguments

result

Returned with the arc cosine of *operand*. (**a8**)

operand

The radian value whose arc cosine will be returned. (**a8**)

Discussion

The FP_ARCCOS subroutine returns the arc cosine of a floating-point number.

The following operation is performed:

$result = \arccos(operand)$

Result and *operand* are native floating-point numbers represented as **a8** variables.

See also

[“Synergy Floating Point API Routines” on page 12-4.](#)

FP_ARCSIN – Return the arc sine of a floating-point number

```
xcall FP_ARCSIN(result, operand)
```

Arguments

result

Returned with the arc sine of *operand*. (**a8**)

operand

The radian value whose arc sine will be returned. (**a8**)

Discussion

The FP_ARCSIN subroutine returns the arc sine of a floating-point number.

The following operation is performed:

$result = \arcsin(operand)$

Result and *operand* are native floating-point numbers represented as **a8** variables.

FP_ARCTAN – Return the arc tangent of a floating-point number

```
xcall FP_ARCTAN(result, operand)
```

Arguments

result

Returned with the arc tangent of *operand*. (**a8**)

operand

The radian value whose arc tangent will be returned. (**a8**)

Discussion

The FP_ARCTAN subroutine returns the arc tangent of a floating-point number.

The following operation is performed:

$result = \arctan(operand)$

Result and *operand* are native floating-point numbers represented as **a8** variables.

FP_CMP – Compare floating-point numbers

```
xcall FP_CMP(result, float1, float2)
```

Arguments

result

Returned with one of the following values: (**n**)

- 0** *Float1* equals *float2*.
- 1** *Float1* is less than *float2*.
- 1** *Float1* is greater than *float2*.

float1

A native floating-point number to compare to *float2*. (**a8**)

float2

A native floating-point number to be compared to *float1*. (**a8**)

Discussion

The FP_CMP subroutine compares two floating-point numbers to see if they are equal or if one is greater than the other.

Result, *float1*, and *float2* are native floating-point numbers represented as **a8** variables.

This routine can also be called as a function.



Before you can use a subroutine as a function, you must first declare it external:

```
external function  
    fp_cmp    ,d
```

Then you can use FP_CMP as follows:

```
if (%fp_cmp(float1, float2)) then  
    writes(TTCHN, "not equal")  
else  
    writes(TTCHN, "equal")
```

FP_COS – Return the cosine of a floating-point number

```
xcall FP_COS(result, operand)
```

Arguments

result

Returned with the cosine of *operand*. (**a8**)

operand

The radian value whose cosine will be returned. (**a8**)

Discussion

The FP_COS subroutine returns the cosine of a floating-point number.

The following operation is performed:

$result = \cos(operand)$

Result and *operand* are native floating-point numbers represented as **a8** variables.

FP_DIV – Divide floating-point numbers

```
xcall FP_DIV(result, operand1, operand2)
```

Arguments

result

Returned with the result of dividing *operand1* by *operand2*. (**a8**)

operand1

The dividend. (**a8**)

operand2

The divisor. (**a8**)

Discussion

The FP_DIV subroutine divides one floating-point number by another.

The following equation is performed:

$$result = operand1 / operand2$$

Result, *operand1*, and *operand2* are native floating-point numbers represented as **a8** variables.

Examples

See the example on [page 12-4](#).

FP_EXPE – Exponentiate a floating-point number

```
xcall FP_EXPE(result, operand)
```

Arguments

result

Returned with the result of exponentiating the operand. (**a8**)

operand

The power to raise the base e. (**a8**)

Discussion

The FP_EXPE subroutine exponentiates the operand base e.

The following equation is performed:

$$result = e^{operand}$$

Result and *operand* are native floating-point numbers represented as **a8** variables.

FP_FROM_NUM – Convert to floating-point from implied-decimal

`xcall FP_FROM_NUM (float, number)`

Arguments

float

Returned with a native floating-point number (C “double” data type). (**a8**)

number

An implied-decimal number to convert to a floating-point number. (**n**)

Discussion

The FP_FROM_NUM subroutine converts an implied-decimal number to a machine-native, floating-point representation.

The following equation is performed:

floating = number

Examples

See the example on [page 12-4](#).

FP_LOG10 – Return the base 10 logarithm of a floating-point number

```
xcall FP_LOG10 (result, operand)
```

Arguments

result

Returned with the base 10 logarithm. (**a8**)

operand

The operand whose logarithm will be returned. (**a8**)

Discussion

The FP_LOG10 subroutine returns the base 10 logarithm of a floating-point number.

The following equation is performed:

$$result = \log_{10}(operand)$$

Result and *operand* are native floating-point numbers represented as **a8** variables.

FP_LOGE – Return the logarithm of a floating-point number

```
xcall FP_LOGE(result, operand)
```

Arguments

result

Returned with the natural logarithm. (**a8**)

operand

The operand whose logarithm will be returned. (**a8**)

Discussion

The FP_LOGE subroutine returns the natural (base e) logarithm of a floating-point number.

The following equation is performed:

$$result = \log_e(operand)$$

Result and *operand* are native floating-point numbers represented as **a8** variables.

FP_MUL – Multiply floating-point numbers

```
xcall FP_MUL(result, operand1, operand2)
```

Arguments

result

Returned with the result of multiplying *operand1* by *operand2*. (**a8**)

operand1

The multiplicand. (**a8**)

operand2

The multiplier. (**a8**)

Discussion

The FP_MUL subroutine multiplies two floating-point numbers.

The following equation is performed:

$$result = operand1 * operand2$$

Result, *operand1*, and *operand2* are native floating-point numbers represented as **a8** variables.

Examples

See the example on [page 12-4](#).

FP_POW – Raise a floating-point number to a floating-point power

```
xcall FP_POW(result, operand1, operand2)
```

Arguments

result

Returned with the result of raising *operand1* to the power of *operand2*. (**a8**)

operand1

The base. (**a8**)

operand2

The exponent. (**a8**)

Discussion

The FP_POW subroutine raises one floating-point number to the power of another.

The following equation is performed:

$result = pow(operand1, operand2)$

Result, *operand1*, and *operand2* are native floating-point numbers represented as **a8** variables.

FP_SIN – Return the sine of a floating-point number

```
xcall FP_SIN(result, operand)
```

Arguments

result

Returned with the sine of *operand*. (**a8**)

operand

The radian value whose sine will be returned. (**a8**)

Discussion

The FP_SIN subroutine returns the sine of a floating-point number.

The following operation is performed:

$result = \sin(operand)$

Result and *operand* are native floating-point numbers represented as **a8** variables.

FP_SQRT – Get the square root of a floating-point number

```
xcall FP_SQRT(result, operand)
```

Arguments

result

Returned with the result of taking the square root of *operand*. (**a8**)

operand

An operand whose square root will be returned. (**a8**)

Discussion

The FP_SQRT subroutine returns the square root of a floating-point number.

The following equation is performed:

$result = \text{sqrt}(operand)$

Result and *operand* are native floating-point numbers represented as **a8** variables.

Examples

See the example on [page 12-4](#).

FP_SUB – Subtract floating-point numbers

```
xcall FP_SUB(result, operand1, operand2)
```

Arguments

result

Returned with the result of subtracting *operand2* from *operand1*. (**a8**)

operand1

The operand to subtract from. (**a8**)

operand2

The operand to subtract. (**a8**)

Discussion

The FP_SUB subroutine subtracts one floating-point number from another.

The following equation is performed:

$$result = operand1 - operand2$$

Result, *operand1*, and *operand2* are native floating-point numbers represented as **a8** variables.

Examples

See the example on [page 12-4](#).

FP_TAN – Return the tangent of a floating-point number

```
xcall FP_TAN(result, operand)
```

Arguments

result

Returned with the tangent of *operand*. (**a8**)

operand

The radian value whose tangent will be returned. (**a8**)

Discussion

The FP_TAN subroutine returns the tangent of a floating-point number.

The following operation is performed:

$result = \tan(operand)$

Result and *operand* are native floating-point numbers represented as **a8** variables.

FP_TO_NUM – Convert from floating-point to implied-decimal

```
xcall FP_TO_NUM(float, number)
```

Arguments

float

A variable that contains a native floating-point number (C “double” data type) to convert to an implied-decimal number. (**a8**)

number

Returned with an implied-decimal number or alpha string. (**a23** or **n**)

Discussion

The FP_TO_NUM subroutine converts a floating-point number to an implied-decimal number.

If the second arguments to the FP_TO_NUM subroutine is alpha, the returned format is as follows:

X.YYYYYYYYYYYYYYYYY eNNN

Therefore, you can inspect all values of floating-point numbers. For very large and very small floating-point values, the implied-decimal value is returned as zero.



To avoid receiving invalid results when using the floating-point routines, do not use the %IMPLIED function with this call. %IMPLIED performs a transformation of the data and allocates temporary memory for the result. Therefore, using %IMPLIED causes temporary memory, rather than the memory location passed to %IMPLIED, to be modified.

Examples

See the example on [page 12-4](#).

13

Synergy HTTP Document Transport API

Introduction to the Synergy HTTP Document Transport API 13-2

Explains HTTP and HTTPS, provides instructions for using the Synergy HTTP document transport API from within your Synergy program, and discusses how to transmit data securely.

Synergy HTTP Document Transport API Routines 13-14

This API includes the following routines:

%HTTP_CLIENT_GET – Send a document retrieval request to a server	13-15
%HTTP_CLIENT_POST – Send a request to post a document to a server.....	13-19
%HTTP_METHOD – Specify a method that prompts for a password	13-23
%HTTP_SERVER_CREATE – Create an HTTP or HTTPS server class object.....	13-24
%HTTP_SERVER_DELETE – Free memory for the HTTP or HTTPS server	13-26
%HTTP_SERVER_RECEIVE – Receive a request from a client	13-27
%HTTP_SERVER_SEND – Send a response to a client.....	13-30

Introduction to the Synergy HTTP Document Transport API

The Synergy HTTP document transport API enables you to send and receive documents via HTTP or HTTPS from within your Synergy program.

What is HTTP?

HTTP, which stands for HyperText Transfer Protocol, is a communication protocol for sending information via sockets. The underlying protocol for the World Wide Web, HTTP defines how messages are formatted and transmitted and what actions Web servers and clients should take in response to commands.

What is HTTPS?

HTTPS, or secure HTTP, is an extension to HTTP that enables you to securely transmit sensitive data over the Web. It uses Netscape's Secure Sockets Layer (SSL) as a sublayer under its regular HTTP application layering to encrypt and decrypt user page requests as well as pages that are returned by the Web server. For example, if you do your banking online, the URL for the log-in page and any other account information you view will start with `https://`. If you submit a form to transfer funds, your browser's HTTPS sublayer will encrypt it, the server will acknowledge the request on the same connection with an equally encrypted reply, and then your browser's HTTPS sublayer will decrypt the acknowledgement for you.

HTTPS and SSL support the use of X.509 digital certificates from the server so that, if necessary, a user can authenticate the sender. A digital certificate is like an electronic ID card that establishes your credentials when doing business or other transactions on the Web. If you want to send encrypted messages, you can apply for a digital certificate from a certificate authority (CA). The CA digitally creates and issues a certificate that contains your name, a serial number, the expiration dates, a copy of the certificate holder's public key (used for encrypting messages and digital signatures), and the digital signature of the certificate authority so that a recipient can verify the certificate is real. The recipient of an encrypted message uses the CA's public key to decrypt the digital certificate attached to the message, verifies it as being issued by the CA, and gets the sender's public key and other information from the certificate, which enables the recipient to send an encrypted reply. X.509 is the most widely used standard for defining digital certificates.

Using the HTTP document transport API

Using the HTTP document transport API, a Synergy program, posing as an HTTP/HTTPS client, is able to send or receive XML data from an HTTP/HTTPS server. Conversely, a Synergy program can pose as an HTTP/HTTPS server to receive XML data from an HTTP/HTTPS client and then send XML data to that client to satisfy the client's request.

This API uses the TCP/IP network protocol.



You must include the **synxml.def** file, which defines some of the standard HTTP status codes and the text associated with these codes, to use the %HTTP_xxx routines.

All memory that you allocate for the body of an HTTP/HTTPS transmission must be allocated with the DM_STATIC option of the %MEM_PROC routine. This memory must be global so it can be manipulated at different levels within the program. A memory allocation error occurs if a memory handle is not allocated with DM_STATIC.

All memory handle and length arguments (*handle* and *length*) are required arguments to the method calls. The memory handles are resized to the needed length by the %HTTP_xxx methods, and the correct length is returned.

You will use the document URI (Uniform Resource Indicator) to set the host IP and port number of the server you will be calling. The URI consists of

- ▶ the protocol being used. Example: "http://".
- ▶ optional user information, in the form

username:password@

Example: "bsmith:iq497j@"



For security reasons, we do not recommend passing *username:password* as clear text in an HTTP packet.

- ▶ the host IP and/or the port number. At the minimum, a server name must be specified. If the port number is not specified for an HTTP connection, port 80 is assumed. If the port number is not specified for an HTTPS connection, the client will connect to port 443 of the host specified in the URI. Example: "http://localhost:3507".
- ▶ the name and location of the file to get or to post to.

If sending to a Synergy HTTP/HTTPS server, this can be a path or logical and filename.
Examples:

"http://myServer:3507/DBLDIR:afile.txt"

"http://myName:myPassword@myServer:3507/c:/location/afile.txt"

If sending to a web server (such as IIS or Apache), *location* is a virtual directory set up on the web server, plus the page you want. Examples:

“http://myServer/*location*/file.jsp”

“http://myServer/*location*/file.asp”

For the client, you must build the URI with this information to send to the server. For the server, you must parse the specified URI to get the filename and location.

A URI can have a maximum of 2000 bytes. URIs that are longer than 2000 bytes will be truncated.

The HTTP document transport API automatically escapes certain characters, such as spaces, in the URI when it sends a request to a server, and it also unescapes any escaped characters upon receiving a request from a client. (See the table on [page 9-325](#) for a list of characters that are escaped and their corresponding escape codes.) If you do not want this behavior to occur automatically, you can set the HTTP_NOESCAPE environment variable. (See [HTTP_NOESCAPE](#) in the “Environment Variables” chapter of *Environment Variables and System Options*.)



The Synergy OPEN statement does not allow escape characters in a file specification. Therefore, if your program contains logic that uses part of the URI as the filename to save the handle data to, you may encounter problems on OPEN if the URI contains escape characters.

The HTTP document transport API supports Synergy client HTTP and HTTPS requests through a proxy server. To specify the name and port of the proxy, use the PROXY_HOST and PROXY_PORT environment variables. (See [PROXY_HOST](#) and [PROXY_PORT](#) in the “Environment Variables” chapter of *Environment Variables and System Options*.) If PROXY_HOST is not defined, the HTTP document transport API will not go through a proxy server.

You can change other default proxy settings using the PROXY_LOCAL and PROXY_SUBNET environment variables. (See [PROXY_LOCAL](#) and [PROXY_SUBNET](#) in the “Environment Variables” chapter of *Environment Variables and System Options*.) If you’re having trouble connecting to local URIs, check the PROXY_SUBNET mask to ensure it matches your local subnet mask. To view your proxy settings in Internet Explorer, select Tools > Internet Options > Connections > LAN Settings.

Transmitting data securely

The Synergy HTTP document transport API interfaces with a third-party library, OpenSSL, to provide SSL support.

Setting up your HTTPS system

Follow these steps to set up your system to use encryption. For details on which version of OpenSSL is required for your operating system, see “[OpenSSL Requirements](#)” in the *Installation Configuration Guide*.

1. Install OpenSSL. See Synergex KnowledgeBase article [100001979](#) for the latest information on where to obtain OpenSSL for the various operating systems and details on installation.
 - ▶ For Windows, go to <http://www.openssl.org/related/binaries.html> and follow the link to download the version for your platform.
 - ▶ For UNIX, obtain the OpenSSL security kit specific to your operating system and version from your operating system vendor, if one is available. For some platforms, you may have to download and build the shared libraries. See Synergex KnowledgeBase article [100001970](#) for details.
 - ▶ For OpenVMS, use the operating system-supplied libraries, HP SSL, available from http://h71000.www7.hp.com/openvms/products/ssl/ssl_download.html.
2. Ensure that the OpenSSL shared libraries are in the correct location or have been added to the correct path.
 - ▶ On Windows, if OpenSSL is installed in the default location (the Windows system directory), you don't need to set anything. If you install to another location, set PATH to the location of the OpenSSL libraries.
 - ▶ On UNIX, whether you install the vendor-supplied OpenSSL libraries or build the shared libraries yourself, you will need to set the library path (LIBPATH on AIX, SHLIB_PATH on HP-UX, and LD_LIBRARY_PATH on all other systems).



If you build the OpenSSL libraries yourself, you will need to build the shared versions of the libraries.

- ▶ On OpenVMS, during installation, HTTPSLIB is set in SYS\$MANAGER:SYNERGY_STARTUP.COM to the full path and filename of **HTTSLIB.EXE**, the HTTPS runtime support shared image file.

You can use the **openssl** command line utility to create a certificate request and key file, append a key file to the created certificate, export certificates from Internet Explorer for use with OpenSSL as a CA file, establish a local CA for testing purposes, and more. Instructions for some of these tasks are provided in the sections that follow. Additional documentation for the **openssl** utility can be found in the **openssl** manpages and at <http://www.openssl.org/docs/>. We also highly

recommend the book *Network Security with OpenSSL* by John Viega, Matt Messier, and Pravir Chandra. If you are new to HTTPS, a book that provides a good high-level description of SSL, certificates, and encryption is *SSL and TLS Essentials: Securing the Web* by Stephen Thomas.

The following HTTPS runtime support DLL files are installed with your Synergy/DE™ distribution:

httpslib.dll (Windows)

httpslib.so (UNIX)

httpslib.exe (OpenVMS)



Some countries are legally restricted from using HTTPS support, and it is illegal to export strong encryption to those countries. Refer to <http://www.bis.doc.gov/Encryption/Default.htm> for current U.S. regulations. You can disable Synergy HTTPS support on a site-by-site basis by removing the HTTPS runtime support DLL.

Making a certificate request

If you want to use HTTPS as either a server or an authenticated client, you must first get a certificate from a trusted certificate authority so your server or client can be authenticated.

1. Use the **openssl** utility to generate a certificate request and create a key file:

```
openssl req -newkey rsa:bits [-keyout key_file] [-out output_file]
-config openssl.cnf
```

where *bits* is the size in bits of the new private key, *key_file* is the name of the file to write the newly created private key to, and *output_file* is the name of the file to write to. This command creates a new certificate request and a new private RSA key of the specified size. If the **-keyout** option is not specified, the filename specified in the configuration file is used as the key file. The **-config** option specifies that the **openssl.cnf** file, distributed with your HTTPS support files, will be used as the configuration file. The certificate request file that is created will contain the country, state, city, organization, department, and common name and e-mail address of the requester.

The example below creates the request file **testreq.pem**:

```
openssl req -newkey rsa:1024 -keyout testkey.pem -out testreq.pem
-config openssl.cnf
```

2. Send the certificate request file generated in step 1 to a public certificate authority, such as VeriSign or Thawte, along with any necessary fees.

The certificate authority will send back a certificate that can be used to authenticate a server or client.

Getting a document off an HTTPS server and downloading it to the client

Before you begin, you must know the URI of the resource to be requested, and you must have configured the client sites with the proper client certificates, if necessary. Your program should do the following:

1. If a passphrase is going to be required to access a PEM file, call `%HTTP_METHOD` (see [page 13-23](#)) on the client so that a Synergy method will be called to prompt for the password.
2. Using `%HTTP_CLIENT_GET` (see [page 13-15](#)), send an HTTPS GET request providing the URL of the page to get, a trusted CA filename, and optionally the client's supported SSL protocols and ciphers and the name of a client certificate file.

The client will authenticate the server certificate, and an SSL connection will be established with the HTTPS server. If the supported protocols and ciphers are not specified, all protocols and ciphers will be accepted. The name of a client certificate is only required when the client is requesting a resource on the server for which the server requires authentication.

3. After a response is received from the HTTPS server, check the response status by checking for a nonzero return code from `%HTTP_CLIENT_GET`.

The response body should contain the requested data. The SSL connection will be closed.

If you want to post a document to an HTTPS server, use `%HTTP_CLIENT_POST` (see [page 13-19](#)) instead of `%HTTP_CLIENT_GET`, and provide the URL of the page to post and the data to be posted as the body of the request.

Making your program accept and respond to an HTTPS request

Your program should do the following:

1. If a passphrase is going to be required to access a PEM file, call `%HTTP_METHOD` (see [page 13-23](#)) on the server so that a Synergy method will be called to prompt for the password.
2. Using `%HTTP_SERVER_CREATE` (see [page 13-24](#)), create a server and initialize Synergy HTTPS by providing the name of a certificate file and optionally the server's supported SSL protocol, supported ciphers, and the names of the trusted CAs.

If no protocols or ciphers are specified, all protocols and ciphers will be accepted. If trusted CAs are specified, client authentication will be performed on all requests.

The Synergy program running as an HTTPS server initializes by connecting to the local host and a known port.

3. Wait to accept an HTTPS GET or POST request. Use `%HTTP_SERVER_RECEIVE` (see [page 13-27](#)) to receive the request and inspect it to find out the resource requested. Process the request and retrieve data as needed.

Your Synergy program initializes the HTTPS response by setting the status value and putting the response data in the body of the HTTPS response. (It puts the response text in the handle and returns the handle and its length as argument values.)

4. Use `%HTTP_SERVER_SEND` (see [page 13-30](#)) to send the HTTPS response to the client.
5. Wait to accept another HTTPS request.
6. When you're finished, use `%HTTP_SERVER_DELETE` (see [page 13-26](#)) to release the memory associated with the server object.

Appending a key file to a PEM certificate

A PEM certificate file may contain both the certificate and private key. If it doesn't contain the private key, the Synergy HTTP document transport API will load a PEM-based key file whose name is derived by appending "key.pem" to the end of the certificate filename. For example, for a certificate file named **test.der** or **test.pem**, the API will load a key file named **testkey.pem**. This file can then be used with the API.

Importing and exporting certificates in Internet Explorer

Internet Explorer acts as a client and keeps its own list of trusted Certificate Authorities. For convenience, you can use this list as a CA file in the **openssl** utility or Synergy HTTP document transport API routines.

Importing certificates

If you want to add trusted certificates to Internet Explorer's list, you can import certificates as follows:

1. From Internet Explorer, select Tools > Internet Options.
2. Select the Content tab and click the Certificates button.
3. Select the Trusted Root Certification Authorities tab.
4. Select all the certificates in this dialog box and click the Import button.
5. Follow the Certificate Manager Import Wizard and select the PEM file you want to include as a trusted CA.

Exporting certificates

To export certificates for use with OpenSSL or the Synergy HTTP document transport API,

1. From Internet Explorer, select Tools > Internet Options.
2. Select the Content tab and click the Certificates button.
3. Select the Trusted Root Certification Authorities tab.
4. Select all of the certificates in this dialog box and click Export button.

5. Follow the Certificate Manager Export Wizard to create a **.p7b** file.
6. Use the **openssl** command line utility to convert the file to PEM format:

```
openssl pkcs7 -inform DER -outform PEM -in file.p7b -out file.pem
-print_certs
```

The **file.pem** file can now be specified in the **openssl** utility or as the **CA_file** argument in the Synergy HTTP document transport API.

Testing your HTTPS setup locally

If you want to test your HTTPS setup locally, without going through a public certificate authority, you can establish a local certificate authority. (In the actual production world, of course, you would use a public CA, such as VeriSign or Thawte.)

1. Use the **openssl** utility to create a local CA. Here's an example:

```
openssl req -newkey rsa -out rootreq.pem -keyout rootkey.pem
-config caopenssl.cnf

openssl x509 -req -in rootreq.pem -signkey rootkey.pem -out rootcert.pem
-extfile caopenssl.cnf

cat rootcert.pem rootkey.pem > root.pem
```

The **req** command, using the **-out** and **-newkey** options, creates a certificate request and its corresponding key. **Rootreq.pem** is the file to write the new certificate request to, **rootkey.pem** is the file to write the newly created private key to, and **caopenssl.cnf** is a configuration file containing certificate extensions to use.

The **x509** command is a certificate utility that can be used to display certificate information, convert certificates to various forms, sign certificate requests like a “mini CA,” or edit certificate trust settings. The **-req** option creates a certificate in the **rootcert.pem** file by reading the **rootreq.pem** file. The **-signkey** option causes **rootreq.pem** to be self-signed using the private key **rootkey.pem**, which was generated by the **openssl req** command. **Caopenssl.cnf** is a configuration file containing certificate extensions to use. If **-extfile** is not specified, no extensions are added to the certificate.

The **cat** command is optional, but it simplifies things for testing purposes. It concatenates the certificate file (**rootcert.pem**) and the key file (**rootkey.pem**) to yield a concatenated file called **root.pem**.

2. If you want to create a certificate and sign it using your local CA, use the **openssl ca** command or the **openssl x509** command. For example:

```
openssl ca -in testreq.pem -out testcert.pem -config caopenssl.cnf
```

or

```
openssl x509 -req -in testreq.pem -extfile caopenssl.cnf -CA root.pem  
-CAkey root.pem -CAcreateserial -out testcert.pem
```

```
cat testcert.pem testkey.pem > test.pem
```

where **testreq.pem** is a file that contains a single certificate request to be signed by the local CA, **testcert.pem** is the file to return signed certificates to, and **caopenssl.cnf** is a configuration file containing certificate extensions to use. (**Testreq.pem** and **testcert.pem** were created using the same syntax as the **openssl** commands shown in step 1.) When the **-CA** option is present, the **x509** command behaves like a mini CA. In this example, **root.pem** is the CA certificate to be used for signing and the CA private key to sign a certificate with. If the **-CAkey** option is not specified, it is assumed that the CA private key is present in the CA certificate file. The **-CAcreateserial** option creates the CA serial number file if it does not exist. **Testkey.pem** is the key file that was created when the original request was made with the **req -newkey** command, and **test.pem** is the concatenated certificate file and key file. Again, concatenating is optional, but it simplifies things for testing.

3. If you want to remove the passphrase from an RSA private key, use the **openssl rsa** command. For example:

```
openssl rsa -in testkey.pem -out testkeyout.pem
```

where **testkey.pem** is the file to read a key from and **testkeyout.pem** is the file to write a key to. If any encryption options are set, a passphrase will be prompted for. The output filename should not be the same as the input filename.



Note that [step 3](#) is a security risk on a public certificate, but it is convenient for local testing.

After you remove the passphrase, the certificate and key files must be reconcatenated.

Ciphers

Several of the HTTP document transport API routines pass an optional *ciphers* argument, which specifies the encryption methods that are in effect in the form of a “cipher list.” A cipher list consists of one or more cipher strings separated by colons. Commas or spaces are also acceptable separators, but colons are normally used. The actual cipher string can take several different forms:

- ▶ It can consist of a single cipher suite, such as RC4-SHA.
- ▶ It can represent a list of cipher suites containing a certain algorithm or of a certain type. For example, SHA1 represents all cipher suites using the digest algorithm SHA1, and SSLv3 represents all SSL v3 algorithms.

- It can consist of one or more lists of cipher suites combined in a single cipher string using the + character. This is used as a logical “and” operation. For example, SHA1+DES represents all cipher suites containing the SHA1 and the DES algorithms.

Each cipher string can be optionally preceded by the following characters:

Character	Meaning
!	The specified ciphers are permanently deleted from the list. The deleted ciphers can never reappear in the list even if they are explicitly stated.
-	The specified ciphers are deleted from the list, but some or all of the ciphers can be added again by later options.
+	The specified ciphers are moved to the end of the list. This option doesn't add any new ciphers; it just moves matching existing ones.
None	The string is simply interpreted as a list of ciphers to be appended to the current preference list. If the list includes any ciphers already present, those ciphers will be ignored (i.e., they will not be moved to the end of the list).

Additionally the cipher string @STRENGTH can be used at any point to sort the current cipher list in order of encryption algorithm key length. When @STRENGTH is specified, HTTPS attempts to select the strongest cipher from the list.

For a current list of permitted cipher strings and their meanings, as well as lists of cipher suite names and their OpenSSL equivalents, see <http://www.openssl.org/docs/apps/ciphers.html#>.

Error messages

The following errors can occur when using the HTTP document transport API:

Message	Cause
Address in use	A socket error occurred when the server attempted to start on a port that was already being used.
Address not available	A socket error occurred. The requested address is not valid in this context.
CA file required	An HTTPS URI was passed to %HTTP_CLIENT_GET or %HTTP_CLIENT_POST, but the CA filename was not passed. A CA filename is required for an HTTPS request.
Cannot get certificate from <i>file</i>	The certificate file does not include the certificate section.

Synergy HTTP Document Transport API

Introduction to the Synergy HTTP Document Transport API

Message	Cause
Cannot get private key from <i>file</i>	The certificate file does not include the private key section.
Cannot load random state	There is not enough random data to seed cryptographic algorithms. Define the HTTP_RANDOM environment variable to a file that the Synergy HTTP document transport API can use to gather random data. (See HTTP_RANDOM in the “Environment Variables” chapter of <i>Environment Variables and System Options</i> for more information.)
Certificate verify failed	The certificate could not be validated using trusted CAs.
Connection refused	A socket error occurred when the client tried to connect to an invalid host and port.
Connection reset	A socket error occurred. If you are using HTTPS functionality, this could mean that only the client or only the server supports SSLv2 or that you tried to send an HTTP request to an HTTPS server. If you are using HTTP, the connection was closed by the remote host.
Error setting keepalive	A socket error occurred when the keepalive value was being set.
Error setting linger	A socket error occurred when the linger value was being set.
Host not found	A socket error occurred when an invalid host name was specified while the client was trying to connect to the server.
http request	The server expected an HTTPS request but received an HTTP request from the client.
HTTPS not available	HTTPS support is not available on the current platform. Either the httpslib DLL or the OpenSSL library is missing, or the OpenSSL library is not the shared version.
Invalid http request	The server did not receive a well-formed HTTP request. The client may have tried to send an HTTPS request instead.
Invalid user information in URI	An invalid character was specified in the user information portion of the URI in a call to %HTTP_CLIENT_GET or %HTTP_CLIENT_POST. Valid characters are any alphanumeric character, escape codes %00 to %FF, semicolon (;), colon (:), ampersand (&), equal sign (=), plus sign (+), dollar sign (\$), comma (,), hyphen (-), underscore (_), period (.), exclamation point (!), tilde (~), asterisk (*), single quotation mark ('), left parenthesis ((), or right parenthesis ()).
No certificate returned	The peer did not have a certificate.

Message	Cause
No ciphers available	Either the ciphers on the client and server do not match or the cipher specified on the client is invalid.
No shared ciphers	The client and server do not support the same ciphers.
No such file or directory	The specified CA file or certificate filename does not exist.
Socket error <i>number</i>	The socket error <i>number</i> occurred. Socket error numbers are usually defined in Microsoft Developer Network (MSDN) help for Microsoft platforms and in the manpages or help pages of the particular socket function that failed for other platforms.
Socket is not connected	A socket error occurred. An attempt to send or receive data failed because the socket was not connected.
SSL handshake failure	This message usually appears on the client when there is a cipher mismatch or a certificate authorization issue. The most common reasons are as follows: ▶ The server does not support the client's SSL protocol version. Ensure the client is using a protocol version supported by the server. ▶ The server and client do not share common cryptographic algorithms. Ensure the client is using a cipher supported by the server. ▶ Server authentication failed. The server must have a valid certificate that can be traced to a CA that the client trusts. ▶ Client authentication failed. The client must have a valid certificate that can be traced to a CA that the server trusts.
Timed out	A socket error occurred when the program timed out while waiting to accept or receive a request.
Unsupported protocol	The client and server protocols do not match.

Sample programs

A sample HTTP program is available in the CodeExchange at <http://www.synergex.com>.

Synergy HTTP Document Transport API Routines

The HTTP document transport API routines begin with `%HTTP_` and are documented on the pages that follow.

%HTTP_CLIENT_GET – Send a document retrieval request to a server

```
status = %HTTP_CLIENT_GET (uri, [timeout], handle, length, error, [headers], [log_file],
&                                     [protocols], [ciphers], [cert_file], [CA_file], [HTTP_RELURI]
&                                     [, version])
```

Return value

status

One of the following values:

0 Successful.

error_number The number of the error that occurred. (Not successful.)

Arguments

uri

The absolute URI (Uniform Resource Indicator) of the document to retrieve. **(a)**

timeout

(optional) The number of seconds to wait from the time a request is sent (after the connection to the server) until a response is received from the server. The default is 0, which specifies wait forever. **(n)**

handle

The static memory handle containing the retrieved document. **(n)**

length

The length of the document to retrieve. **(n)**

error

Returned with an error message if the return value is nonzero. **(a)**

headers

(optional) An array of one or more HTTP header strings. Each string has the form *header_name*[: *value*]. **([*]a)**

log_file

(optional) The name and location of a log file to log HTTP packet contents. **(a)**

protocols

(optional) One or more SSL protocol versions in effect, combined with a plus sign (for example, `SSLVER_SSL2+SSLVER_SSL3`): **(n)**

SSLVER_SSL2 SSL version 2

SSLVER_SSL3 SSL version 3

SSLVER_TLS1 TLS version 1

SSLVER_ALL All protocols are available. (default)

ciphers

(optional) A cipher list that specifies the encryption methods in effect. **(a)**

cert_file

(optional) The name of the client certificate file. **(a)**

CA_file

(optional) The name of a file containing certificate authorities that the client trusts. **(a)**

HTTP_RELURI

(optional) The absolute URI is translated and sent as a relative path in the HTTP request. If `HTTP_RELURI` is not passed, the absolute URI is sent (with the path exactly as specified).

version

(optional) A string that contains the HTTP version number to be placed in the document header. The default version number is 1.0. **(a)**

Discussion

`%HTTP_CLIENT_GET` sends a request to an HTTP or HTTPS server to retrieve a document. The URI determines what type of connection will be made to the server. If you pass an HTTP URI to `%HTTP_CLIENT_GET`, the client establishes an HTTP connection. If, on the other hand, you pass an HTTPS URI, the client establishes an SSL connection to an HTTPS server. The document is returned to the client from the HTTP or HTTPS server in the form of a memory handle. See [“Using the HTTP document transport API” on page 13-3](#) for details about the *uri* syntax.

Unless the `HTTP_NOESCAPE` environment variable is set, `%HTTP_CLIENT_GET` automatically escapes, before the HTTP request is made, certain characters in the URI. (See the table on [page 9-325](#) for a list of characters that are escaped and their corresponding escape codes.) The escaped URI will appear in the HTTP packet log. For example, if you made the following call:

```
ret=%http_client_get("http://server/hello there.html",10,h,sz,err,,log)
```

you would see this in the log file:

```
GET http://server/hello%20there.html HTTP/1.0
```


To disable automatic escaping, set `HTTP_NOESCAPE`. (See [HTTP_NOESCAPE](#) in the “Environment Variables” chapter of *Environment Variables and System Options*.)

If *handle* and *length* have a value of 0, `%HTTP_CLIENT_GET` allocates a memory handle for you. If, instead of 0, you pass a valid memory handle, that handle is used, but `%HTTP_CLIENT_GET` resizes the handle to the size of the retrieved document. It’s your responsibility to free any memory allocated by this routine.

If an error occurs, an error message is returned in the *error* field. Otherwise, *error* is blank.

For most applications, the standard default HTTP headers will work as provided. However, some HTTP servers require extra header values to process the HTTP message properly. The *headers* argument therefore enables you to add or modify headers for client GET requests. If you pass in a header name that was previously set, the last header value passed for that name will be the one sent in the HTTP message. An individual HTTP header can have a maximum of 499 bytes. Headers longer than 499 bytes will be truncated.

Passing a *log_file* causes both the HTTP/HTTPS request packet and the HTTP/HTTPS response packet to be logged to the specified file. This logging feature is available primarily for debugging purposes and should not normally be turned on in a secure production environment. Logs of a request packet include the HTTP method (in this case, GET), the URI, the HTTP version, all HTTP headers, and the body of the HTTP packet. Logs of a response packet include the HTTP version, status, reason, all HTTP headers, and the body of the HTTP packet. Log entries are always appended to an existing log file. If the log file does not exist, it is created. A log file can only be created on a local drive; remote drives are not supported.

If *ciphers* is not specified, the ciphers designated by the cipher list format value `DEFAULT` are in effect. See “[Ciphers](#)” on page 13-10 for more detailed information and a list of supported ciphers.

If specified, *cert_file* and *CA_file* must be the names of existing files in PEM or DER format. A PEM certificate file may contain both the certificate and private key. If it doesn’t contain the private key, Synergy HTTPS will load a PEM-based key file whose name is derived by appending “key.pem” to the end of the certificate filename. For example, for a certificate file named **test.der** or **test.pem**, Synergy HTTPS will load a key file named **testkey.pem**. If an authentication error occurs, the error is returned and the connection is terminated.



If you pass an HTTPS URI but don’t specify a *CA_file*, server verification will be turned off. Because this poses a security risk, we recommend that you not do it unless you only need encryption without verification.

Examples

```
headerArray[1] = "Content-Type:text/xml"
clear timeout, handle, length, errormsg
URI = "http://www.synergex.com/ie_home.htm"
logfile = "c:\clt.log"
rtn = %http_client_get(uri, timeout, handle, length, errormsg,
    &                                headerArray, logfile)
```

Also see the CodeExchange at <http://www.synergex.com>.

%HTTP_CLIENT_POST – Send a request to post a document to a server

```
status = %HTTP_CLIENT_POST (uri, [timeout], send_handle, send_length, receive_handle,
&
    receive_length, error, [headers], [log_file], [protocols],
&
    [ciphers], [cert_file], [CA_file], [HTTP_RELURI][, version])
```

Return value

status

One of the following values:

0 Successful.

error_number The number of the error that occurred. (Not successful.)

Arguments

uri

The absolute URI (Uniform Resource Indicator) of the document to send to the HTTP server. **(a)**

timeout

(optional) The number of seconds to wait from the time a request is sent (after the connection to the server) until a response is received from the server. The default is 0, which specifies wait forever. **(n)**

send_handle

The static memory handle of the document to send to the HTTP server. **(n)**

send_length

The length of the document to send to the HTTP server. **(n)**

receive_handle

The static memory handle of the document that will be received in response to the post request. **(n)**

receive_length

The length of the document that will be received in response to the post request. **(n)**

error

Returned with an error message if the return value is nonzero. **(a)**

headers

(optional) An array of one or more HTTP header strings. Each string has the form *header_name*[: *value*]. ([*]**a**)

log_file

(optional) The name and location of a log file to log HTTP packet contents. (**a**)

protocols

(optional) One or more SSL protocol versions in effect, combined with a plus sign (for example, SSLVER_SSL2+SSLVER_SSL3): (**n**)

SSLVER_SSL2 SSL version 2

SSLVER_SSL3 SSL version 3

SSLVER_TLS1 TLS version 1

SSLVER_ALL All protocols are available. (default)

ciphers

(optional) A cipher list that specifies the encryption methods in effect. (**a**)

cert_file

(optional) The name of the client certificate file. (**a**)

CA_file

(optional) the name of a file containing certificate authorities that the client trusts. (**a**)

HTTP_RELURI

(optional) The absolute URI is translated and sent as a relative path in the HTTP request. If HTTP_RELURI is not passed, the absolute URI is sent (with the path exactly as specified).

version

(optional) A string that contains the HTTP version number to be placed in the document header. The default version number is 1.0. (**a**)

Discussion

%HTTP_CLIENT_POST sends a “post” request to an HTTP or HTTPS server, which enables the client to send the specified document to the server. The URI determines what type of connection will be made to the server. If you pass an HTTP URI to %HTTP_CLIENT_POST, the client establishes an HTTP connection. If, on the other hand, you pass an HTTPS URI, the client establishes an SSL connection to an HTTPS server. See [“Using the HTTP document transport API” on page 13-3](#) for details about the *uri* syntax.

Unless the `HTTP_NOESCAPE` environment variable is set, `%HTTP_CLIENT_POST` automatically escapes, before the HTTP request is made, certain characters in the URI. (See the table on [page 9-325](#) for a list of characters that are escaped and their corresponding escape codes.) The escaped URI will appear in the HTTP packet log.

For example, if you made the following call:

```
ret=%http_client_post("http://server/hello there.html",10,sh,ssz,rh,rsz,
    &                                err,,log)
```

you would see this in the log file:

```
POST http://server/hello%20there.html HTTP/1.0
```

To disable automatic escaping, set `HTTP_NOESCAPE`. (See [HTTP_NOESCAPE](#) in the “Environment Variables” chapter of *Environment Variables and System Options* for details.)

If `receive_handle` and `receive_length` have a value of 0, `%HTTP_CLIENT_POST` allocates a memory handle for you. If, instead, you pass a valid memory handle (instead of 0), that handle is used, but `%HTTP_CLIENT_POST` resizes the handle to the correct size of the retrieved document. It’s your responsibility to free any memory allocated by this routine.

If an error occurs, an error message is returned in the `error` field. Otherwise, `error` is blank.

For most applications, the standard default HTTP headers will work as provided. However, some HTTP servers require extra header values to process the HTTP message properly. The `headers` argument therefore enables you to add or modify headers for client POST requests. If you pass in a header name that was previously set, the last header value passed for that name will be the one sent in the HTTP message. An individual HTTP header can have a maximum of 499 bytes. Headers longer than 499 bytes will be truncated.

Passing a `log_file` causes both the HTTP/HTTPS request packet and the HTTP/HTTPS response packet to be logged to the specified file. This logging feature is available primarily for debugging purposes and should not normally be turned on in a secure production environment. Logs of a request packet include the HTTP method (in this case, POST), the URI, the HTTP version, all HTTP headers, and the body of the HTTP packet. Logs of a response packet include the HTTP version, status, reason, all HTTP headers, and the body of the HTTP packet. Log entries are always appended to an existing log file. If the log file does not exist, it is created. A log file can only be created on a local drive; remote drives are not supported.

If `ciphers` is not specified, the ciphers designated by the cipher list format value `DEFAULT` are in effect. See “[Ciphers](#)” on [page 13-10](#) for more detailed information and a list of supported ciphers.

If specified, `cert_file` and `CA_file` must be the names of existing files in PEM or DER format. A PEM certificate file may contain both the certificate and private key. If it doesn’t contain the private key, Synergy HTTPS will load a PEM-based key file whose name is derived by appending

“key.pem” to the end of the certificate filename. For example, for a certificate file named **test.der** or **test.pem**, Synergy HTTPS will load a key file named **testkey.pem**. If an authentication error occurs, the error is returned and the connection is terminated.



If you pass an HTTPS URI but don't specify a *CA_file*, server verification will be turned off. Because this poses a security risk, we recommend that you not do it unless you only need encryption without verification.

Examples

```
clear timeout, errormsg
URI = "http://hostname/filename.ext"
logfile = "c:\post.log"
headerArray[1] = "Content-Type:text/xml"
.
.
.
; Declare and fill the send memory handle and length with the document
; to post to the HTTP server
rtn = %http_client_post(URI, timeout, send_hndl, send_len, rec_hndl,
    &                                rec_len, errormsg, headerArray, logfile)
.
.
.
; Test rtn and rec_hndl, rec_len for a valid response document
```

Also see the CodeExchange at <http://www.synergex.com>.

%HTTP_METHOD – Specify a method that prompts for a password

status = %HTTP_METHOD (*callback_type*, *method*, *error_msg*)

Return value

status

One of the following values:

0 Successful.

error_number The number of the error that occurred. (Not successful.)

Arguments

callback_type

The callback type, which currently must be CALLBACK_PWD. (**n**)

method

The name of the Synergy callback routine to call. (**a**)

error_msg

Returned with descriptive text of the error. (**a**)

Discussion

%HTTP_METHOD specifies the name of a user-defined Synergy method that prompts the user for a Privacy Enhanced Mail (PEM) password. This method must have one Synergy alpha argument that returns the password to use. For example, the method may look something like this:

```
subroutine mypasswd
  pwd      , a
proc
  pwd = "secret"
  xreturn
endsubroutine
```

The %HTTP_METHOD function is defined as ^VAL.

%HTTP_SERVER_CREATE – Create an HTTP or HTTPS server class object

```
status = %HTTP_SERVER_CREATE (instance_id, port, error, [protocols], [ciphers], [cert_file]
&                                     [, CA_file])
```

Return value

status

One of the following values:

0 Successful.

error_number The number of the error that occurred. (Not successful.)

Arguments

instance_id

Returned with the static memory handle of this instance of a Synergy HTTP server, which will be used as the *instance_id* by all other calls for the HTTP server. **(n)**

port

The number of the port on which the server will listen. **(n)**

error

Returned with an error message if the return value is nonzero. **(a)**

protocols

(optional) One or more SSL protocol versions in effect, combined with a plus sign (for example, SSLVER_SSL2+SSLVER_SSL3): **(n)**

SSLVER_SSL2 SSL version 2

SSLVER_SSL3 SSL version 3

SSLVER_TLS1 TLS version 1

SSLVER_ALL All protocols are available. (default)

ciphers

(optional) A cipher list that specifies the encryption methods in effect. **(a)**

cert_file

(optional) The name of the server certificate file. **(a)**

CA_file

(optional) the name of a file containing certificate authorities that the server trusts. **(a)**

Discussion

%HTTP_SERVER_CREATE creates a socket to listen for requests on the specified *port*. The *instance_id* is a returned memory handle that is used by the other %HTTP_SERVER_XXX calls.

You must call %HTTP_SERVER_CREATE before calling any of the other server methods. Use %HTTP_SERVER_DELETE to free the memory allocated by this call.

If an error occurs, an error message is returned in the *error* field. Otherwise, *error* is blank.

If *ciphers* is not specified, the ciphers designated by the cipher list format value DEFAULT will be in effect. See “Ciphers” on page 13-10 for more detailed information and a list of supported ciphers.

Client authentication is optional depending on the information requested from the server. To create an HTTPS server that only accepts HTTPS connections, pass the *cert_file* argument. If *cert_file* is not specified, %HTTP_SERVER_CREATE creates an HTTP server. If *CA_file* is specified, client authentication will occur when a request is received by %HTTP_SERVER_RECEIVE.

If specified, *cert_file* and *CA_file* must be the name of existing files in PEM or DER format. A PEM certificate file may contain both the certificate and private key. If it doesn't contain the private key, Synergy HTTPS will load a PEM-based key file whose name is derived by appending “key.pem” to the end of the certificate filename. For example, for a certificate file named **test.der** or **test.pem**, Synergy HTTPS will load a key file named **testkey.pem**.

Examples

```
UsePort = 80
rtnvlu = %http_server_create(httpsvr, UsePort, errorMsg)

do
  begin
    rtnvlu = %http_server_receive(httpsvr, URI, method, timeout,
    &                                request_handle, request_length,
    &                                errorMsg, headerArray, logfile)
    .
    .
    .
    rtnvlu = %http_server_send(httpsvr, status, reason, response_handle
    &                                response_length, errorMsg, headerArray,
    &                                logfile)
    .
    .
    .
  end
until(ExitCondition)
rtnvlu = %http_server_delete(httpsvr)
.
.
.
```

Also see the CodeExchange at <http://www.synergex.com>.

%HTTP_SERVER_DELETE – Free memory for the HTTP or HTTPS server

status = %HTTP_SERVER_DELETE (*instance_id*)

Return value

status

Always returns **0**.

Arguments

instance_id

The ID of the created instance of a Synergy HTTP or HTTPS server. (**n**)

Discussion

%HTTP_SERVER_DELETE deletes the HTTP or HTTPS server object, releases all memory associated with it, and closes the socket on which the server is listening.

The *instance_id* is returned by %HTTP_SERVER_CREATE.

Examples

See Examples on [page 13-25](#).

Also see the CodeExchange at <http://www.synergex.com>.

%HTTP_SERVER_RECEIVE – Receive a request from a client

```
status = %HTTP_SERVER_RECEIVE(instance_id, uri, method, [timeout], handle, length, error,  

&                                     [headers], [log_file][, fork])
```

Return value

status

One of the following values:

0 Successful.

error_number The number of the error that occurred. (Not successful.)

Arguments

instance_id

The ID of the created instance of a Synergy HTTP or HTTPS server. (**n**)

uri

Returned with the URI (Uniform Resource Indicator) of the requested document. (**a**)

method

Returned with the HTTP method from the client (for example, GET, POST, or HEAD). (**a**)

timeout

(optional) The number of seconds to wait for a request from the client before an idle TCP/IP socket time-out occurs. The default is 0, which specifies wait forever. (**n**)

handle

Returned with the static memory handle of the requested document. (**n**)

length

Returned with the length of the requested document. (**n**)

error

Returned with an error message if the return value is nonzero. (**a**)

headers

(optional) Returned with an array of one or more HTTP header strings. Each string has the form *header_name*[: *value*]. (**[*]a**)

log_file

(optional) The name and location of a log file to log HTTP packet contents. (**a**)

UNIX

fork

(optional) One of the following values:

- 0** No child is created.
- 1** Create a child. The parent continues to accept new requests until the timeout period expires.
- 2** Create a child. The parent continues to accept new requests and never returns, regardless of any timeout value.

See the Discussion for additional information.

Discussion

%HTTP_SERVER_RECEIVE waits for a request from an HTTP or HTTPS client and returns the URI, method, handle, and length of the document requested. If the call fails, an error number is returned, and an error message is returned in the *error* argument. Otherwise, *error* is blank.

Unless the HTTP_NOESCAPE environment variable is set, %HTTP_SERVER_RECEIVE automatically unescapes all of the escaped characters present in the URI. (See the table on [page 9-325](#) for a list of escape codes that will be affected.) The escape codes received in the URI will appear in the HTTP packet log. For example, if the server received the following packet:

```
GET http://server/hello%20there.html HTTP/1.0
```

%HTTP_SERVER_RECEIVE would unescape the URI to yield

```
http://server/hello there.html
```

To disable automatic unescaping, set HTTP_NOESCAPE. (See [HTTP_NOESCAPE](#) in the “Environment Variables” chapter of *Environment Variables and System Options* for details.)

Call %HTTP_SERVER_RECEIVE in a loop with the %HTTP_SERVER_SEND method to retrieve and handle client requests. Then format the responses and send them back to the client.

The *instance_id* is returned by %HTTP_SERVER_CREATE.

Method can be any valid HTTP method. The two most commonly used methods are GET (which requests a document from a server) and POST (which sends a document to a server).

If *handle* and *length* have a value of 0, %HTTP_SERVER_RECEIVE allocates a memory handle for you. It's your responsibility to free any memory allocated by this routine.

The *headers* argument enables a Synergy HTTP or HTTPS server to receive HTTP header values. Make sure the array of strings you create is large enough to accept all possible headers. If it's too small, any received headers that don't fit will not be put in the array.

Passing a *log_file* causes the HTTP request packet to be logged to the specified file. This logging feature is available primarily for debugging purposes. Logs of a request packet include the HTTP method (in this case, RECEIVE), the URI, the HTTP version, all HTTP headers, and the body of the HTTP packet. Log entries are always appended to an existing log file. If the log file does not exist, it is created. A log file can only be created on a local drive; remote drives are not supported.

UNIX

You should not use *fork* if any SQL Connection remote sessions are open or if you have any remote *xfServer* files.

Using the *timeout* parameter with a *fork* value of 1 enables you to shut down the accept capability if necessary. If *fork* is passed with a value of 1, the parent will go back to accept the next connection request. When the child finishes the request, it should stop. When %HTTP_SERVER_RECEIVE returns, *fork* is returned with the value of the UNIX **fork(2)** routine that was used to create the child process. If the returned *fork* value is 0, the child process is processing the request. If the returned *fork* value is 1, the accept returned and the parent is running, taking the *timeout* parameter into account.

If *fork* is passed with a value of 2, the main accept loop will never time out but will still pass the *timeout* value to the rest of the HTTP operations. If the returned *fork* value is 0, the child process is processing the request. Otherwise, the routine never returns (except from an error).

Regardless of the *fork* value that is passed, a return *fork* value of -1 means a system error has occurred. You can use %SYSERR to capture the error and then store it in a local variable for later processing. (See %SYSERR on page 9-338 for more information.)

%HTTP_SERVER_RECEIVE creates an identical copy of the process, except all channels opened by the original have been closed in the child. Therefore, you must reopen any files needed by the child.



If the Synergy server quits, all of the child processes will quit as well.

For an HTTPS server, client authentication occurs when a request is received by %HTTP_SERVER_RECEIVE if a CA file was passed in the call to %HTTP_SERVER_CREATE. If an authentication error occurs, %HTTP_SERVER_RECEIVE returns the error and terminates the connection.

Examples

See Examples on page 13-25.

Also see the CodeExchange at <http://www.synergex.com>.

%HTTP_SERVER_SEND – Send a response to a client

```
status = %HTTP_SERVER_SEND (instance_id, status, reason, handle, length, error, [headers],
&                                [log_file], [version][, dont_close])
```

Return value

status

One of the following values:

0 Successful.

error_number The number of the error that occurred. (Not successful.)

Arguments

instance_id

The ID of the created instance of a Synergy HTTP or HTTPS server. **(n)**

status

The HTTP status code for the response message. **(n)**

reason

The HTTP reason code associated with the status code. **(a)**

handle

The static memory handle of the document. **(n)**

length

The length of the document. **(n)**

error

Returned with an error message if the return value is nonzero. **(a)**

headers

(optional) An array of one or more HTTP header strings. Each string has the form *header_name*[: *value*]. **([*]a)**

log_file

(optional) The name and location of a log file to log HTTP packet contents. **(a)**

version

(optional) A string that contains the HTTP version number to be placed in the document header. The default version number is 1.0. **(a)**

dont_close

(optional) If passed and nonzero, do not close the socket when %HTTP_SERVER_SEND completes successfully. (**n**)

Discussion

Every time a client makes a request, a response is required. The %HTTP_SERVER_SEND method sends a response back to the client.

The *instance_id* is returned by %HTTP_SERVER_CREATE.

If the *handle* and *length* are zero, the response HTTP message will not contain a body.

Status and *reason* must be of valid HTTP types. Standard status codes are numbered as follows:

100 - 199	Informational codes
200 - 299	Successful codes
300 - 399	Redirection codes
400 - 499	Client error codes
500 - 599	Server error codes

For your convenience, the most common status and reason codes are defined in the **synxml.def** file.

If you send valid *handle* and *length* arguments, you are responsible for freeing any memory allocated by this routine.

If an error occurs, an error message is returned in the *error* field. Otherwise, *error* is blank.

The *headers* argument enables a Synergy HTTP or HTTPS server to add or modify HTTP headers in its response message.

Passing a *log_file* causes the HTTP response packet to be logged to the specified file. This logging feature is available primarily for debugging purposes. Logs of a response packet include the HTTP version, status, reason, all HTTP headers, and the body of the HTTP packet. Log entries are always appended to an existing log file. If the log file does not exist, it is created. A log file can only be created on a local drive; remote drives are not supported.

Examples

See Examples on [page 13-25](#).

Also see the CodeExchange at <http://www.synergex.com>.

14

Synergy Symbol Table API

Introduction to the Synergy Symbol Table API Routines 14-2

Describes the requirements for using the Synergy symbol table API, explains access lists and search lists, and provides a sample program.

Synergy Symbol Table API Routines 14-7

This API includes the following routines:

%NSPC_ADD – Add an entry to a symbol table	14-8
%NSPC_ATOS – Map entry in symbol table to search index	14-9
NSPC_CLOSE – Close a symbol table	14-10
NSPC_DELETE – Delete an entry within a symbol table	14-11
%NSPC_FIND – Find an entry in a symbol table	14-12
NSPC_GETDATA – Retrieve an entry's data from a symbol table	14-13
NSPC_MOVE – Move an entry within a symbol table	14-14
%NSPC_OPEN – Open a symbol table	14-15
NSPC_PUTDATA – Modify an entry's data in a symbol table	14-17
NSPC_RESET – Reset a symbol table	14-18
%NSPC_STOA – Map search index to entry in symbol table	14-19

Introduction to the Synergy Symbol Table API Routines

The Synergy symbol table API provides runtime maintenance of sets of names, along with ancillary information associated with those names. These names are simply text strings of up to 255 characters. The ancillary information associated with each name can be up to 100 characters.

Each entry within a symbol table has three components: a name, an access code, and user-defined data. Each entry is added to two lists: an access list and a search list, which are described below.

As entries are added to a symbol table, memory is expanded as required. Deleting an item from a symbol table reclaims the memory associated with that item immediately.

Requirements

The file **DBLDIR:namspc.def** defines the macros and constants required by the symbol table API. You must include this file in your program using the `.INCLUDE` compiler directive.

All programs that use the symbol table API must be built using the compiler options **-T** (which trims trailing null arguments from XCALLs and function references) and **-X** (which automatically declares all undefined function references as external `^VAL` functions).

Access list

The access list is a user-configurable list. The access code for an entry is that entry's position within the access list. An entry is initially added at the end of the access list but can subsequently be moved anywhere within the list using `%NSPC_MOVE`.

Search list

The search list maintains an alphabetical order of the entries by name. This allows for high-speed lookups of entry names. This list cannot be reordered by the user.

Sample program

The following example is two separate functions that the user must call. One reads **synergy.ini** and populates the symbol table, and the other does a search to see if a particular entry is there.

```

;=====
; Module:          prcini.dbl
;
; Description:      Process an .INI file
;=====

function prcini_file ,^val ,reentrant

; Description: Read an .ini file. Returns ID of the loaded symbol table.

    a_file          ,a

; Notes: This routine parses an .ini file and stores the initialization
; settings into a symbol table for future retrieval.

.include "DBLDIR:namspc.def"

.define MAXITM      ,10          ;The maximum parse tokens we'll need
.define MAXSECLEN    ,64          ;Maximum length of a section identifier
.define LINLEN       ,100         ;Line size

stack record
    chn              ,i4          ;Input processing channel
    new_id            ,i4          ;New entry id
    nsid              ,i4          ;The symbol table ID
    icnt              ,i4          ;Loaded item count
    linsiz            ,i4          ;Input line length
    ndx               ,i4          ;Item index
    ipos              ,[MAXITM]i4 ;Item positions
    ilen              ,[MAXITM]i4 ;Item lengths
    ityp              ,[MAXITM]i4 ;Item types
    pos               ,i4          ;Same as the arrays for a given entry
    len               ,i4
    typ               ,i4
    ix                ,i4          ;A scratch integer
    group ident       ,a           ;Item storage
    id1               ,[MAXSECLEN]a1
endgroup
    group line        ,a           ;Input line
    ln1               ,[LINLEN]a1
endgroup

```

Synergy Symbol Table API

Introduction to the Synergy Symbol Table API Routines

```
secnam          ,a(MAXSECLen) ;The saved section name
tagname         ,a(MAXSECLen) ;The current tag name

proc
clear nsid                      ;Init fields
open(chn,i,a_file) [err=nofile] ;Open the .ini file
repeat                      ;Skip to the first section
begin
    call nextline
    if (linsiz .eq. 0)
        goto done
    if (idl .eq. '[')
        exitloop
    end
nsid = %nspc_open(D_NSPC_CASE,LINLEN,,MAXSECLen*2)
repeat
begin
    call nextitem                ;Get the section's name
    if ((ndx.gt.icnt) .or. (typ .ne. I_IDENT))
        nextloop
    secnam = ident
    upcase secnam
    repeat
        begin
            call nextline                ;Get next nonblank line
            if (linsiz .eq. 0)
                goto done
            if (idl .eq. '[')                ;Process another section
                exitloop
            tagname = ident                ;Get the tag name
            upcase tagname
            new_id = %nspc_add(nsid,%atrim(secnam)+ "!" + %atrim(tagname),
&                                line(pos,linsiz))
        end
    end
done,
    close chn                      ;Close the channel and return
    freturn nsid                    ; with success flagged

nofile,
    freturn 0                      ;Return with error flagged
```

```

; Get next nonblank line.  If at EOF, linsiz will be zero.

nextline,
  repeat
    begin
      reads(chn,line,eof)                ;Get the next line
      if ((%rdlen .eq. 0) .or. (%atrim(line) .eq. " "))
        nextloop                        ;Ignore empty lines
      linsiz = %trim(line)
      xcall s_parse(line(1:linsiz), 1, MAXITM, ipos, ilen, ityp, icnt,
&                                     ndx)
      clear ndx                        ;Get the first item
      call nextitem
      if ((ndx .le. icnt) .and. (idl .ne. ''))
        ;Exit if not an empty line
      return
    end
eof,
  clear linsiz                        ;Signal EOF
  return

; Get the next parsed item on the line

nextitem,
  len = ilen[ndx += 1]                ;Move to the next item
  pos = ipos[ndx]
  using (typ = ityp[ndx]) select
(I_SPACE),
  goto nextitem                        ;Ignore spaces
(I_ANUM,I_IDENT),
  begin
    ident = line(pos:len)                ;Get case-sensitive identifier
    typ = I_IDENT                        ;Map ANUM to IDENT
  end
(I_SPECIAL),
  idl = ln1(pos)                        ;Get the special character
(),
  clear idl
  endusing
  return
endfunction

subroutine prcini_close

```

Synergy Symbol Table API

Introduction to the Synergy Symbol Table API Routines

```
; Close the symbol table

a_nsid          ,n                ;ID of .ini file's symbol table

proc
  xcall nspc_close(a_nsid)
  xreturn
endsubroutine

function prcini_find,^val,reentrant

; Find an .ini file entry

a_nsid          ,n                ;ID of .ini file's symbol table
a_section       ,a                ;The section name
a_entry         ,a                ;The entry tag
a_result        ,a                ;Returned found entry

; Return Value: TRUE if the entry was found

record
  result        ,i4
  section       ,a(MAXSECLen)
  entry_tag     ,a(MAXSECLen)

proc
  clear section, entry_tag
  if (^passed(a_section) .and. a_section)
    begin
      section = a_section
      upcase section
    end
  if (^passed(a_entry) .and. a_entry)
    entry_tag = a_entry
  result = %nspc_find(a_nsid,%atrim(section)+"!"+"%atrim(entry_tag),
&                  a_result)
  freturn(result)
endfunction
```

Synergy Symbol Table API Routines

The symbol table API routines begin with `%NSPC_` and are documented on the pages that follow.

%NSPC_ADD – Add an entry to a symbol table

```
access_code = %NSPC_ADD(id, entry_name, [data][, access_code])
```

Return value

access_code

The access code for the new entry, or 0 if the entry already exists. **(n)**

Arguments

id

The ID of the symbol table. **(n)**

entry_name

The name of the entry. **(a)**

data

(optional) The user-defined data to associate with the entry. **(a)**

access_code

(optional) If passed, move the added item to the position specified within the symbol table. **(n)**

Discussion

The %NSPC_ADD function adds an entry to a symbol table.

If an entry named *entry_name* already exists in the symbol table, the new entry is not added.

Entry_name can be up to 255 characters in length.

Data can be up to 100 characters in length, as defined in %NSPC_OPEN.

The *access_code* argument enables you to request an access code for the entry being added. Passing *access_code* is equivalent to calling NSPC_ADD without the argument, followed by an NSPC_MOVE with the returned access code as the *old_entry* argument and the requested access code as the *new_entry* argument.

Examples

The following example adds a **username** entry to the **user_id** symbol table.

```
new_user = %nspc_add(user_id, username, user_data)
```


%NSPC_ATOS – Map entry in symbol table to search index

index = %NSPC_ATOS (*id*, *access_code*)

Return value

index

The search index for a given entry in a symbol table based on its access code. (**n**)

Arguments

id

The ID of the symbol table. (**n**)

access_code

The access code of the entry to reference. (**n**)

Discussion

The %NSPC_ATOS function retrieves the search index associated with a given entry in a symbol table.



The search index and access code for any given entry may change with every addition, deletion, or modification of the associated symbol table. Be sure the access code has not changed since it was obtained.

See also

[%NSPC_STOA](#) on page 14-19 for mapping a search index to an entry.

Examples

The following example retrieves the search index for entry **user**.

```
srch = %nspc_atos(user_id, user)
```

NSPC_CLOSE – Close a symbol table

```
xcall NSPC_CLOSE(id)
```

or

```
return = %NSPC_CLOSE(id)
```

Return value

return

Always a value of 1. (**n**)

Arguments

id

The ID of the symbol table. (**n**)

Discussion

When a symbol table is no longer required, it should be removed. The %NSPC_CLOSE function deletes an entire symbol table.

We recommend that you XCALL NSPC_CLOSE instead of accessing it as a function.

Examples

The following example deletes symbol table **user_id**.

```
xcall nspc_close(user_id)
```

NSPC_DELETE – Delete an entry within a symbol table

```
xcall NSPC_DELETE(id, access_code)
```

or

```
return = %NSPC_DELETE(id, access_code)
```

Return value

return

Always a value of 1. **(n)**

Arguments

id

The ID of the symbol table. **(n)**

access_code

The access code of the entry to reference. **(n)**

Discussion

The %NSPC_DELETE function deletes an entry from a symbol table.

We recommend that you XCALL NSPC_DELETE instead of accessing it as a function.

When an entry is deleted from a symbol table, the access code and search index for other entries within the symbol table may change.

Examples

The following example deletes entry **old_user** from symbol table **user_id**.

```
xcall nspc_delete(user_id, old_user)
```

%NSPC_FIND – Find an entry in a symbol table

access_code = %NSPC_FIND(*id*, [*entry_name*], [*data*][, *found_id*])

Return value

access_code

If *entry_name* is specified, this function returns either the access code of the found entry or zero if not found. If *entry_name* is not specified, the number of entries within *id* is returned. **(n)**

Arguments

id

The ID of the symbol table to search. **(n)**

entry_name

(optional) The name of the entry for which to search. **(a)**

data

(optional) The returned user-defined data associated with the entry. **(a)**

found_id

(optional) The returned symbol table ID. **(n)**

Discussion

The %NSPC_FIND function searches for an entry within a symbol table.

If *entry_name* is specified, *id* is searched for the entry *entry_name*.

If *data* is specified and an entry is found, *data* is loaded with the user-defined data associated with the entry. The maximum size of *data* is 100 characters, as defined in %NSPC_OPEN.



Because an entry's access code can change for every add, delete, or move within a symbol table, a FIND should always immediately precede any usage of an entry's access code.

Examples

The following example searches for entry **new_user** in symbol table **user_id**. If **new_user** is not found, it adds it to the symbol table.

```
if (.not. %nspc_find(user_id, new_user)
    user_id = %nspc_add(user_id, new_user, user_data)
```

NSPC_GETDATA – Retrieve an entry’s data from a symbol table

```
xcall NSPC_GETDATA (id, access_code, data[, entry_name])  
or  
return = %NSPC_GETDATA (id, access_code, data[, entry_name])
```

Return value

return
Always a value of 1. **(n)**

Arguments

id
The ID of the symbol table. **(n)**

access_code
The access code of the entry. **(n)**

data
The returned user-defined data associated with the entry. **(a)**

entry_name
(optional) The returned name of the entry. **(a)**

Discussion

The %NSPC_GETDATA function retrieves the data for an entry within a symbol table.

If D_NSPC_CASE was not specified when the symbol table was created, the *entry_name* returned is all in one case, either uppercase or lowercase, depending on the search engine in the environment.

Data can be up to 100 characters in length, as defined in %NSPC_OPEN.

We recommend that you XCALL NSPC_GETDATA instead of accessing it as a function.

See also

[NSPC_PUTDATA](#) on page 14-17 for modifying an entry’s data.

Examples

The following example gets the user-defined data and name for entry **user**.

```
xcall nspc_getdata(user_id, user, user_data, username)
```

NSPC_MOVE – Move an entry within a symbol table

```
xcall NSPC_MOVE(id, old_entry, new_entry)  
  
or  
  
return = %NSPC_MOVE(id, old_entry, new_entry)
```

Return value

return
Always a value of 1. (**n**)

Arguments

id
The ID of the symbol table. (**n**)

old_entry
The current access code for the entry. (**n**)

new_entry
The new access code for the entry. (**n**)

Discussion

The %NSPC_MOVE function modifies the access code of an entry within a symbol table.

Entries between *old_entry* and *new_entry* are shifted accordingly.

We recommend that you XCALL NSPC_MOVE instead of accessing it as a function.

Examples

The following example moves an entry from access code **curr_user** to **new_user** in symbol table **user_id**.

```
xcall nspc_move(user_id, current_user, new_user)
```

%NSPC_OPEN – Open a symbol table

id = %NSPC_OPEN ([*flags*], [*size*], {*reserved*}, {*reserved*}, [*begin_count*][, *expand_count*])

Return value

id

An identifier for the symbol table created. (**n**)

Arguments

flags

(optional) One of the following control flags defined in **namspc.def** or the sum of D_NSPC_CASE and D_NSPC_SPACE: (**n**)

D_NSPC_NONE No flags specified (default).

D_NSPC_CASE Case significant.

D_NSPC_SPACE Leading and trailing spaces significant.

size

(optional) The size of the user data to be associated with each entry. (The maximum is 100.) (**n**)

{*reserved*}

A reserved placeholder for internal use. If passed, the value must be 0.

begin_count

(optional) The initial number of entries to allocate. (**n**)

expand_count

(optional) The number of entries by which to expand when required. (**n**)

Discussion

The %NSPC_OPEN function creates a symbol table object.

If *flags* contains D_NSPC_CASE, the case of any letters in the entry names is significant. If *flags* contains D_NSPC_SPACE, leading and trailing spaces in the entry names are significant. D_NSPC_NONE is defined for convenience and should be used if neither of the above flags are used.

Size can be a maximum of 100 bytes. If *size* is not specified, it defaults to 4 bytes.

Space is initially allocated for *begin_count* number of entries. Whenever the allocations for the symbol table are depleted, an additional *expand_count* number of entries are allocated, using the same arguments.

If *length* is not specified, it defaults to 10 bytes. If *begin_count* or *expand_count* are not specified, each defaults to 32 entries.

Examples

The following example creates a symbol table for the user name, with each entry large enough to contain **user_data**.

```
user_id = %nspc_open(, ^size(user_data))
```


NSPC_PUTDATA – Modify an entry’s data in a symbol table

```
xcall NSPC_PUTDATA (id, access_code, data)
```

or

```
return = %NSPC_PUTDATA (id, access_code, data)
```

Return value

return

Always a value of 1. (**n**)

Arguments

id

The ID of the symbol table. (**n**)

access_code

The access code of the entry. (**n**)

data

The user-defined data to associate with the entry. (**a**)

Discussion

The %NSPC_PUTDATA function modifies the user-defined data for an entry within a symbol table.

Data can be up to 100 characters in length, as defined in %NSPC_OPEN.

We recommend that you XCALL NSPC_PUTDATA instead of accessing it as a function.

See also

[NSPC_GETDATA on page 14-13](#) for retrieving an entry’s data.

Examples

The following example updates the user-defined data for **user**.

```
xcall nspc_putdata(user_id, user, user_data)
```

NSPC_RESET – Reset a symbol table

```
xcall NSPC_RESET(id)
```

or

```
return = %NSPC_RESET(id)
```

Return value

return

Always a value of 1. (**n**)

Arguments

id

The ID of the symbol table. (**n**)

Discussion

The %NSPC_RESET function resets a symbol table to its initial state, using the arguments passed on the initial call to %NSPC_OPEN.

A symbol table should be reset when you want to release unused memory. When entries are deleted in a symbol table, resources are still allocated until the symbol table is reset or closed.

We recommend that you XCALL NSPC_RESET instead of accessing it as a function.

Examples

The following example initializes symbol table **user_id**.

```
xcall nspc_reset(user_id)
```

%NSPC_STOA – Map search index to entry in symbol table

access_code = %NSPC_STOA(*id*, *index*)

Return value

access_code

One of the following values: (**n**)

If <i>index</i> is	%NSPC_STOA returns
Between 1 and the number of entries within the symbol table	The access code of the entry that corresponds to the search index
Less than 1	1
Greater than the number of entries within the symbol table	The number of entries

Arguments

id

The ID of the symbol table. (**n**)

index

The search index for an entry. (**n**)

Discussion

The %NSPC_STOA function retrieves the entry associated with a search index in a symbol table.



The search index and access code for any given entry may change with every addition, deletion, or modification of the associated symbol table.

See also

[%NSPC_ATOS](#) on page 14-9 for mapping an entry to a search index.

Examples

The following example retrieves the entry access code for search index **srch**.

```
user = %nspc_stoa(user_id, srch)
```


15

Synergy Routine Call Block API

Introduction to the Synergy Routine Call Block API 15-2

Describes how to make calls using a routine call block.

Synergy Routine Call Block API Routines 15-5

This API includes the following routines:

%RCB_CALL – Call a routine using a routine call block	15-6
%RCB_CREATE – Create a routine call block	15-8
RCB_DELARG – Delete an argument from a routine call block	15-10
RCB_DELETE – Delete a routine call block	15-11
RCB_INSARG – Insert argument into an existing routine call block.....	15-12
RCB_SETARG – Set an argument in the routine call block.....	15-15
RCB_SETARGS – Set multiple arguments in a single call	15-19
RCB_SETFNC – Specify a routine to call	15-20

Introduction to the Synergy Routine Call Block API

The Synergy routine call block API enables you to dynamically generate routine calls at runtime, rather than at compile time. Generating routine calls dynamically is useful when you need to adjust calls and/or the arguments passed to them based on changing conditions in your program. The routine call block API can be used to dynamically generate both local and remote routine calls. (If you are using this API remotely, you should be using *x/NetLink Synergy Edition*.)

In Synergy/DE, dynamic call generation requires creating and using a routine call block (RCB). An RCB is a block of memory that contains the information necessary (the routine name, ELB, and arguments) to make a routine call. Once you have created an RCB, it is identified by a memory handle known as the *rcbid*, the routine call block ID. An RCB can be used repeatedly, altered (for example, by inserting or changing arguments), or completely rebuilt for another call.

The routine call block API can also be used to improve performance in situations where many calls are being made to the same routine with the same arguments. By reducing the number of times that the RCB has to be created, performance is enhanced.

Making calls using a routine call block

This section explains how to create and use RCBs to generate routine calls dynamically. To use the RCB routines, you must `.INCLUDE DBLDIR:rcbapi.def` in your programs.



If you are making calls remotely, refer to [Part II](#) of the *Developing Distributed Synergy Applications* manual for more information about the `RX_xxx` routines and using *x/ServerPlus* to make remote calls.

Follow these steps:

1. If you are making dynamically generated calls remotely, request an *x/ServerPlus* session with `%RX_START_REMOTE`.
2. Create the RCB with `%RCB_CREATE`. When you create the RCB, you specify the size. `%RCB_CREATE` returns the `rcbid`, which identifies the call block.
3. Specify the Synergy routine to call:

If you are making calls	Do this
Locally	Set the routine name with <code>RCB_SETFNC</code> . This step sets the name of the routine that will be called when the RCB is passed to <code>%RCB_CALL</code> .
Remotely	Specify the remote routine to be called with <code>RX_SETRTMFNC</code> .

4. Set up the arguments with `RCB_SETARG`, `RCB_SETARGS`, and/or `RCB_INSARG`. You can add arguments one at a time with `RCB_SETARG` or in a group with `RCB_SETARGS`. To add more arguments later, use `RCB_INSARG`. To remove arguments later, use `RCB_DELARG`.
5. Make the call with `%RCB_CALL` and pass the *rcbid*.
6. Adjust and recall as necessary. You can add additional arguments with `RCB_INSARG`, overwrite a particular argument with `RCB_SETARG`, and remove a particular argument with `RCB_DELARG`.
7. Delete the block with `RCB_DELETE`. RCBs that are not explicitly deleted with `RCB_DELETE` are deleted when the runtime exits. However, we recommend that you delete the block when you're finished using it to avoid tying up system resources.
8. If you are making calls remotely, close the *x/ServerPlus* session with `RX_SHUTDOWN_REMOTE`.

[Figure 15-1](#) shows the general process flow for performing this procedure both locally and remotely.

Synergy Routine Call Block API

Introduction to the Synergy Routine Call Block API

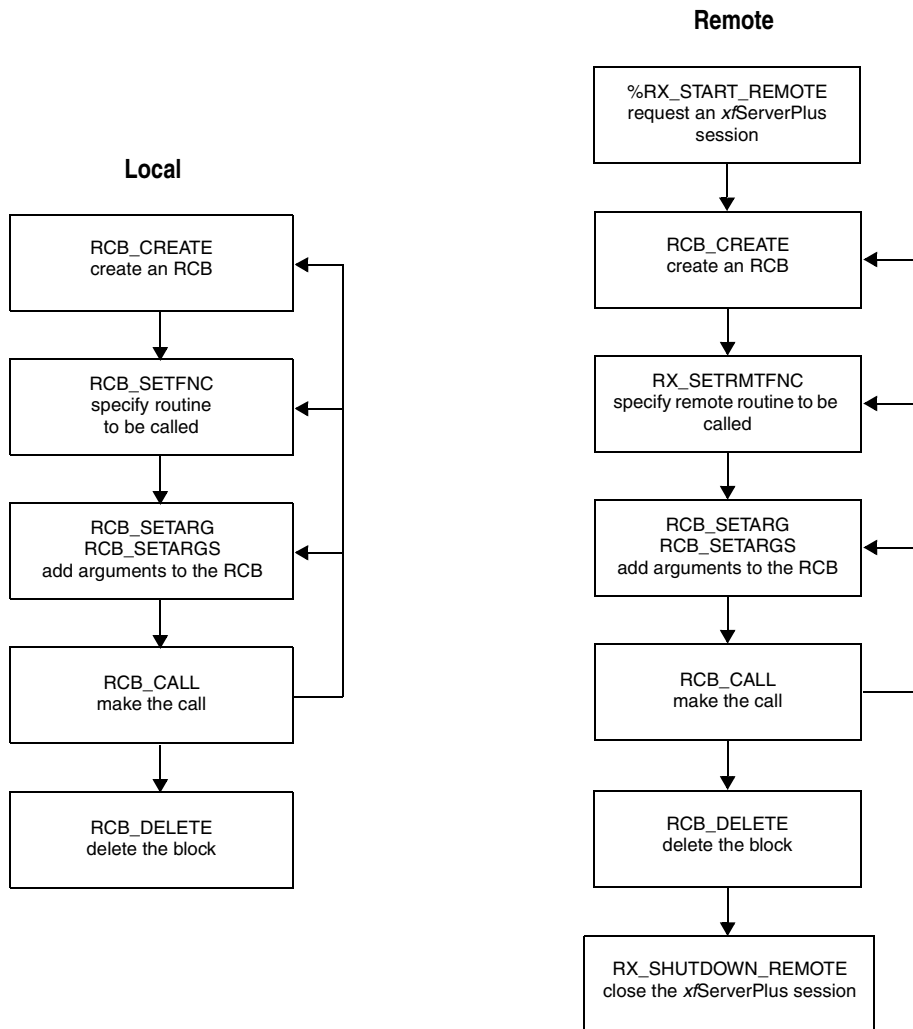


Figure 15-1. Making local and remote calls using a routine call block.

Synergy Routine Call Block API Routines

The Synergy routine call block API routines begin with `%RCB_` or `RCB_` and are documented on the pages that follow.

%RCB_CALL – Call a routine using a routine call block

```
result = %RCB_CALL(rcbid)
```

or

```
xcall RCB_CALL(rcbid)
```

Return value

result

The ^VAL result of the Synergy function. This is meaningful only if the routine being called is a ^VAL function. If the routine being called is a subroutine or a non-^VAL function, *result* is set to 1. If the routine being called is a non-^VAL function, the FRETURN value is returned as the first argument of the routine call block argument list.

Arguments

rcbid

The identifier for the routine call block. This is the return value from %RCB_CREATE. (**n**)

Discussion

This function is used to call a routine once the routine call block is built.

If the function to be called is specified through %RX_SETRMTFNC, a remote call is executed. If the function to be called is specified through %RCB_SETFNC, a local call is executed. When %RCB_CALL is used to call a remote routine, all arguments specified as ^VAL are passed as an integer, and all arguments specified as ^REF have the data passed according to the type and length specified in the %RCB_SETARG call. If an argument has not been specified, it is passed as a null argument.

See also

[“Parameter definitions” on page 2-33.](#)

Examples

```
subroutine ABC
    arg1,    n
    arg2,    n

proc
    arg1 = arg1 + arg2
    xreturn
endsubroutine
```

```

function DEF
    arg1,    n
    arg2,    n

proc
    freturn(arg1 + arg2)
endfunction

function GHI, ^VAL
    arg1,    n
    arg2,    n

proc
    freturn(arg1 + arg2)
endfunction

subroutine JKL
    rcbid,    i4
    a,        i4
    b,        i4
    result,   i4
    value,    i4

proc
    a = 3
    b = 4
    rcbid = %rcb_create(2)
    xcall rcb_setfnc(rcbid, "ABC")
    xcall rcb_setargs(rcbid, 1, a, b)
    result = %rcb_call(rcbid)      ;result = 1, a = 7, b = 4
    xcall rcb_delete(rcbid)
    a = 3
    b = 4
    rcbid = %rcb_create(3)
    xcall rcb_setfnc(rcbid, "DEF")
    xcall rcb_setargs(rcbid, 1, result, a, b)
    value = %rcb_call(rcbid)      ;value = 1, result = 7, a = 3, b = 4
    xcall rcb_delete(rcbid)
    a = 3
    b = 4
    rcbid = %rcb_create(2)
    xcall rcb_setfnc(rcbid, "GHI")
    xcall rcb_setargs(rcbid, 1, a, b)
    result = %rcb_call(rcbid)      ;result = 7, a = 3, b = 4
    xcall rcb_delete(rcbid)
    xreturn
endsubroutine

```

%RCB_CREATE – Create a routine call block

```
id = %RCB_CREATE (numargs, [flags][, old_rcbid])
```

Return value

id

The ID of the created routine call block. (n)

Arguments

numargs

The number of arguments to reserve space for initially. Zero is a valid value, indicating that the argument list contains no arguments. (n)

flags

(optional) One or more of the following: (n)

DM_STATIC Make the routine call block ID persistent to all routines.

DRCB_REPLICATE Replicate the caller's argument list.

old_rcbid

(optional) A previously created call block that can now be reused. (n)

Discussion

%RCB_CREATE creates a routine call block in the size specified.

Note that *numargs* specifies only the number of arguments; the arguments are not present until they are added with RCB_SETARG or RCB_SETARGS.

DRCB_REPLICATE can be added to DM_STATIC to duplicate the caller's argument list and make the routine call block persistent until it is explicitly deleted. If DRCB_REPLICATE is specified and remote execution is attempted, all arguments are assumed to be passed by descriptor. An explicit RCB_SETARG done for each ^VAL and ^REF argument for remote execution. If DM_STATIC is not set, the routine call block ID is local to this routine and is cleaned up automatically without a call to RCB_DELETE when the current routine exits via FRETURN or XRETURN.

Specifying *old_rcbid* enables you to reuse a block that is no longer needed. The old information is released, and the block is initialized to hold the number of arguments specified.

Examples

```

record
    rcbid      ,i4
    arg_1      ,a3

proc
    rcbid = %rcb_create(1)           ;Create an RCB block with 1 arg
    xcall rcb_setarg(rcbid, arg_1, 1) ;Set up arg_1 as argument #1
    xcall rcb_setfnc(rcbid, "routine") ;Set function to be called
    xcall rcb_call(rcbid)             ;Call function as xcall routine(arg_1)
    xcall rcb_delete(rcbid)           ;Delete RCB block
end

```

Also see Examples on [page 15-6](#).

RCB_DELARG – Delete an argument from a routine call block

```
xcall RCB_DELARG (rcbid, argnum)
```

Arguments

rcbid

The identifier for the routine call block. (**n**)

argnum

The one-based index of the argument to be deleted. (**n**)

Discussion

If *rcbid* is invalid, an error is signaled.

If *argnum* is not passed or is less than or equal to 0, an error is signaled.

Examples

```
record
    rcbid      , i4
    arg_1      , a3

proc
    rcbid = %rcb_create(3)           ;Create an RCB block with 3 args
    xcall rcb_setarg(rcbid, arg_1, 1) ;Set up arg_1 as argument #1
    xcall rcb_delarg(rcbid, 3)        ;Delete argument #3
    xcall rcb_delarg(rcbid, 2)        ;Delete argument #2
    xcall rcb_setfnc(rcbid, "routine") ;Set function to be called
    xcall rcb_call(rcbid)             ;Call function as xcall routine(arg_1)
    xcall rcb_delete(rcbid)           ;Delete RCB block
end
```

RCB_DELETE – Delete a routine call block

```
xcall RCB_DELETE(rcbid)
```

Arguments

rcbid

The ID of the routine call block to delete. Once deleted, the block cannot be used. (n)

Discussion

Any routine call blocks that are not explicitly deleted are deleted when the runtime exits. However, we recommend that you delete the block when you are finished using it to avoid tying up system resources.

Examples

```
record
    rcbid      ,i4
    arg_1      ,a3

proc
    rcbid = %rcb_create(1)           ;Create an RCB block with 1 arg
    xcall rcb_setarg(rcbid, arg_1, 1) ;Set up arg_1 as argument #1
    xcall rcb_setfnc(rcbid, "routine") ;Set function to be called
    xcall rcb_call(rcbid)             ;Call function as xcall routine(arg_1)
    xcall rcb_delete(rcbid)           ;Delete RCB block
end
```

Also see Examples on [page 15-6](#).

RCB_INSARG – Insert argument into an existing routine call block

```
xcall RCB_INSARG (rcbid, argnum, [data, [type], [length], [prec], [flags], [numdims],  
& [dim1, ..., dim9]])
```

Arguments

rcbid

The ID of a routine call block previously created with %RCB_CREATE. (**n**)

argnum

The one-based index of the argument. (**n**)

data

(optional) The data area to be referenced by the argument. (**a** or **n**)

type

(optional) The type of the argument: (**n**)

D_TYPE_A	Alpha
D_TYPE_D	Decimal
D_TYPE_ID	Implied-decimal
D_TYPE_I	Integer
D_TYPE_P	Packed
D_TYPE_IP	Implied-packed
D_TYPE_OBJ	Object handle
D_TYPE_HANDLE	<i>Data</i> is a memory handle that references a variable-length argument whose size may exceed 64K (x/NetLink Synergy Edition only).
D_TYPE_MEMARG	<i>Data</i> is a memory handle that references the argument's data.
D_TYPE_REF	Address passed by reference
D_TYPE_VAL	Integer passed by value

length

(optional) The length, in bytes, of the argument. (**n**)

prec

(optional) The precision (decimal places) of the argument. (**n**)

flags

(optional) A specification of other features: **(n)**

DRCB_READONLY The data passed cannot be modified. This flag is ignored if *type* is D_TYPE_VAL. DRCB_READONLY is implied if an arithmetic expression or literal is passed.

DRCB_ARRAY The data passed is to be an array according to the *numdims* and *dim1* through *dim9* arguments. *Numdims* and *dim1* through *dim9* are ignored unless DRCB_ARRAY is specified.

numdims

(optional) The number of dimensions of the argument (if the argument is an array). Valid values are 1 through 9. **(n)**

dim1 through *dim9*

(optional) Up to nine dimension specifications (if the argument is an array). **(n)**

Discussion

If *rcbid* is invalid, an error is signaled.

If *argnum* is not passed or is less than or equal to 0, an error is signaled.

The *data* argument must be passed if any of the other optional arguments are passed, because the other arguments describe the data. If this argument (or any subsequent optional argument) is not passed, the argument is assumed to be a null argument at the time RCB_CALL is executed. If *type* is D_TYPE_HVA or D_TYPE_OBJ, *data* is a memory handle.

If *type* is not passed or has a value of 0, the data type of *data* is used. If you are using D_TYPE_HVA or D_TYPE_OBJ, the *type* argument is not optional.

If *length* is not passed or has a value of 0, the length (^SIZE) of *data* is used. If *type* is D_TYPE_VAL, *length* is used only for the source data, and the length of the argument is set to 4 (32 bits). If *type* is D_TYPE_HVA or D_TYPE_OBJ and *length* is passed, *length* must be less than the size of the memory associated with the memory handle.

If *prec* is not passed, the precision of *data* is used. This argument is used only when *type* is D_TYPE_ID or D_TYPE_IP.

If the argument is an array, specify the DRCB_ARRAY flag and the *numdims* and *dim#* arguments.

The D_TYPE_HVA type is used to pass a single nonarrayed argument as a memory handle (in *data*) when you're using *xfNetLink* Synergy. This argument can be larger than 64K. RCB_INSARG looks at the memory handle, which indicates the location of the data on the client, and passes the data to *xfServerPlus*. *xfServerPlus* creates a memory area, puts the data in it, and then returns the memory handle for that area to the Synergy server routine. Therefore, your Synergy routine should be written to expect an **i4** value. Any return data from *xfServerPlus* is placed back in the same memory area on the client that it originally came from. If the memory area allocated to the handle is

resized within the Synergy routine on the server, it is resized accordingly on the client upon returning from the routine. Trailing blanks are trimmed when the data is returned to the client, so the resized memory area on the client might not be exactly the same size it was on the server. Do not use the D_TYPE_HVA type when using the routine call block API locally.

The D_TYPE_OBJ type is used to specify that the *data* argument is a memory handle containing the data to be used for the argument. The type of the argument is specified by adding any valid data type (except D_TYPE_HVA) to D_TYPE_OBJ (for example, D_TYPE_OBJ + D_TYPE_D).

D_TYPE_OBJ can be used with x/NetLink Synergy or when using the routine call block API locally. The routine receiving the argument should declare the argument in a normal manner.

On 32-bit systems, when D_TYPE_OBJ is specified, the size of an argument or a single element of an array is limited to 64K. When the scope of an array is larger than 64K, D_TYPE_OBJ can be used to specify the argument as long as the size of a single element does not exceed 64K.

Examples

```
record
    rcbid          ,i4
    arg_1          ,a3
    arg_2          ,d3
    arg_3          ,d6.3
    arg_4          ,a4
    arg_5          ,i4

proc
    rcbid = %rcb_create(2)                ;Create an RCB block with 2 args
    xcall rcb_setarg(rcbid, arg_1, 1)      ;Set up arg_1 as argument #1
    xcall rcb_setarg(rcbid, arg_3, 2)      ;Set up arg_3 as argument #2
    xcall rcb_insarg(rcbid, 2, arg_2)      ;Expand argument block by
                                           ; inserting arg_2 as argument #2
    xcall rcb_insarg(rcbid, 4, arg_4, D_TYPE_D) ;Expand argument block
                                           ; by inserting arg_4 as
                                           ; decimal
    xcall rcb_insarg(rcbid, 5, arg_5,,,, DRCB_READONLY) ;Set up arg_5
                                           ; as a literal
    xcall rcb_setfnc(rcbid, "routine")     ;Set function to be called
    xcall rcb_call(rcbid)                  ;Call function as xcall routine(arg_1,
                                           ; arg_2, arg_3, ^d(arg_4), arg_5)
    xcall rcb_setarg(rcbid,, 3)            ;Set up arg_3 as a null argument
    xcall rcb_call(rcbid)                  ;Call function as xcall routine(arg_1,
                                           ; arg_2,, ^d(arg_4), arg_5)
    xcall rcb_delarg(rcbid, 3)             ;Remove argument #3
    xcall rcb_call(rcbid)                  ;Call function as xcall routine(arg_1,
                                           ; arg_2, ^d(arg_4), arg_5)
    xcall rcb_delete(rcbid)                ;Delete RCB block
end
```

RCB_SETARG – Set an argument in the routine call block

```
xcall RCB_SETARG (rcbid, [data], argnum, [type], [length], [precision], [flags], [numdims],
& [dim1, ..., dim9])
```

Arguments

rcbid

The ID of a routine call block previously created with %RCB_CREATE. (**n**)

data

(optional) The data area to be referenced by the argument. (**a** or **n**)

argnum

The one-based index of the argument. (**n**)

type

(optional) The type of the argument: (**n**)

D_TYPE_A	Alpha
D_TYPE_D	Decimal
D_TYPE_ID	Implied-decimal
D_TYPE_I	Integer
D_TYPE_P	Packed
D_TYPE_IP	Implied-packed
D_TYPE_OBJ	Object handle
D_TYPE_HANDLE	<i>Data</i> is a memory handle that references a variable-length argument whose size may exceed 64K (x/NetLink Synergy Edition only).
D_TYPE_MEMARG	<i>Data</i> is a memory handle that references the argument's data.
D_TYPE_REF	Address passed by reference
D_TYPE_VAL	Integer passed by value

length

(optional) The length, in bytes, of the argument. (**n**)

precision

(optional) The precision (decimal places) of the argument. (**n**)

flags

(optional) A specification of other features: **(n)**

DRCB_READONLY The data passed cannot be modified. This flag is ignored if *type* is `D_TYPE_VAL`. `DRCB_READONLY` is implied if an arithmetic expression or literal is passed.

DRCB_ARRAY The data passed is to be an array according to the *numdims* and *dim1* through *dim9* arguments. *Numdims* and *dim1* through *dim9* are ignored unless `DRCB_ARRAY` is specified.

If both `DRCB_READONLY` and `DRCB_ARRAY` are to be specified, the values may be added together.

numdims

(optional) The number of dimensions of the argument (if the argument is an array). Valid values are 1 through 9. **(n)**

dim1 through *dim9*

(optional) Up to nine dimension specifications (if the argument is an array). **(n)**

Discussion

Using this routine, you can create routine call block argument entries for any data type, including `^REF` and `^VAL` arguments.

If *data* is not passed, the argument is set to a null argument and is not passed.

Unless they're explicitly overridden by the `RCB_SETARG` arguments, all other features of the argument (for example, length, type, and whether it's global or temporary) are derived from the features of *data*. The data referenced by the argument at the time of the `RCB_CALL` is the data passed to the subroutine, with the exception of `^VAL` arguments that are determined at the time of the `RCB_SETARG` call.

If *length* or *precision* is specified, the dimensional qualities of the *data* variable are lost unless explicitly specified through *numdims* and *dim1* through *dim9*. If *numdims* is specified as zero, the array attribute is turned off. If *data* is a real array, any modifications to the length or dimensions must fall within the bounds of the real array or an "Invalid subscript specified" error (`$ERR_SUBSCR`) results. If *data* is a real array and if a different *type* or *length* is specified, the resulting argument is not an array unless `DRCB_ARRAY` has been specified along with the dimensional information. If *type* is `D_TYPE_MEMARG` or `D_TYPE_HANDLE`, *data* is a memory handle.

If *argnum* is not passed, or if it is less than or equal to 0, or greater than the current number of arguments (`%RCB_CREATE`'s arguments plus `%RCB_INSARG`'s arguments minus `%RCB_DELARG`'s arguments), a "Value out of range" error (`$ERR_OUTRNG`) is signaled.

If *type* is not passed or has a value of 0, the data type of *data* is used. If you are using D_TYPE_MEMARG or D_TYPE_HANDLE, the *type* argument is not optional.

If *length* is not passed or has a value of 0, the length (^SIZE) of *data* is used. If *type* is D_TYPE_VAL, *length* is used only for the source data, and the length of the argument is set to 4 (32 bits). If *type* is D_TYPE_MEMARG or D_TYPE_HANDLE and *length* is passed, *length* must be less than the size of the memory associated with the memory handle.

If *precision* is not passed, the precision of *data* is used. This argument is used only when *type* is D_TYPE_ID or D_TYPE_IP.

If the argument is an array, specify the DRCB_ARRAY flag and the *numdims* and *dim#* arguments.

If you pass the same data reference on multiple calls to RCB_SETARG, all of the arguments reference the same data. The exception to this rule is if the type is D_TYPE_VAL, in which case the value of data at the time of the call to RCB_SETARG is used.

You may use this routine to overwrite an argument that was previously specified in another call to RCB_SETARG.

The D_TYPE_MEMARG type is used to specify that *data* is a memory handle referencing the data to be used for the argument. The data type to be applied to the argument when it is passed is specified by adding one or more of the *type* qualifiers (except D_TYPE_HANDLE) to D_TYPE_MEMARG (for example, D_TYPE_MEMARG + D_TYPE_D). The other RCB_SETARG arguments can be specified to customize the argument. The scope of the argument is limited to the size of the data associated with the memory handle. When using D_TYPE_MEMARG on 32-bit systems, the size of an argument or single array element is limited to 64K. The entire array, however, may be larger than 64K. D_TYPE_MEMARG can be used with *x/NetLink* Synergy or when using the routine call block API locally. The routine receiving the argument should declare the argument in a normal manner.

The D_TYPE_HANDLE type is used to pass a variable-length argument as a memory handle (in *data*) when you're using *x/NetLink* Synergy. This argument can be larger than 64K. RCB_SETARG looks at the memory handle, which indicates the location of the data on the client, and passes the data to *x/Server Plus*. *x/ServerPlus* creates a memory area, puts the data in it, and then passes the memory handle for that area to the Synergy server routine. Therefore, your Synergy routine should be written to expect an **i4** value. Any return data from *x/ServerPlus* is placed back in the memory area on the client that it originally came from. If the memory area allocated to the handle is resized within the Synergy routine on the server, it is resized accordingly on the client upon returning from the routine. Trailing blanks are trimmed when the data is returned to the client, so the resized memory area on the client may be smaller than it was on the server. Do not use D_TYPE_HANDLE when using the routine call block API locally.

Examples

```
record
    rcbid      ,i4
    arg_1      ,a3
    arg_2      ,d3
    arg_3      ,d6.3
    arg_4      ,a4
    arg_5      ,i4

proc
    rcbid = %rcb_create(5)                ;Create an RCB block with 5 args
    xcall rcb_setarg(rcbid, arg_1, 1)      ;Set up arg_1 as argument #1
    xcall rcb_setarg(rcbid, arg_3, 3)      ;Set up arg_3 as argument #3
    xcall rcb_setarg(rcbid, arg_5, 5)      ;Set up arg_5 as argument #5
    xcall rcb_setarg(rcbid, arg_4, 4)      ;Set up arg_4 as argument #4
    xcall rcb_setarg(rcbid, arg_2, 2)      ;Set up arg_2 as argument #2
    xcall rcb_setfnc(rcbid, "routine")     ;Set function to be called
    xcall rcb_call(rcbid)                  ;Call function as xcall routine(arg_1,
                                          ; arg_2, arg_3, arg_4, arg_5)
    xcall rcb_delete(rcbid)                ;Delete RCB block
end
```

RCB_SETARGS – Set multiple arguments in a single call

```
xcall RCB_SETARGS (rcbid, startargnum, arglist)
```

Arguments

rcbid

The ID of a routine call block previously created with %RCB_CREATE. (**n**)

startargnum

The one-based starting position for setting arguments. (**n**)

arglist

The argument list. At least one argument must be set. (**a** or **n**)

Discussion

The RCB_SETARGS subroutine can only be used when data is passed by descriptor. (In other words, it cannot be used for ^REF and ^VAL arguments.)

If *startargnum* is less than or equal to 0 or greater than the number of arguments specified in %RCB_CREATE, an error is signaled.

You can include as many arguments in *arglist* as you want, as long as the total number set in the routine call block does not exceed the number specified in %RCB_CREATE.

Examples

```
record
    rcbid      ,i4
    arg_1      ,a3
    arg_2      ,d3
    arg_3      ,d6.3
    arg_4      ,a4
    arg_5      ,i4

proc
    rcbid = %rcb_create(5)           ;Create an RCB block with 5 args
    xcall rcb_setargs(rcbid, 1, arg_1, arg_2, arg_3, arg_4, arg_5)
    xcall rcb_setfnc(rcbid, "routine") ;Set function to be called
    xcall rcb_call(rcbid)           ;Call function as xcall routine(arg_1,
                                   ; arg_2, arg_3, arg_4, arg_5)
    xcall rcb_delete(rcbid)         ;Delete RCB block
end
```

Also see Examples on [page 15-6](#).

RCB_SETFNC – Specify a routine to call

```
xcall RCB_SETFNC (rcbid, routine[, elb])
```

or

```
xcall RCB_SETFNC (rcbid,, address)
```

Arguments

rcbid

The ID of the call block for the routine. (**n**)

routine

The name of a subroutine or function to be executed. (**a**)

elb

(optional) The name of the ELB in which this routine is located. This argument is not required if the ELB is explicitly opened before calling RCB_CALL or if the ELB is linked. (**a**)

address

The address for the ELB in which the routine is located. (**n**)

Discussion

RCB_SETFNC specifies a routine to call when the RCB is passed to RCB_CALL.

Any errors that result from the call itself (for example, \$ERR_RTNNF, \$ERR_NOMEM, \$ERR_INVRCBHND, \$ERR_NOTRCBHND, or \$ERR_XFBADMTHID) are signaled.

If *elb* is not already open, it is opened, along with any ELBs that it is linked against. If the specified ELB or any linked ELB is not found, the runtime generates a “File not found” error (\$ERR_FNF).

VMS

If the routine resides in a shared image, it will only be found if the shared image is explicitly specified or an OPENELB has been done on the shared image.

Examples

The example below specifies that the M_COLUMN routine in **WND:tklib.elb** will be called when **rcbid** is passed to RCB_CALL.

```
xcall rcb_setfnc(rcbid, "M_COLUMN", "WND:tklib.elb")
```

The following example specifies that the M_LDCOL routine will be called when **rcbid** is passed to RCB_CALL.

```
xcall rcb_setfnc(rcbid, "M_LDCOL")
```

Also see Examples on [page 15-6](#).

16

Synergy Socket API

Introduction to the Synergy Socket API 16-3

Describes the differences between Synergy Language sockets and Berkeley sockets, handling lost sockets, and Synergy socket function errors and provides some examples.

Synergy Socket API Routines 16-14

This API includes the following routines:

%SS_ACCEPT – Accept a connection on a stream socket.....	16-15
%SS_BIND – Bind a name to an unbound socket	16-16
%SS_CLOSE – Close a socket.....	16-17
%SS_CONNECT – Connect sockets or establish peer address	16-18
%SS_FATAL – Notify server that program is terminating abnormally.....	16-19
%SS_GETHOSTBYADDR – Get the host name corresponding to an Internet address	16-20
%SS_GETHOSTBYNAME – Get the Internet address corresponding to a host name.....	16-21
%SS_GETHOSTNAME – Get the name of the local host.....	16-22
%SS_GETPEERNAME – Return the name of the peer socket	16-23
%SS_GETSERVBYNAME – Return the port number given the service name	16-24
%SS_GETSERVBYPORT – Return the service name given the port number.....	16-25
%SS_GETSOCKNAME – Return the name associated with a socket	16-26
%SS_HTONL – Convert long integers from host to network byte order	16-27
%SS_HTONS – Convert short integers from host to network byte order.....	16-28
%SS_INET_ADDR – Convert a dotted text form Internet address	16-29
%SS_INET_NTOA – Convert a numeric Internet address	16-30
%SS_LISTEN – Listen for connections	16-31
%SS_NTOHL – Convert long integers from network to host byte order	16-32
%SS_NTOHS – Convert short integers from network to host byte order.....	16-33
%SS_RECV – Receive bytes from a socket.....	16-34
%SS_RECVBUF – Receive bytes from a socket into a buffer	16-36

%SS_RECVFROM – Receive bytes from a datagram socket..... 16-38

%SS_SELECT – Wait for data 16-40

%SS_SEND – Send bytes to a connected peer or established peer address 16-41

%SS_SENDBUF – Send bytes from a socket buffer to a connected peer..... 16-43

%SS_SENDTO – Send bytes through a datagram socket 16-45

%SS_SOCKET – Return a socket descriptor 16-47

Introduction to the Synergy Socket API

Sockets are the foundation for all TCP/IP computing, and most of today's network computing features have sockets at their core. The Synergy socket API gives you direct access to sockets from your Synergy applications.

With this new portable enabling capability, you can use sockets to help distribute the functionality of your applications across LAN (or inter-network) without having to write C code—which requires significant accommodation on each platform. You can also enable your applications to connect to server programs written in any language, or enable other language applications to access your Synergy application logic.

We provide basic API information only. However, many books are available on the subject, for both UNIX (where sockets originated) and Winsock (the Microsoft implementation), if you'd like to read a full discussion.

System requirements

Your system must be properly configured for TCP/IP network communications. Consult your system administration documentation for information on configuring your system for TCP/IP networking.

Assumptions

The Synergy socket API documentation is for users already familiar with Berkeley Socket programming. Implementation differences are discussed below in the “Differences between Synergy sockets and Berkeley Sockets” section.

Those who are not already familiar with socket programming may need additional reference material.

Differences between Synergy Language sockets and Berkeley Sockets

- ▶ Synergy Language sockets only support IP Internet addresses.
- ▶ All Synergy Language socket functions begin with “SS_”.
- ▶ All Synergy Language socket functions return zero if successful (except %SS_HTONL, %SS_HTONS, %SS_NTOHL, and %SS_NTOHS, which return converted integer values). If successful, all functions return the system-specific error code defined in **synsock.def**.
- ▶ Since all Synergy Language socket functions return the error status, some functions use arguments for additional return values.

- ▶ The *hostent* structure associated with the **gethost*** functions in Berkeley Sockets is hidden within the Synergy socket API. Only the *h_name* and *h_addr* members of this structure are passed to and from the socket API.
The *h_addr_list* member can be used to access all addresses associated with a hostname. Multiple calls to `%SS_GETHOSTBYNAME` can access these addresses. The *h_addr_type* member is always `AF_INET`.
- ▶ All Synergy Language socket functions return variables in host byte order. However, `%SS_HTONL`, `%SS_HTONS`, `%SS_NTOHL`, and `%SS_NTOHS` convert host byte order to network byte order (and back again) when sending and receiving portable data over the network.
- ▶ The *sockaddr_in* structure is hidden within the Synergy socket API. The members *sin_port* and *sin_addr* are the only ones used by Synergy Language sockets. The *sin_family* member is always `AF_INET`.



Unlike conventional C Berkeley Sockets programming, Synergy Language socket Internet addresses and ports are shared among the various functions in host byte order and should not be converted to network byte order.

- ▶ Synergy Language sockets offer blocking socket calls only.
- ▶ Synergy Language sockets do not support out-of-band data.
- ▶ Synergy Language sockets do not implement the following Berkeley Socket functions:

<code>getsockopt</code>	<code>getprotobyname</code>
<code>setsockopt</code>	<code>shutdown</code>
<code>getprotobyname</code>	

(Although you cannot call these functions, `SS_CLOSE` does use `shutdown` before closing the socket, and `SS_SOCKET` sets the `SO_KEEPALIVE` `setsockopt` option for all sockets and the `SO_LINGER` and `TCP_NODELAY` `setsockopt` options for `SS_SOCKET_STREAM` sockets.)

Calling the socket functions

When you use the Synergy socket API, you must `.INCLUDE DBLDIR:synsock.def`, which is included in your distribution. The `synsock.def` file includes all of the socket types, flags, Internet addresses, and error status definitions.

Handling lost sockets

If, while waiting for a packet, the program on the other end of a socket abruptly closes the socket or terminates unexpectedly, %SS_SELECT returns immediately to the calling program without indicating that an error has occurred, which makes it appear that a message has arrived.

%SS_RECV using SS_MSG_PEEK also returns without indicating an error condition but it does *not* modify the buffer (**peek_length**) with the length of the message.

The **peek_length** variable (which is an **i4** integer variable specified as the *buf* argument in %SS_RECV) is set to 0. When %SS_RECV specifies SS_MSG_PEEK, **peek_length** is filled with the length of the pending message. You can test to see if **peek_length** has been changed by using %SS_RECV with SS_MSG_PEEK to get the incoming message length. %SS_RECV returns immediately without indicating an error when the socket has been lost, but it does *not* update the **peek_length**. Therefore, we recommend that you both test the return status and check for a **peek_length** with a value of 0 before using %SS_RECVBUF to attempt to retrieve the message. If the socket is lost during %SS_RECVBUF, the appropriate error is returned.

The %SS_RECVBUF function loops on the RECV until the entire packet has been received. It is conceivable, although highly unlikely, that an infinite loop may result if the length in the packet is longer than the actual packet or if the packet has been corrupted so that this appears to be the case.

Synergy socket function errors

Two types of errors are possible in the Synergy socket API: *signaled* and *returned*.

Signaled errors are caused by improper calls to socket functions. They are severe errors that stop the runtime and can be signaled for any of the following reasons:

- ▶ Missing or incorrect number of subroutine arguments
- ▶ Passing literals when a subroutine expects writable arguments
- ▶ Passing values that are invalid or out of range

Returned errors are those errors returned as a value from a Synergy Language socket function. Unlike signaled errors, returned errors are not caused by improper syntax in function calls; they reflect the state of the socket subsystem. Returned errors should be trapped and handled appropriately. Some of the causes of returned errors are as follows:

- ▶ Socket is not valid.
- ▶ Socket is not connected.
- ▶ Socket is already connected.
- ▶ Network subsystem failed.
- ▶ Network is unreachable.
- ▶ Connection timed out.

All of the Synergy Language socket functions return error values, except for the data-conversion socket functions `%SS_NTOHL`, `%SS_NTOHS`, `%SS_HTONL`, and `%SS_HTONS`.

A list of error codes returned by each function is shown in the error code matrix that begins on [page 16-7](#). All errors marked with an X can be returned by the particular function. Those marked with an * can also be returned by the function but have a different description depending on the function. Supplemental descriptions are included on the following pages.

The error code matrix was compiled from system documentation for Windows, OpenVMS, and more than eight varieties of UNIX and contains the socket errors from **synsock.def**. Any errors that are not mapped in the error code matrix are mapped to the `SS_EUNKNOWN` error code. Some error codes in **synsock.def** aren't listed in the matrix because they were not documented as being used on the various platforms.



Not all platforms return every error code listed in the error code matrix. However, we recommend that you handle all errors for each function to ensure that your application is portable to other platforms. For example, `SS_NOTINITIALISED` is generally only returned on Windows clients. Even though you might be developing on UNIX or OpenVMS, it is useful to handle the error so your application can later be ported to Windows.

Since the Synergy Language socket error codes use proprietary numeric values, mapped from the system-level error codes, a mechanism is provided for accessing the system-level error values. This can be done by calling `%SYSERR` right after calling the particular socket function. Calling `%SYSERR` is especially useful when a socket function returns the error value of `SS_EUNKNOWN`; this way you can look up the system error code (on the particular system) to find out what caused the problem.

UNIX

If you use `FORK` or `SPAWN` in a program that uses sockets, you should expect to retry the current operation (for example, `RECV`, `ACCEPT`, etc.) with an `SS_` function that returns `SS_EINTR`. This is normal UNIX behavior when a child program exits.

Error code matrix

- X Error returned by the function
- * Error returned by the function with a supplemental description as documented in the function's description
- Error not returned by this function

Error value	Description	ACCEPT	BIND	CLOSE	CONNECT	FATAL	GETHOSTBYADDR	GETHOSTBYNAME	GETHOSTNAME	GETPEERNAME	GETSOCKNAME	GETSERVBYNAME	GETSERVBYPORT	INET_ADDR	INET_NTOA	LISTEN	RECV	RECVBUF	RECVFROM	SELECT	SEND	SENDBUF	SENDTO	SOCKET
SS_EACCES	See function documentation.	–	*	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	*	*	*	–
SS_EADDRINUSE	Internet address or port is in use.	–	X	–	X	–	–	–	–	–	–	–	–	–	–	X	–	–	–	–	–	–	–	–
SS_EADDRNOTVAIL	Address is not available from the local machine.	–	X	–	X	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	X	–	
SS_EBADF	Socket parameter is not valid.	X	X	X	X	X	–	–	–	X	X	–	–	–	–	X	X	X	X	X	X	X	X	–
SS_EBADSTRING	Bad character string provided.	–	–	–	–	–	–	–	–	–	–	–	–	X	–	–	–	–	–	–	–	–	–	–
SS_ECONNABORTED	The virtual circuit was aborted due to time-out or other failure.	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	X	X	X	–	X	X	X	–
SS_ECONNREFUSD	The attempt to connect was rejected.	–	–	–	X	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
SS_ECONNRESET	The virtual circuit was reset by the remote side.	–	–	–	*	–	–	–	–	–	–	–	–	–	–	–	X	X	X	–	X	X	X	–
SS_EDESTADDRRQ	See function documentation.	–	–	–	*	–	–	–	–	–	–	–	–	–	–	*	–	–	–	–	*	*	*	–
SS_EHOSTDOWN	Networking subsystem is not up.	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	X

Synergy Socket API

Introduction to the Synergy Socket API

Error value	Description	ACCEPT	BIND	CLOSE	CONNECT	FATAL	GETHOSTBYADDR	GETHOSTBYNAME	GETHOSTNAME	GETPEERNAME	GETSOCKNAME	GETSERVBYNAME	GETSERVBYPORT	INET_ADDR	INET_NTOA	LISTEN	RECV	RECVBUF	RECVFROM	SELECT	SEND	SENDBUF	SENDTO	SOCKET
SS_EINTR	Call was interrupted by a signal.	X	-	X	X	-	-	-	-	-	-	X	X	-	-	-	X	X	X	X	X	X	X	-
SS_EINVAL	See function documentation.	*	*	-	-	-	-	-	-	-	*	-	-	-	-	*	*	*	-	-	-	-	-	-
SS_EISCONN	Socket is already connected.	-	X	-	X	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	X	-
SS_EMFILE	No more descriptors available.	X	-	-	X	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	-	X
SS_MSGSIZE	Buffer too small to receive entire datagram, or buffer containing datagram is too large for sending.	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	X	-	X	-	X	-
SS_ENETDOWN	Network subsystem has failed.	X	X	X	X	-	-	-	-	X	-	X	X	-	-	X	X	X	X	X	X	X	X	X
SS_ENETRESET	Connection must be reset.	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X	X	X	-
SS_ENETUNREACH	The network is unreachable.	-	-	X	X	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-
SS_ENFILE	System's table of open files is full.	X	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X
SS_ENOBUFS	No buffer space is available.	X	X	-	X	-	-	-	-	X	X	-	-	-	-	X	X	X	X	X	X	X	X	X
SS_ENOMEM	Insufficient memory for operation.	X	X	-	-	-	-	-	-	X	X	-	-	-	-	-	-	-	-	-	-	-	-	-
SS_NOTSOCK	Socket descriptor is not a socket.	X	X	X	X	X	-	-	-	X	X	-	-	-	-	X	X	X	X	X	X	X	X	-
SS_NOTCONN	Socket is not connected.	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	X	X	X	X	X	X	X	-

Error value	Description	ACCEPT	BIND	CLOSE	CONNECT	FATAL	GETHOSTBYADDR	GETHOSTBYNAME	GETHOSTNAME	GETPEERNAME	GETSOCKNAME	GETSERVBYNAME	GETSERVBYPORT	INET_ADDR	INET_NTOA	LISTEN	RECV	RECVBUF	RECVFROM	SELECT	SEND	SENDBUF	SENDTO	SOCKET
SS_ENULL	Call failed. No variables loaded.	-	-	-	-	-	X	X	X	-	-	X	X	-	-	-	-	-	-	-	-	-	-	-
SS_EOPNOTSUPP	Socket is not of a type that supports this function call.	X	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-
SS_ETIMEDOUT	Connection timed out.	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-
SS_EUNKNOWN	Unknown error returned. Use %SYSERR to obtain system error.	X	X	X	X	-	-	-	-	X	X	-	-	-	-	X	X	X	X	X	X	X	X	X
SS_HOST_NOT_FOUND	Authoritative answer host not found.	-	-	-	-	-	-	-	-	-	-	X	X	-	-	-	-	-	-	-	-	-	-	-
SS_NO_DATA	Valid name, no data record of requested type.	-	-	-	-	-	-	-	-	-	-	X	X	-	-	-	-	-	-	-	-	-	-	-
SS_NO_RECOVERY	Nonrecoverable errors occurred.	-	-	-	-	-	-	-	-	-	-	X	X	-	-	-	-	-	-	-	-	-	-	-
SS_NOTINITIALISED Note: WSAStartup() is automatically called in the Synergy Language runtime by the %ss_socket, %ss_gethostbyname, %gethostbyaddr, %gethostname, and Synergy/DE™/Server functions.	A successful call to WSAStartup() must occur before using this function (Windows only).	X	X	X	X	-	-	-	-	X	X	-	-	-	-	X	X	X	X	X	X	X	X	X
SS_TRY_AGAIN	Nonauthoritative host not found, or SERVERFAIL.	-	-	-	-	-	-	-	-	-	-	X	X	-	-	-	-	-	-	-	-	-	-	-

Synergy socket programming examples

An example SMTP (Simple Mail Transport Protocol) e-mail sender is included in the db\examples directory of your Synergy Language distribution.

The following is a basic TCP client/server programming example. The client connects to a server and sends it a message string. The server receives this message string and prints it out to the console. Since this message string is variable length, this program uses the SS_MSG_PEEK option of %SS_RECV.

```

;-----
;           Synergy Socket API - TCP Client Example
;
;  Module:  client.dbl
;
;-----

#include "DBLDIR:synsock.def"

#define SERVERPORT ,1234
#define TTCHN      ,100

record
    err          ,i4
    socket        ,D_ADDR
    server_in_addr ,i4
    server_name   ,a40

    group packet
        length    ,i4
        message    ,a256
    endgroup

proc
    xcall flags(7004020,1)
    open(TTCHN, o, "tt:")

;Create the socket for the client
    err = %ss_socket(socket, SS SOCK_STREAM)
    if (err .ne. SS_SUCCESS)
        goto unexpected_failure

;Get the server name to connect to
    writes(TTCHN, "Enter server to connect to:")
    reads(TTCHN, server_name)

```

```
;Get IP address for the server we're connecting to
err = %ss_gethostbyname(server_name, server_in_addr)
if (err .ne. SS_SUCCESS)
    goto unexpected_failure

;Connect to the server
err = %ss_connect(socket, SERVERPORT, server_in_addr)
if (err .ne. SS_SUCCESS)
    goto unexpected_failure

;Get message to send to server
writes(TTCHN, "Enter message to send to server:")
reads(TTCHN, packet.message)

;Length of packet to send including packet length field
; itself converted to network byte order for portability
packet.length = %ss_htonl(4 + %trim(packet.message))

;Send the packet
err = %ss_sendbuf(socket, packet(1:%ss_ntohl(packet.length)))
if (err .ne. SS_SUCCESS)
    goto unexpected_failure

;Close the socket
err = %ss_close(socket)
if (err .ne. SS_SUCCESS)
    goto unexpected_failure

goto program_shutdown

unexpected_failure,
    writes(TTCHN, "Unexpected failure in call to socket function.")
    writes(TTCHN, "Error code returned was: " + %string(err))
    writes(TTCHN, "Program terminated.")

program_shutdown,
    close(TTCHN)
    stop
end
```

Synergy Socket API

Introduction to the Synergy Socket API

```
;*-----  
;  
;          Synergy Socket API - TCP Server Example  
;  
; Module:  server.dbl  
;  
;*-----  
  
.include "DBLDIR:synsock.def"  
  
.define SERVERPORT ,1234  
.define TTCHN      ,100  
  
record  
    err                ,i4  
    socket              ,D_ADDR  
    client_socket      ,D_ADDR  
    bytes_received     ,i4  
    packet_length      ,i4  
    packet_length_peek ,i4  
  
    group packet  
        length        ,i4  
        message        ,a256  
    endgroup  
  
proc  
    xcall flags(7004020,1)  
    open(TTCHN, o, "tt:")  
  
;Create the server's socket  
    err = %ss_socket(socket, SS SOCK_STREAM)  
    if (err .ne. SS_SUCCESS)  
        goto unexpected_failure  
  
;Bind server port and use any network interface in host  
    err = %ss_bind(socket, SERVERPORT, SS_INADDR_ANY)  
    if (err .ne. SS_SUCCESS)  
        goto unexpected_failure  
  
;Listen for connections  
    err = %ss_listen(socket)  
    if (err .ne. SS_SUCCESS)  
        goto unexpected_failure  
    writes(TTCHN, "Server is ready to accept a connection from a  
&         client...")  
    writes(TTCHN, " ")
```

```
;Accept a connection from a client
err = %ss_accept(socket, client_socket)
if (err .ne. SS_SUCCESS)
    goto unexpected_failure

;Peek into packet to find out the packet length
err = %ss_recv(client_socket, ^A(packet_length_peek), bytes_received,
&
    SS_MSG_PEEK)
if (err .ne. SS_SUCCESS)
    goto unexpected_failure

;Receive packet based on the packet length we obtained with the peek
err = %ss_recvbuf(client_socket,
&
    packet(1:%ss_ntohl(packet_length_peek)))
if (err .ne. SS_SUCCESS)
    goto unexpected_failure

;Close the client's socket descriptor
err = %ss_close(client_socket)
if (err .ne. SS_SUCCESS)
    goto unexpected_failure

;Close the server's socket descriptor
err = %ss_close(socket)
if (err .ne. SS_SUCCESS)
    goto unexpected_failure

;Print out the client's message
writes(TTCHN, "Client's message is as follows:")
writes(TTCHN, packet.message(1:%ss_ntohl(packet.length)-4))

goto program_shutdown

unexpected_failure,
    writes(TTCHN, "Unexpected failure in call to socket function.")
    writes(TTCHN, "Error code returned was: " + %string(err))
    writes(TTCHN, "Program terminated.")

program_shutdown,
    close(TTCHN)
    stop
end
```

Synergy Socket API Routines

The Synergy socket API routines begin with %SS_.

%SS_ACCEPT – Accept a connection on a stream socket

status = %SS_ACCEPT (*socket*, *peer_socket*[, *in_port*, *in_addr*])

Return value

status

If successful, %SS_ACCEPT loads *peer_socket* with the new socket descriptor and returns SS_SUCCESS (0). If unsuccessful, %SS_ACCEPT returns one of the error codes listed in the error code matrix that begins on [page 16-7](#). (i4)

Arguments

socket

Socket created by %SS_SOCKET, bound to an address with %SS_BIND, and that has successfully issued a call to %SS_LISTEN. (i4)

peer_socket

Returned with the peer socket descriptor. (i4)

in_port

(optional) Returned with the Internet port of the connecting entity. (i4)

in_addr

(optional) Returned with the Internet address of the connecting entity. (i4)



If either *in_port* or *in_addr* is passed, they must both be passed.

Discussion

%SS_ACCEPT accepts a connection on a stream socket. It is used by stream socket server applications only.

%SS_ACCEPT extracts the first connection on the pending queue, creates a new socket with the same properties as *socket*, and returns the new socket descriptor in *peer_socket*.

If the listen queue has no connection requests, %SS_ACCEPT blocks until a connection is present.

If *in_port* and *in_addr* are passed, %SS_ACCEPT loads them with the Internet port and Internet address of the connecting entity.

Supplemental error code descriptions

For this function, the supplemental error code descriptions are

SS_EINVAL The socket is not accepting connections.

%SS_BIND – Bind a name to an unbound socket

status = %SS_BIND(*socket*, *in_port*, *in_addr*)

Return value

status

If successful, %SS_BIND returns SS_SUCCESS (0). If unsuccessful, %SS_BIND returns one of the error codes listed in the error code matrix that begins on [page 16-7](#). (n)

Arguments

socket

A descriptor of an open socket created by %SS_SOCKET. (i4)

in_port

The local Internet port to associate with the socket. (i4)

in_addr

The local Internet address to associate with the socket. (i4)

Discussion

%SS_BIND binds a name to an unbound socket. *In_port* and *in_addr* are used to name it.

Specify SS_INADDR_ANY for *in_addr* unless you wish to specify a particular Internet address to associate with the socket. If *in_addr* is set to SS_INADDR_ANY, any appropriate network interface is used for sending or receiving data. This simplifies programming in the presence of multihomed hosts.

If *in_port* is zero, a random port greater than 1023 is assigned. (Clients typically specify zero, while servers specify well-known ports.)



It is unnecessary for a connection-oriented client to call %SS_BIND for a socket if it wishes to use any address and a random port, because %SS_CONNECT automatically performs this type of bind.

Supplemental error code descriptions

For this function, the supplemental error code descriptions are

SS_EACCES	The requested address is protected, and the current user has inadequate permission to access it.
SS_EINVAL	The socket is already bound to an address.

%SS_CLOSE – Close a socket

status = %SS_CLOSE(*socket* [, SS_PURGE])

Return value

status

If successful, %SS_CLOSE returns SS_SUCCESS (0). If unsuccessful, %SS_CLOSE returns one of the error codes listed in the error code matrix that begins on [page 16-7](#). (n)

Arguments

socket

A socket descriptor. (i4)

UNIX

SS_PURGE

(optional) Do not apply the socket shutdown logic.

Discussion

%SS_CLOSE closes a connection and deletes a socket descriptor.

UNIX

You may want to specify SS_PURGE when using the FORK subroutine on a process to ensure that the FORK'd process does not force the connection to be terminated for the parent process when it closes the socket.

The accepted socket (with %SS_ACCEPT) on the server needs to be purged after (and only after) it knows the client has cloned the socket and is using it, because the child now owns the socket. The server socket should be purged in the child program.

%SS_CONNECT – Connect sockets or establish peer address

status = %SS_CONNECT(*socket*, *in_port*, *in_addr*)

Return value

status

If successful, %SS_CONNECT returns a SS_SUCCESS (0). If unsuccessful, %SS_CONNECT returns one of the error codes listed in the error code matrix that begins on [page 16-7](#). (n)

Arguments

socket

A descriptor of an open socket created by %SS_SOCKET. (i4)

in_port

The Internet port of a target socket. (i4)

in_addr

The Internet address of a target socket. (i4)

Discussion

%SS_CONNECT connects two stream sockets or establishes a peer address for a datagram socket.

If *socket* is a stream socket, %SS_CONNECT attempts to make a connection to the target stream socket specified by *in_port* and *in_addr*. A stream socket client does not have to call %SS_BIND to bind a local address before calling %SS_CONNECT. %SS_CONNECT automatically binds a random TCP port greater than 1023 to the socket before making a connection.

If *socket* is a datagram socket, %SS_CONNECT specifies the peer where datagrams will be sent or received on subsequent %SS_SEND and %SS_RECV calls.

Supplemental error code descriptions

For this function, the supplemental error code descriptions are

SS_EDESTADDRREQ A destination address is required.

SS_ECONNRESET The remote peer has accepted and then aborted the connection before the connect call could complete.

%SS_FATAL – Notify server that program is terminating abnormally

status = %SS_FATAL (*socket*, *message*)

Return value

status

If successful, %SS_FATAL returns SS_SUCCESS (0). If unsuccessful, %SS_FATAL returns one of the error codes listed in the error code matrix that begins on [page 16-7](#). (**n**)

Arguments

socket

A descriptor of an open socket created by %SS_SOCKET. (**i4**)

message

A string to send through the socket if the program ends before the socket is properly closed. (**a**)

Discussion

%SS_FATAL provides the ability to send a final error packet to the other end of a socket if the program ends before the socket is properly closed. Thus, if a broken socket occurs, the other end of the socket doesn't have to continue to wait indefinitely.

Message can contain two %s and three %d specifiers, as follows:

"pre-text %s %s %d %d %d post-text"

The %s and %d specifiers can occur anywhere in the string and be delimited by any or no characters, but they must be specified in the order shown above. The %s and %d specifiers are replaced, respectively, with the program name, the module in which the program terminated, the line number where the program terminated, the number of the Synergy Language error that terminated the program, and any system error number.

You don't have to use all five specifiers, but if you don't use them all, you can only truncate the string from the right. For example, "%s/%s/%d/%d", "%s/%s/%d", "%s/%s", and "%s" are all valid, but "%s/%d" is not valid. If these specifiers are not specified in the proper order, or if the string contains other % specifiers, an error is generated.

%SS_GETHOSTBYADDR – Get the host name corresponding to an Internet address

status = %SS_GETHOSTBYADDR(*in_addr*; *h_name*)

Return value

status

If successful, %SS_GETHOSTBYADDR returns SS_SUCCESS (0). If unsuccessful, %SS_GETHOSTBYADDR returns SS_ENULL. (**n**)

Arguments

in_addr

Specifies an Internet address. (**i4**)

h_name

Returned with the host name. (**a**)

Discussion

%SS_GETHOSTBYADDR retrieves a host name using the host address *in_addr* as a search parameter.

%SS_GETHOSTBYNAME – Get the Internet address corresponding to a host name

status = %SS_GETHOSTBYNAME(*h_name*, *in_addr*[], *pos*)

Return value

status

If successful, %SS_GETHOSTBYNAME returns SS_SUCCESS (0). If unsuccessful, %SS_GETHOSTBYNAME returns SS_ENULL. **(n)**

Arguments

h_name

Specifies the host name. **(a)**

in_addr

Returned with the Internet address. **(i4)**

pos

(optional) Specifies the host address. The default value is 1 (the primary address). **(n)**

Discussion

%SS_GETHOSTBYNAME retrieves an Internet address using the host name in *h_name* as a search parameter.

A position, *pos*, can be specified to retrieve an address other than the primary address. If *pos* is greater than the number of addresses present, %SS_GETHOSTBYNAME returns SS_ENULL.

When using this option, %SS_GETHOSTBYNAME is usually called in a loop, incrementing *pos* until it returns SS_ENULL.



Do not use %SS_GETHOSTBYNAME to resolve dotted IP address strings. An application with a dotted IP address to resolve should use %SS_INET_ADDR to convert the string to an IP address, and then %SS_GETHOSTBYADDR to obtain the host name.

%SS_GETHOSTNAME – Get the name of the local host

status = %SS_GETHOSTNAME(*h_name*)

Return value

status

If successful, %SS_GETHOSTNAME returns SS_SUCCESS (0). If unsuccessful, %SS_GETHOSTNAME returns SS_ENULL. (**n**)

Arguments

h_name

Returned with the host name. (**a**)

Discussion

%SS_GETHOSTNAME returns the local host name in *h_name*. The form of *h_name* is dependent on the system and its particular implementation. For example, a simple host name might be **socrates**, and a fully qualified domain name might be **socrates.synergex.com**.

%SS_GETHOSTNAME returns either form of host name.

See also

[“Introduction to the Synergy Socket API” on page 16-3.](#)

%SS_GETPEERNAME – Return the name of the peer socket

status = %SS_GETPEERNAME(*socket*, *in_port*, *in_addr*)

Return value

status

If successful, %SS_GETPEERNAME returns SS_SUCCESS (0). If unsuccessful, %SS_GETPEERNAME returns one of the error codes listed in the error code matrix that begins on [page 16-7](#). (n)

Arguments

socket

A descriptor of an open socket created by %SS_SOCKET. (i4)

in_port

Returned with the Internet port of the peer. (i4)

in_addr

Returned with the Internet address of the peer. (i4)

Discussion

%SS_GETPEERNAME returns the name of the peer for a connected stream socket or a datagram socket with an established peer address.

%SS_GETSERVBYNAME – Return the port number given the service name

status = %SS_GETSERVBYNAME (*service*, [*type*], *port*, *protocol*)

Return value

status

If successful, %SS_GETSERVBYNAME returns SS_SUCCESS (0). If unsuccessful, %SS_GETSERVBYNAME returns SS_ENULL. (**n**)

Arguments

service

The service name for which to return a port number. (**a**)

type

(optional) One of the following types of protocols to access the service: (**a**)

tcp Transmission control protocol.

udp User datagram protocol.

port

Returned with the port number for the specified service. (**n**)

protocol

Returned with the name of the network protocol for the specified service. (**a**)

Discussion

%SS_GETSERVBYNAME gets the port and network protocol from the **/etc/services** file for the specified service name (and optional protocol type). If *type* is not specified, the first match is returned, and *protocol* will be either “tcp” or “udp.”

Examples

The example below returns the port and protocol for the **vtxnet** service.

```
status = %ss_getservbyname("vtxnet", , port, proto)
```

%SS_GETSERVBYPOR – Return the service name given the port number

status = %SS_GETSERVBYPOR (*port*, [*type*], *service*, *protocol*)

Return value

status

If successful, %SS_GETSERVBYPOR returns SS_SUCCESS (0). If unsuccessful, %SS_GETSERVBYPOR returns SS_ENULL. (**n**)

Arguments

port

The port number for which to return a service name. (**n**)

type

(optional) One of the following types of protocols to access the service: (**a**)

tcp Transmission control protocol.

udp User datagram protocol.

service

Returned with the name of the service on the specified port. (**a**)

protocol

Returned with the name of the network protocol on the specified port. (**a**)

Discussion

%SS_GETSERVBYPOR gets the service name and network protocol from the **/etc/services** file for the specified port number (and optional protocol type). If *type* is not specified, the first match is returned, and *protocol* will be either “tcp” or “udp.”

Examples

The example below returns the service name and protocol for port 1.

```
status = %ss_getservbyport(1, tcp, name, proto)
```

%SS_GETSOCKNAME – Return the name associated with a socket

status = %SS_GETSOCKNAME (*socket*, *in_port*, *in_addr*)

Return value

status

If successful, %SS_GETSOCKNAME returns SS_SUCCESS (0). If unsuccessful, it returns one of the error codes listed in the error code matrix that begins on [page 16-7](#). (**n**)

Arguments

socket

A descriptor of an open socket created by %SS_SOCKET. (**i4**)

in_port

Returned with the Internet port of the socket. (**i4**)

in_addr

Returned with the Internet address of the socket. (**i4**)

Discussion

%SS_GETSOCKNAME returns the name of a socket, which is the local association. This function is especially useful when an %SS_CONNECT call has been made without doing an %SS_BIND first; this call provides the only method of determining the local association set by the system.

If a socket was bound with SS_INADDR_ANY, indicating that any of the host's IP addresses should be used for the socket, %SS_GETSOCKNAME does not necessarily return information about the host IP address, unless there has been a call to %SS_CONNECT or %SS_ACCEPT. This is because on a multihomed host, the IP address that will be used for the socket is unknown unless the socket is connected.

Supplemental error code descriptions

For this function, the supplemental error code descriptions are

SS_EINVAL	The socket has not been bound to an address with %SS_BIND or the socket has been shut down.

%SS_HTONL – Convert long integers from host to network byte order

integer = %SS_HTONL (*hostlong*)

Return value

integer

A long integer in network byte order. (**i4**)

Arguments

hostlong

A long integer in host byte order. (**i4**)

Discussion

%SS_HTONL converts 32-bit (long) unsigned integers from host byte order (native endian type) to network byte order.

Host byte order is the format in which bytes are ordered internally on a specific host. Network byte order is the format in which data bytes are supposed to be transmitted through a network. All hosts on a network must send data in network byte order. If your host does not have an internal data representation format identical to the network byte order, you must use %SS_HTONL.



In the Synergy socket API, all Internet address and port arguments (*in_addr* and *in_port*) are in host byte order, and should not be converted to network byte order. The %SS_HTONL function should only be used when sending portable data over the network.

%SS_HTONS – Convert short integers from host to network byte order

integer = %SS_HTONS (*hostshort*)

Return value

integer

A short integer in network byte order. (**i2**)

Arguments

hostshort

A short integer in host byte order. (**i2**)

Discussion

%SS_HTONS converts 16-bit (short) unsigned integers from host byte order to network byte order.

Host byte order is the format in which bytes are ordered internally on a specific host. Network byte order is the format in which data bytes are supposed to be transmitted through a network. All hosts on a network must send data in network byte order. If your host does not have an internal data representation format identical to the network byte order, you must use %SS_HTONL.



In the Synergy socket API, all Internet address and port arguments (*in_addr* and *in_port*) are in host byte order, and should not be converted to network byte order. The %SS_HTONS function should only be used when sending portable data over the network.

%SS_INET_ADDR – Convert a dotted text form Internet address

status = %SS_INET_ADDR(*charstring*, *in_addr*)

Return value

status

If *charstring* is valid, %SS_INET_ADDR returns SS_SUCCESS (0). If *charstring* is invalid, %SS_INET_ADDR returns SS_EBADSTRING. (**n**)

Arguments

charstring

An Internet address in dotted text form. (**a**)

in_addr

Returned with the Internet address in numeric form. (**i4**)

Discussion

%SS_INET_ADDR converts an ASCII string containing a valid Internet address using dot notation into a numeric Internet address.

An example of a valid Internet address in dot notation is 204.33.191.3.

%SS_INET_NTOA – Convert a numeric Internet address

status = %SS_INET_NTOA (*in_addr*, *charstring*)

Return value

status

If successful, %SS_INET_NTOA returns SS_SUCCESS (0). (**n**)

Arguments

in_addr

An Internet address in numeric form. (**i4**)

charstring

Returned with the Internet address in dotted text form. (**a**)

Discussion

%SS_INET_NTOA translates a numeric Internet address into a dot notation Internet address.

An example of a valid Internet address in dot notation is 204.33.191.3

%SS_LISTEN – Listen for connections

status = %SS_LISTEN(*socket*[, *backlog*])

Return value

status

If successful, %SS_LISTEN returns SS_SUCCESS (0). If unsuccessful, it returns one of the error codes listed in the error code matrix that begins on [page 16-7](#). (n)

Arguments

socket

A descriptor of an open socket created by %SS_SOCKET. (i4)

backlog

(optional) Specifies the maximum number of pending connections (1 to 5) that may be queued on the stream socket at any given time. (n)

Discussion

%SS_LISTEN creates a queue for pending connection requests on socket. Connections can then be accepted with %SS_ACCEPT. If a connection request arrives with the queue full, the client receives a time-out.

If *backlog* is not passed, the default value is 5.

%SS_LISTEN is used by stream socket server applications only.

Supplemental error code descriptions

For this function, the supplemental error code descriptions are

SS_EINVAL	The socket has not been bound with %SS_BIND or is already connected.
SS_EDESTADDRREQ	The socket has not been bound to an address using %SS_BIND.

%SS_NTOHL – Convert long integers from network to host byte order

integer = %SS_NTOHL (*netlong*)

Return value

integer

A long integer in host byte order. **(i4)**

Arguments

netlong

A long integer in network byte order. **(i4)**

Discussion

%SS_NTOHL converts 32-bit (long) unsigned integers from network byte order to host byte order.

Host byte order is the format in which bytes are ordered internally on a specific host. Network byte order is the format in which data bytes are supposed to be transmitted through a network. All hosts on a network must send data in network byte order. If your host does not have an internal data representation format identical to the network byte order, you must use %SS_HTONL.



In the Synergy socket API, all Internet address and port arguments (*in_addr* and *in_port*) are in host byte order, and should not be converted to network byte order. The %SS_NTOHL function should only be used when sending portable data over the network.

%SS_NTOHS – Convert short integers from network to host byte order

integer = %SS_NTOHS (*netshort*)

Return value

integer

A short integer in host byte order. (**i2**)

Arguments

netshort

A short integer in host byte order. (**i2**)

Discussion

%SS_NTOHS converts 16-bit unsigned (short) integers from network byte order to host byte order.

Host byte order is the format in which bytes are ordered internally on a specific host. Network byte order is the format in which data bytes are supposed to be transmitted through a network. All hosts on a network must send data in network byte order. If your host does not have an internal data representation format identical to the network byte order, you must use %SS_HTONL.



In the Synergy socket API, all Internet address and port arguments (*in_addr* and *in_port*) are in host byte order, and should not be converted to network byte order. The %SS_NTOHS function should only be used when sending portable data over the network.

%SS_RECV – Receive bytes from a socket

status = %SS_RECV (*socket*, *buf*, *bytes_received*[, SS_MSG_PEEK])

Return value

status

If successful, %SS_RECV returns SS_SUCCESS (0). If unsuccessful, %SS_RECV returns one of the error codes listed in the error code matrix that begins on [page 16-7](#). (n)

Arguments

socket

A descriptor of an open socket created by %SS_SOCKET. (i4)

buf

Buffer returned with the incoming data. (a)

bytes_received

Returned with the number of bytes received and placed into *buf*. (i4)

SS_MSG_PEEK

(optional) Peek into the data that is next in line to be received without removing it from the system's buffers. (n)

Discussion

%SS_RECV reads incoming data from a connected stream socket or from a datagram socket with an established peer address.

To receive data from a datagram socket without an established peer address, use %SS_RECVFROM.

If no incoming data is available at the socket, %SS_RECV blocks until data arrives.

Connected stream sockets

%SS_RECV returns all available data up to, but not exceeding, the size of *buf*. This implies that %SS_RECV should be called in a loop so that all bytes sent from the peer stream socket are received. Therefore, we recommend you use %SS_RECVBUF, which performs the loop until all the data has been received.

Unlike %SS_RECVBUF, however, %SS_RECV can be used to peek into the data to be received without actually removing that data from the system's buffers. To do this, pass the optional SS_MSG_PEEK flag.

Datagram socket with an established peer address

%SS_RECV extracts data from the first queued datagram, up to the size of *buf*. If the datagram is larger than the size of *buf*, the overflow data is lost, and %SS_RECV returns the error SS_EMSGSIZE.

Supplemental error code descriptions

For this function, the supplemental error code descriptions are

SS_EINVAL The socket has not been bound with %SS_BIND.

%SS_RECVBUF – Receive bytes from a socket into a buffer

status = %SS_RECVBUF(*socket*, *buf*)

Return value

status

If successful, %SS_RECVBUF returns SS_SUCCESS (0). If unsuccessful, it returns one of the error codes listed in the error code matrix that begins on [page 16-7](#). (n)

Arguments

socket

A descriptor of an open socket created by %SS_SOCKET. (i4)

buf

Buffer returned with the incoming data. (a)

Discussion

%SS_RECVBUF is used for reading an exact amount of incoming data from a connected stream socket. %SS_RECVBUF offers better execution performance than calling the %SS_RECV function in a loop while serving the same purpose.

If no incoming data is available at the socket, or if %SS_RECVBUF hasn't finished reading in all the incoming data, %SS_RECVBUF blocks (even if the remote side disconnects).

%SS_SENDBUF is the complimentary function used in a peer Synergy Language sockets application.

The following source code for an %SS_SENDBUF() function in a C language socket application acts as a complimentary function to %SS_RECVBUF:

```
int ss_sendbuf(int socket, char *buf, int len)
{
    register int i, cnt;

    for (cnt = len; cnt > 0; cnt -= i, buf += i)
    {
        if ((i = send(socket, buf, cnt, 0)) < 0)
            return (-1);
    }

    return (len);
}
```

Supplemental error code descriptions

For this function, the supplemental error code descriptions are

`SS_EINVAL` The socket has not been bound with `%SS_BIND`.

%SS_RECVFROM – Receive bytes from a datagram socket

status = %SS_RECVFROM(*socket*, *buf*, *bytes_received*, [*in_port*, *in_addr*][, SS_MSG_PEEK])

Return value

status

If successful, %SS_RECVFROM returns SS_SUCCESS (0). If unsuccessful, %SS_RECVFROM returns one of the error codes listed in the error code matrix that begins on [page 16-7](#). (n)

Arguments

socket

A descriptor of an open socket created by %SS_SOCKET. (i4)

buf

Buffer returned with the incoming data. (a)

bytes_received

Returned with the number of bytes received and placed into *buf*. (i4)

in_port

(optional) The Internet port of the sending socket. (i4)

in_addr

(optional) The Internet address of the sending socket. (i4)

SS_MSG_PEEK

(optional) Peek into the data that is next in line to be received without removing it from the system's buffers. (n)



If either *in_port* or *in_addr* is passed, they must both be passed.

Discussion

%SS_RECVFROM is used for reading incoming data from a bound datagram socket, where there has not been a previous call to %SS_CONNECT. %SS_CONNECT establishes a permanent peer address. %SS_RECVFROM can optionally return the source port and address.

Datagram socket

%SS_RECVFROM extracts data from the first queued datagram up to the size of *buf*. If the datagram is larger than the size of *buf*, the overflow data is lost, and %SS_RECVFROM returns the error SS_MSGSIZE. %SS_RECVFROM enables optional retrieval of the datagram source port and address in *in_port* and *in_addr*.

Connected stream sockets

For stream sockets, %SS_RECVFROM functions identically to %SS_RECV, except that *in_port* and *in_addr* are ignored.

If no incoming data is available at the socket, %SS_RECVFROM blocks until data arrives.

%SS_SELECT – Wait for data

status = %SS_SELECT(*socket*, *seconds*, *microseconds*)

Return value

status

If successful, %SS_SELECT returns SS_SUCCESS (0). If the time expires, it returns SS_ETIMEDOUT. (**n**)

Arguments

socket

A descriptor of an open socket created by %SS_SOCKET. (**i4**)

seconds

The number of seconds to wait for data to arrive on the socket. (**n**)

microseconds

The number of microseconds to wait for data to arrive on the socket. (**n**)

Discussion

Because it is virtually impossible to know when a message is going to be sent or how long it is going to be, you can use %SS_SELECT to wait for data to arrive on the specified socket or for the specified time to expire. If data is already available, %SS_SELECT returns immediately. If the specified time elapses before the message arrives, %SS_SELECT returns control to the calling program so that other computations may be accomplished before checking again to see if the message has arrived.

Seconds and *microseconds* can be used together to indicate traditional time. For example, if you want to wait 10.234 seconds, the *seconds* argument should be 10 and the *microseconds* argument should be 234000.

If a socket is lost while waiting for a packet, %SS_SELECT returns immediately but does not indicate that an error has occurred. (See [“Handling lost sockets” on page 16-5](#) for more information.)

%SS_SEND – Send bytes to a connected peer or established peer address

status = %SS_SEND (*socket*, *buf*, *bytes_sent*)

Return value

status

If successful, %SS_SEND returns SS_SUCCESS (0). If unsuccessful, %SS_SEND returns one of the error codes listed in the error code matrix that begins on [page 16-7](#). (n)

Arguments

socket

A descriptor of an open socket created by %SS_SOCKET. (i4)

buf

Buffer containing the data to be sent. (a)

bytes_sent

Returned with the number of bytes sent from *buf*. (i4)

Discussion

%SS_SEND writes outgoing data to a connected stream socket, or to a datagram socket with an established peer address. To send data from a datagram socket without an established peer address, use %SS_SENDTO.



Successful completion of %SS_SEND does not indicate successful delivery of data.

If no buffer space is available within the transport system to hold the data to be transmitted, %SS_SEND blocks.

Connected stream sockets

An %SS_SEND on a stream socket may output fewer bytes than requested, but this is not considered an error condition. Therefore, you may need to call %SS_SEND in a loop until all the bytes are sent to the peer stream socket. The %SS_SENDBUF function performs this loop for you until all requested bytes have been sent.

Datagram socket with an established peer address

If the quantity of data exceeds the maximum IP packet size of the underlying subnets, no data is transmitted and %SS_SEND returns the SS_EMSGSIZE error.

Supplemental error code descriptions

For this function, the supplemental error code descriptions are

SS_EACCES	The requested address is a broadcast address, but the appropriate flag was not set.
SS_EDESTADDRREQ	The socket has not been bound to an address using %SS_BIND.

%SS_SENDBUF – Send bytes from a socket buffer to a connected peer

status = %SS_SENDBUF (*socket*, *buf*)

Return value

status

If successful, %SS_SENDBUF returns SS_SUCCESS (0). If unsuccessful, it returns one of the error codes listed in the error code matrix that begins on [page 16-7](#). (n)

Arguments

socket

A descriptor of an open socket created by %SS_SOCKET. (i4)

buf

Buffer containing the data to be sent. (a)

Discussion

%SS_SENDBUF writes outgoing data through a connected stream socket in a loop to ensure the entire data packet is delivered. %SS_SENDBUF is equivalent to the %SS_SEND function called in a loop.

If no buffer space is available within the transport system to hold the data to be transmitted, or if %SS_SENDBUF hasn't finished sending all the data, %SS_SENDBUF blocks.

%SS_RECVBUF is the complimentary function used in a peer Synergy Language socket application. The following source code for an %SS_RECVBUF() function in a C language socket application acts as a complimentary function to %SS_SENDBUF:

```
int ss_recvbuf(int socket, char *buf, int len)
{
    int i, cnt;

    for (cnt = len; cnt > 0; cnt -= i, buf += i)
    {
        if ((i = recv(socket, buf, cnt, 0)) < 0)
            return (-1);
    }

    return(len);
}
```

Supplemental error code descriptions

For this function, the supplemental error code descriptions are

SS_EACCES	The requested address is a broadcast address, but the appropriate flag was not set.
SS_EDESTADDRREQ	The socket has not been bound to an address using %SS_BIND.

%SS_SENDTO – Send bytes through a datagram socket

status = %SS_SENDTO (*socket*, *buf*, *bytes_sent*, *in_port*, *in_addr*)

Return value

status

If successful, %SS_SENDTO returns SS_SUCCESS (0). If unsuccessful, %SS_SENDTO returns one of the error codes listed in the error code matrix that begins on [page 16-7](#). (n)

Arguments

socket

A descriptor of an open socket created by %SS_SOCKET. (i4)

buf

Buffer containing the data to be sent. (a)

bytes_sent

Returned with the number of bytes sent from *buf*. (i4)

in_port

Specifies the Internet port of the target socket. (i4)

in_addr

Specifies the Internet address of the target socket. (i4)

Discussion

%SS_SENDTO is used for writing outgoing data through a datagram socket to a specified peer address (*in_port* and *in_addr*) where there has not been a previous call to %SS_CONNECT. (%SS_CONNECT establishes a permanent peer address.)



Successful completion of %SS_SENDTO does not indicate successful delivery of data.

If no buffer space is available within the transport system to hold the data to be transmitted, %SS_SENDTO blocks.

Datagram socket

If the packet data exceeds the maximum IP packet size of the underlying subnets, no data is transmitted and the error SS_MSGSIZE is returned.

Connected stream sockets

For stream sockets, %SS_SENDTO functions identically to %SS_SEND, except that *in_port* and *in_addr* are ignored.

Supplemental error code descriptions

For this function, the supplemental error code descriptions are

SS_EACCES	The requested address is a broadcast address, but the appropriate flag was not set.
SS_EDESTADDRREQ	A destination address is required.

%SS_SOCKET – Return a socket descriptor

status = %SS_SOCKET(*socket*, *type*)

Return value

status

If successful, %SS_SOCKET returns SS_SUCCESS (0). If unsuccessful, %SS_SOCKET returns one of the error codes listed in the error code matrix that begins on [page 16-7](#). (n)

Arguments

socket

Returned with the socket descriptor. (**D_ADDR**)

type

One of the following communication semantics: (n)

SS SOCK_STREAM

SS SOCK_DGRAM

Discussion

%SS_SOCKET creates a new socket descriptor of the specified *type* and returns it in *socket*. This socket descriptor is then used in other %SS_XXX functions.



%SS_SOCKET only supports the AF_INET address family and the default socket protocols.

17

Synergy Windows Printing API

Introduction to the Synergy Windows Printing API 17-2

Discusses the metafile created by the Synergy Windows printing API.

Synergy Windows Printing API Routines 17-4

This API includes the following routines:

%WPR_EXECUTE – Execute printing.....	17-15
%WPR_GETDEVICE – Retrieve printer settings	17-20
%WPR_INFO – Set and retrieve extended device capabilities	17-27
%WPR_PRINT – Perform print operation to selected printer	17-36
%WPR_SETDEVICE – Set printer settings.....	17-42

Introduction to the Synergy Windows Printing API

The Synergy Windows printing API provides access to the Microsoft Windows printing API and device drivers and enables you to use extended printer features in a true Windows environment. To provide features such as file persistence, device independence, and print preview, the API creates a Windows enhanced metafile and plays records into the metafile. The metafile is then played back to a device (the printer or the screen) to be printed or viewed.

Internally, a metafile is an array of variable-length structures called metafile records. The first records in the metafile specify general information such as the resolution of the device on which the picture was created, the dimensions of the picture, and so on. The remaining records, which constitute the bulk of any metafile, correspond to the graphics device interface (GDI) functions required to draw a picture or write out text.

Although a metafile is a collection of structures that can store a picture in a device-independent format, the picture is not a bitmap. For example, when an application creates a picture measuring 2 by 2 inches on a VGA display and stores that picture in a metafile, the picture maintains its original dimensions when printed on a 300-dpi laser printer or copied over a network and displayed in another application running on an 8514/A video display. If the same graphic is stored in a bitmap file and then printed or displayed, it may become distorted because the resolution and/or aspect ratio of the new target device does not match the target of the original image.

When a metafile is created, it is given a “reference device context.” This means that the size of the page (or other device surface if the device is not a printer), resolution, and device capabilities are defined by that device. Playing back a metafile to a device other than the reference device may or may not work, depending on the level of device-specific support expected by the metafile operations. When Synergy plays back a metafile, the output is scaled to the page size of the destination device. If the page sizes don’t match, the output may be distorted.



You can use Adobe® Acrobat® (which requires an end-user license) to print from the Synergy Windows printing API to a PDF file, which is more device-independent than a Windows metafile. A user can then use Adobe Reader® (which is available for free) to view or print the PDF file correctly to any device.

The device independence of the print files depends on which features you use and which features are common among your various printer drivers. You can create a document under one printer, save the file, and print it out to another printer. However, if your file is created using features that are not available to the second printer, the file will not print as formatted. For example, if you create the file using a Times-Roman font and then try to print it to a daisy-wheel Arial font printer, the file will not be printed in Times-Roman. If the file is created using common fonts, point sizes, and attributes (for example, italic or bold), it will print the same on the second printer as on the first.

By default, a metafile is deleted after it is printed or previewed. However, you also have the option to save the metafile so it can be replayed to the printer or screen later. (Refer to the DWP_PRINT subfunction of %WPR_EXECUTE on [page 17-15](#) or to the DWP_DELPRINTER subfunction of %WPR_INFO on [page 17-15](#) for details.)

The default metafile extension for the Windows printing API is **.emf**.

Synergy Windows Printing API Routines

The Synergy Windows printing API routines begin with `%WPR_` and are documented on the pages that follow.

Each of these functions requires you to pass a report handle. To get this report handle, call `%WPR_INFO(report_handle, DWP_GETPRINTER)` before you call any other `%WPR_` function. This function returns a report handle that you can then pass to subsequent `%WPR_` functions.

You must `.INCLUDE DBLDIR:winprint.def` when using the Windows printing API subroutines.



Make sure you initialize your print data structures before using them. If you don't, unpredictable behavior may occur. For example, you can do a "clear `^i(^m(font_handle))`" to set all members to integer 0, and then set the font name, point size, and so forth.

See the first example under "Sample programs" on page 17-7 for the simplest printing routine you can write to replace an `LPQUE` statement. For sample output from the Windows printing API, refer to the *Migrating Your Application to Windows* document, available on the Online Manuals CD.

Recommended calling sequence

The following table describes the recommended and allowable calling sequence for the Windows printing API functions. The Report sequence column lists the calls that are commonly used when building the report, in the order in which they should be called. The other columns show the recommended and allowable subfunctions that you can use after the function call in the Report sequence column.

Report sequence	%WPR_SETDEVICE subfunctions	%WPR_GETDEVICE subfunctions	%WPR_INFO subfunctions	%WPR_PRINT subfunctions	%WPR_EXECUTE subfunctions
%WPR_INFO(..., DWP_GETPRINTER)	DWP_COLLATE DWP_COLOR DWP_COPIES DWP_DUPLEX DWP_FORMNAME DWP_ORIENTATION DWP_PAPERLENGTH DWP_PAPERSIZE DWP_PAPERSOURCE DWP_PAPERWIDTH DWP_QUALITY DWP_SCALE DWP_SPECS	DWP_COLLATE DWP_COLOR DWP_COPIES DWP_DUPLEX DWP_FORMNAME DWP_ORIENTATION DWP_PAPERLENGTH DWP_PAPERSIZE DWP_PAPERSOURCE DWP_PAPERWIDTH DWP_PIXELHEIGHT DWP_PIXELWIDTH DWP_PHYSICALHEIGHT DWP_PHYSICALWIDTH DWP_QUALITY DWP_RESOLUTIONX DWP_RESOLUTIONY DWP_SCALE DWP_SPECS DWP_XOFFSET DWP_YOFFSET	DWP_PRINTDLG	DWP_SETPAGES	
%WPR_PRINT(..., DWP_BEGINJOB	DWP_BACKCOLOR DWP_COLLATE DWP_COLOR DWP_COPIES DWP_DUPLEX DWP_FONT DWP_FORMNAME DWP_ORIENTATION DWP_PAPERLENGTH DWP_PAPERSIZE DWP_PAPERSOURCE DWP_PAPERWIDTH DWP_QUALITY DWP_SCALE DWP_SPECS DWP_TEXTCOLOR	DWP_BACKCOLOR DWP_COLLATE DWP_COLOR DWP_COPIES DWP_DUPLEX DWP_FONT DWP_FORMNAME DWP_ORIENTATION DWP_PAPERLENGTH DWP_PAPERSIZE DWP_PAPERSOURCE DWP_PAPERWIDTH DWP_PIXELHEIGHT DWP_PIXELWIDTH DWP_PHYSICALHEIGHT DWP_PHYSICALWIDTH DWP_QUALITY DWP_RESOLUTIONX DWP_RESOLUTIONY DWP_SCALE DWP_SPECS DWP_TEXTCOLOR DWP_XOFFSET DWP_YOFFSET	DWP_GETPEN DWP_LARGESTFONT DWP_TEXTMETRICS DWP_X_FROM_COL DWP_Y_FROM_ROW	DWP_ADDFILE DWP_SETPAGES	

Synergy Windows Printing API

Synergy Windows Printing API Routines

Report sequence	%WPR_SETDEVICE subfunctions	%WPR_GETDEVICE subfunctions	%WPR_INFO subfunctions	%WPR_PRINT subfunctions	%WPR_EXECUTE subfunctions
%WPR_PRINT(..., DWP_BEGINPAGE	DWP_BACKCOLOR DWP_FONT DWP_TEXTCOLOR	DWP_BACKCOLOR DWP_COLLATE DWP_COLOR DWP_COPIES DWP_DUPLEX DWP_FONT DWP_FORMNAME DWP_ORIENTATION DWP_PAPERLENGTH DWP_PAPERSIZE DWP_PIXELHEIGHT DWP_PIXELWIDTH DWP_PHYSICALHEIGHT DWP_PHYSICALWIDTH DWP_QUALITY DWP_RESOLUTIONX DWP_RESOLUTIONY DWP_SCALE DWP_SPECS DWP_TEXTCOLOR DWP_XOFFSET DWP_YOFFSET	DWP_GETPEN DWP_LARGESTFONT DWP_TEXTMETRICS DWP_X_FROM_COL DWP_Y_FROM_ROW	DWP_BITMAP DWP_LINE DWP_FILL DWP_SETPAGES DWP_WRITEOUT	
%WPR_PRINT(..., DWP_ENDPAGE		DWP_COLLATE DWP_COLOR DWP_COPIES DWP_DUPLEX DWP_FORMNAME DWP_ORIENTATION DWP_PAPERLENGTH DWP_PAPERSIZE DWP_PIXELHEIGHT DWP_PIXELWIDTH DWP_PHYSICALHEIGHT DWP_PHYSICALWIDTH DWP_QUALITY DWP_RESOLUTIONX DWP_RESOLUTIONY DWP_SCALE DWP_SPECS DWP_XOFFSET DWP_YOFFSET		DWP_SETPAGES	

Report sequence	%WPR_SETDEVICE subfunctions	%WPR_GETDEVICE subfunctions	%WPR_INFO subfunctions	%WPR_PRINT subfunctions	%WPR_EXECUTE subfunctions
%WPR_PRINT(..., DWP_ENDJOB		DWP_COLLATE DWP_COLOR DWP_COPIES DWP_DUPLEX DWP_FORMNAME DWP_ORIENTATION DWP_PAPERLENGTH DWP_PAPERSIZE DWP_PIXELHEIGHT DWP_PIXELWIDTH DWP_PHYSICALHEIGHT DWP_PHYSICALWIDTH DWP_QUALITY DWP_RESOLUTIONX DWP_RESOLUTIONY DWP_SCALE DWP_SPECS DWP_XOFFSET DWP_YOFFSET	DWP_DELPEN DWP_GETFILENAME	DWP_SETPAGES	DWP_PREVIEW DWP_PRINT
%WPR_INFO(..., DWP_DELPRINTER)					

Sample programs

The following piece of code is the equivalent of `LPQUE(file_to_print)`, where *file_to_print* is the name of the file you want to print.

```
if (%wpr_info(handle, DWP_GETPRINTER))           ;Get the report handle
begin
    xcall wpr_print(handle, DWP_BEGINJOB)         ;Create the metafile
    xcall wpr_print(handle, DWP_ADDFILE, file_to_print) ;Specify the
                                                    ; file to print
    xcall wpr_print(handle, DWP_ENDJOB)           ;Close the metafile
    xcall wpr_execute(handle, DWP_PRINT)          ;Begin printing
    xcall wpr_info(handle, DWP_DELPRINTER)        ;Release the printer
end
```

The program below (**wprsampl.dbl**) is included in the `dbl\examples` directory of your Synergy Language distribution.

```
; Sample program to get you started with the Synergy Language Windows
; printing API. Although there are Toolkit calls, they are for messages
; and samples only. The core routines work without the Toolkit.

.include "DBLDIR:winprint.def"
.include "WND:tools.def"
```

```
record print_info
    lpp      ,i4
    cpl      ,i4
    flags    ,i4
    unused   ,a20
    printerid,a80
    driver    ,a32

record scratch
    ctr      ,i4
    file      ,a12    ,"c:\meta.smf"
    shorttext ,a60

record myHandles
    pHandles ,4d2
    pPassFail,4d2
    nothing  ,i4      ,0
    aPrinter ,i4      ,0
    bPrinter ,i4      ,0

record textmetricstuff
    cRow      ,i4      ,0
    cCol      ,i4      ,0
    rHeight   ,i4      ,0
    strLen    ,i4      ,0
    strHeight ,i4      ,0
    m_font_handle ,i4
    f_Handle  ,i4

proc
    open(1, o, "tt:")
    xcall u_start(,,,,,,,,10)

;Create a report handle, look at the text metrics, create a
; metafile, view it, and print it.
;Get the printer first...above all else
    aPrinter = %wpr_info(nothing, DWP_GETPRINTER)
    writes(1, "And the handle is " + %string(aPrinter))

;Create the meta file
    writes(1, "Creating metafile....please wait...")
    ;Calculate row height for printing
    rHeight = %wpr_info(aPrinter, DWP_Y_FROM_ROW, 2)
;Start the presses
    xcall wpr_print(aPrinter, DWP_BEGINJOB, file)
    xcall wpr_print(aPrinter, DWP_BEGINPAGE)
;Set some static text
    shorttext = " of some text I created with a Synergy/DE metafile."
```

```
;And write out a page
  for ctr from 1 thru 55
    begin
      xcall wpr_print(aPrinter, DWP_WRITEOUT, 0, (rHeight * ctr),
        & "Line "+ %string(ctr) + shorttext)
    end
;All done, shut it down
  xcall wpr_print(aPrinter, DWP_ENDPAGE)
  xcall wpr_print(aPrinter, DWP_ENDJOB)
;Let's look at our handiwork
  writes(1, "Preview....")
  xcall wpr_execute(aPrinter, DWP_PREVIEW, file, "Groovy, look at me",
    & TRUE)
;And print it
  writes(1, "And print it....")
  xcall wpr_execute(aPrinter, DWP_PRINT)
;All that's left is to clean up after the party
  xcall wpr_info(aPrinter, DWP_DELPINTER)
;Avoid accidental usage
  aPrinter = 0
  xcall u_message("Pretty cool, eh?")

;-----
;Here's some text info...
;Calculate and display text metric stuff
  writes(1, " ")
  writes(1, "Here's some text metric stuff")
  bPrinter = %wpr_info(nothing, DWP_GETPINTER)
  cCol = %wpr_info(bPrinter, DWP_X_FROM_COL, 10)
  cRow = %wpr_info(bPrinter, DWP_Y_FROM_ROW, 20)
;Simple way to calculate row height
  rHeight = %wpr_info(bPrinter, DWP_Y_FROM_ROW, 2)
  strLen = %wpr_info(bPrinter, DWP_TEXTWIDTH)
  writes(1, "Col 10 translated to " + %string(cCol) + " pixels")
  writes(1, "Row 20 translated to " + %string(cRow) + " pixels")
  writes(1, "Row Height is " + %string(rHeight) + " pixels")
  strLen = %wpr_info(bPrinter, DWP_TEXTWIDTH)
  writes(1, "Average char width is " + %string(strLen))
  strLen = %wpr_info(bPrinter, DWP_TEXTWIDTH, "m")
  writes(1, "String length of 'm' is " + %string(strLen))
  strLen = %wpr_info(bPrinter, DWP_TEXTWIDTH, "W")
  writes(1, "String length of 'W' is " + %string(strLen))
  strLen = %wpr_info(bPrinter, DWP_TEXTWIDTH, ".")
  writes(1, "String lenght of '.' is " + %string(strLen))
  strLen = %wpr_info(bPrinter, DWP_TEXTWIDTH, "TextString")
  writes(1, "Width of 'TextString' is " + %string(strLen))
  strHeight = %wpr_info(bPrinter, DWP_TEXTHEIGHT)
```

```
writes(1, "Average string height is " + %string(strHeight))
strHeight = %wpr_info(bPrinter, DWP_TEXTHEIGHT, "L")
writes(1, "String Height of 'L' is " + %string(strHeight))
strHeight = %wpr_info(bPrinter, DWP_TEXTHEIGHT, "m")
writes(1, "String Height of 'm' is " + %string(strHeight))
strHeight = %wpr_info(bPrinter, DWP_TEXTHEIGHT, "-")
writes(1, "String Height of '-' is " + %string(strHeight))

;Allocate text metric memory handle
m_font_handle = %mem_proc(DM_ALLOC, ^size(text_specs))
f_handle = %wpr_info(bPrinter, DWP_TEXTMETRICS,
& ^m(text_specs, m_font_handle))
;Display text_specs struc
writes(1, " ")
xcall u_message("Here are a bunch of text metric numbers")
writes(1, "Text metric structure info...")
;Int values
writes(1, " Height      " + %string(^m(text_specs.height,
& m_font_handle)))
writes(1, " Ascent     " + %string(^m(text_specs.ascent,
& m_font_handle)))
writes(1, " Descent     " + %string(^m(text_specs.descent,
& m_font_handle)))
writes(1, " IntLeading    " + %string(^m(text_specs.intleading,
& m_font_handle)))
writes(1, " ExtLeading    " + %string(^m(text_specs.extleading,
& m_font_handle)))
writes(1, " AveWidth     " + %string(^m(text_specs.avewidth,
& m_font_handle)))
writes(1, " MaxWidth     " + %string(^m(text_specs.maxwidth,
& m_font_handle)))
writes(1, " Weight       " + %string(^m(text_specs.weight,
& m_font_handle)))
writes(1, " OverHang     " + %string(^m(text_specs.overhang,
& m_font_handle)))
writes(1, " DigitizedX   " + %string(^m(text_specs.digitizedX,
& m_font_handle)))
writes(1, " DigitizedY   " + %string(^m(text_specs.digitizedY,
& m_font_handle)));Char values
writes(1, " FirstChar    " + ^m(text_specs.firstchar, m_font_handle))
writes(1, " LastChar     " + ^m(text_specs.lastchar, m_font_handle))
writes(1, " DefChar      " + ^m(text_specs.defaultchar, m_font_handle))
writes(1, " BreakChar    " + ^m(text_specs.breakchar, m_font_handle))
;Byte values
writes(1, " Italic?      " + %string(^m(text_specs.italic,
& m_font_handle)))
```

```

    writes(1, " UnderLine? " + %string(^m(text_specs.underlined,
&      m_font_handle)))
    writes(1, " Strikeout? " + %string(^m(text_specs.strikeout,
&      m_font_handle)))
    writes(1, " Pitch      " + %string(^m(text_specs.pitch,
&      m_font_handle)))
    writes(1, " Family      " + %string(^m(text_specs.charset,
&      m_font_handle)))
    writes(1, " Filler      " + ^m(text_specs.text_filler, m_font_handle))
    xcall wpr_info(bPrinter, DWP_DELPRIINTER)
    bPrinter = 0
    xcall u_message("Well?")
;-----
;Let's incorporate Toolkit
;See how I can get U_PRINTSETUP settings into my printer
    writes(1, " ")
    writes(1, "Let's play with the toolkit.")
    writes(1, "Change some setting in the first 4 u_printsetup dialog
&      boxes, ")
    writes(1, " and see if they hold for the following 4 DWP_PRINTDLGs.")
    writes(1, " Getting Printer handles: DWP_GETPRIINTER combined with " +
&      " u_printsetup")
    for ctr from 1 thru 4
      begin
        xcall u_printsetup
        pHandles(ctr) = %wpr_info(nothing, DWP_GETPRIINTER)
        writes(1, "   Returned pHandle for loop " + %string(ctr) +
&      " " + %string(pHandles(ctr)))
      end
    writes(1, " Setting based on Printer handles: DWP_PRINTDLG")
    for ctr from 1 thru 4
      begin
        pPassFail(ctr) = %wpr_info(pHandles(ctr), DWP_PRINTDLG)
        writes(1, "   Returned pHandle for loop " + %string(ctr) +
&      " " + %string(pHandles(ctr)) + " PassFail is " +
&      %string(pPassFail(ctr)))
      end
    writes(1, " Viewing changed setting: DWP_PRINTDLG")
    for ctr from 1 thru 4
      begin
        pPassFail(ctr) = %wpr_info(pHandles(ctr), DWP_PRINTDLG)
        writes(1, "   Returned pHandle for loop " + %string(ctr) +
&      " " + %string(pHandles(ctr)) + " PassFail is " +
&      %string(pPassFail(ctr)))
      end
    writes(1, "Check that u_printquery wasn't hammered")
    writes(1, "Should be the last u_printsetup, not DWP_PRINTDLG")

```

```
xcall u_printsetup
writes(1, "Deleting Printer handles: DWP_DELPRIINTER")
for ctr from 1 thru 4
  begin
    pPassFail(ctr) = %wpr_info(pHandles(ctr), DWP_DELPRIINTER)
    writes(1, "  Delete pHandle for loop " + %string(ctr) +
&      " " + %string(pHandles(ctr)) + " PassFail is " +
&      %string(pPassFail(ctr)))
  end
  xcall u_message("What fun can you have?")
;What fun can you have?
end
```

The following is an example of basic print processing that displays the Print dialog and enables the user to select pages. It is included as **tpri.dbl** in the `dbl\examples` directory of your Synergy Language distribution. (Another example in that directory, **tpri2.dbl**, assumes the default printer and accesses the previewer. The user can change printers from the previewer as long as the application responds by regenerating the metafile when needed.)

```
main test_print
;
.include "DBLDIR:winprint.def"
.align

.define D_NUMPAGES  ,3      ;Just for our example - # of pages to print

;Macro to allow easy access to elements of the dlg structure defined below
.define M_DLG(e)      ^m(dialog_specs.e,dlg)

stack record
  job      ,D_ADDR          ;Report handle
  page     ,D_NATINT        ;Current page number
  row      ,D_NATINT        ;Current row number
  rowht    ,D_NATINT        ;Height of a row in pixels
  pageht   ,D_NATINT        ;Height of the page in pixels
  pagewd   ,D_NATINT        ;Width of the page in pixels
  nmrows   ,D_NATINT        ;Number of rows on a page
  y        ,D_NATINT        ;Where to print row (vertically)
  dlg      ,a ^size(dialog_specs) ;Structure for DWP_PRINTDLG

proc
  xcall u_start
  xcall u_update
  job = %wpr_info(job, DWP_GETPRINTER);Get a report handle

  clear ^i(dlg)                ;Initialize structure to all 0s
```

```

    M_DLG(flags) = DWP_PRINTDLG_STYLEPRINT| ;Use Print dialog (not Setup)
&    DWP_PRINTDLG_SHOWTOFILE|           ;Allow "Print to file"
&    DWP_PRINTDLG_DISABLECURRENTPAGE|   ;Don't allow "Current page"
&    DWP_PRINTDLG_DISABLESELECT|        ;Don't allow "Selection"
&    DWP_PRINTDLG_COLLATE                ;Collate if multiple copies

    M_DLG(pages) = DWP_PRINTDLG_ALLPAGES ;Start with "All" checked
    M_DLG(copies) = 1                    ; and with 1 copy

;Now ask the user where to print, how many copies, what pages, etc.
if (%wpr_info(job, DWP_PRINTDLG, dlg) == DWP_PRINTDLG_PRINT)
    begin                                ;They said to print it
        call generate;Generate the report
        ;Now we'll let the user preview it first
        using %wpr_execute(job, DWP_PREVIEW,,,
&        DWP_PREVIEW_TOOLBAR|DWP_PREVIEW_PRINTDLG) select
(DWP_PREVIEW_PRINT),
        xcall wpr_execute(job, DWP_PRINT)    ;They said print it
(DWP_PREVIEW_REGEN),
        begin
            call generate                    ;We need to regenerate it
            xcall wpr_execute(job, DWP_PRINT) ;Then print it
        end
    endusing
end
xcall wpr_info(job, DWP_DELPRINTER)          ;Release the report handle
xcall u_finish
stop

;
;Routine to generate the report
generate,
    xcall wpr_print(job, DWP_BEGINJOB,, 1)    ;Start the job
    rowht = %wpr_info(job, DWP_TEXTHEIGHT, "A");Get height of a row
    xcall wpr_getdevice(job, DWP_PIXELHEIGHT, pageht) ;Page height
    xcall wpr_getdevice(job, DWP_PIXELWIDTH, pagewd) ;Page width
    nmrows = pageht / rowht                    ;Compute how many rows will fit
    for page from 1 thru D_NUMPAGES            ;For each page to print...
    begin
        xcall wpr_print(job, DWP_BEGINPAGE)    ;Start the page
        y = 0                                  ;Start at vertical pixel 0
        for row from 1 thru nmrows              ;For each row...
        begin
            xcall wpr_print(job, DWP_WRITEOUT, 0, y, "This is row " +
&            %string(row))
            y += rowht                          ;Move down a row for next one
        end
    end

```

```
        ;Just to be fancy, print the page number centered on the bottom
        ; (see below)
        xcall print_centered(job, (pageht - rowht), pagewd, "Page " +
&                                %string(page))
        xcall wpr_print(job, DWP_ENDPAGE)          ;Finished with this page
    end
    xcall wpr_print(job, DWP_ENDJOB)                ;Job is complete
    return
endmain

subroutine print_centered
;
; Description:      Print text centered horizontally on the page
;
; Arguments:
    a_job          ,n                ;Report handle (in, req)
    a_y            ,n                ;Vertical pixel coordinate (in, req)
    a_pagewidth    ,n                ;Width of the page in pixels (in, req)
    a_text         ,a                ;Text to print (in, req)
;
;Note: We do not trim the text for performance reasons. If you need it
; trimmed, trim it before you pass it.
;

.align
stack record
    textwidth      ,D_NATINT         ;Width of the text in pixels
    x              ,D_NATINT         ;Computed horizontal coordinate for
                                    ; the text

proc
    textwidth = %wpr_info(a_job, DWP_TEXTWIDTH, a_text)    ;Get width of
                                                            ; string
    if (textwidth >= a_pagewidth) then ;As wide as the page or wider?
        x = 0                                             ;Just start at the left and truncate
    else
                                                ;There is room for centering
        x = (a_pagewidth - textwidth) / 2 ;Compute left of centered text
    xcall wpr_print(a_job, DWP_WRITEOUT, x, a_y, a_text)    ;And print it
    xreturn
endsubroutine
```


%WPR_EXECUTE – Execute printing

WIN

return = %WPR_EXECUTE (*report_handle*, *operation*[, *arguments*])

Return value

return

Defined by the specified operation. See the Discussion section below.

Arguments

report_handle

The report handle returned by the DWP_GETPRINTER subfunction of %WPR_INFO. See [page 17-30](#). (n)

operation

One of the printer operations listed in the Discussion section below. (n)

arguments

(optional) Any arguments used by the specified operation. See the Discussion section below.

Discussion

If an invalid *report_handle* is passed, the runtime generates a trappable error. If a Win32 function fails, the DWP_ subfunction returns false and %SYSERR returns the status code from the last system call. Refer to the Windows **winerror.h** file or look under “Runtime error codes” on your Microsoft Developer Network CD.

Operation can be one of the following:

DWP_PRINT, [*filename*][, *keep_file*]

Plays the metafile to the printer selected in DWP_GETPRINTER. If *filename* is not a Synergy metafile, an error is returned.

Filename is passed to reprint an existing metafile. If *filename* is passed and the report handle points to a current print job, *filename* is verified against the current print job value. If *filename* is not passed, the current print job value is used.

Keep_file is passed as true to save the file after printing. The default is false, which deletes the metafile.

DWP_PRINT returns true if successful and false if unsuccessful.

DWP_PREVIEW, [*filename*], [*previewer_title*], [*flags*][, *help_routine*]

Opens a preview window displaying the metafile. (See [“Print previewer” on page 17-17.](#)) If *filename* is not a Synergy metafile, an error is returned.

Previewer_title is the text that is displayed on the preview window’s title bar. The default value for *previewer_title* is “Synergy Print Preview.” The maximum length of *previewer_title* is 128 characters, including the “ - page *n* of *n*” appended trailing text.

If specified, *flags* is one or more of the following values:

DWP_PREVIEW_TOOLBAR The toolbar is displayed in the preview window.

DWP_PREVIEW_PRINTDLG The Print dialog box is available to the user from the preview window via the Print entry on the File menu.

Help_routine is the name of a Synergy Language routine to handle the user’s request for Help. If *help_routine* is not passed or is blank, the Help menu item does not appear.

If Help is invoked, the previewer remains visible on the screen. Any windows placed by *help_routine* must also be removed by the help routine.

DWP_PREVIEW returns one of the following:

DWP_PREVIEW_PRINT The user opted to print the job by selecting File > Print now, by clicking the Print button on the toolbar, or by selecting File > Print and then clicking the Print button in the Print dialog.

DWP_PREVIEW_CANCEL The user closed the print previewer without printing the job by closing the window, pressing ESC, or selecting File > Exit.

DWP_PREVIEW_REGEN The user opted to print the job, but the metafile for the document should be regenerated prior to printing, because the user selected a printer with different resolution or modified the page characteristics in some way that would render the metafile output incorrectly. (See [page 17-15](#) for more information.)

Print previewer

The preview window is displayed when DWP_PREVIEW is passed. From this window you can preview any page (or range of pages) in a document to see how it will look when it is printed. You can zoom in or out, changing the size of the page as displayed in the window. (See [figure 17-1.](#))



Some truncation may occur when you zoom a document that contains a font that is not device independent and scalable. We strongly recommend using TrueType or OpenType fonts with the print previewer. Other fonts may only provide a limited number of point sizes and therefore will not scale well. Additionally, some truncation may occur when you use a fixed-pitch TrueType or OpenType font. You can eliminate this truncation by using the *enhanced_preview* flag in %WPR_PRINT's DWP_BEGINJOB subfunction. For more information, see [page 17-36.](#)

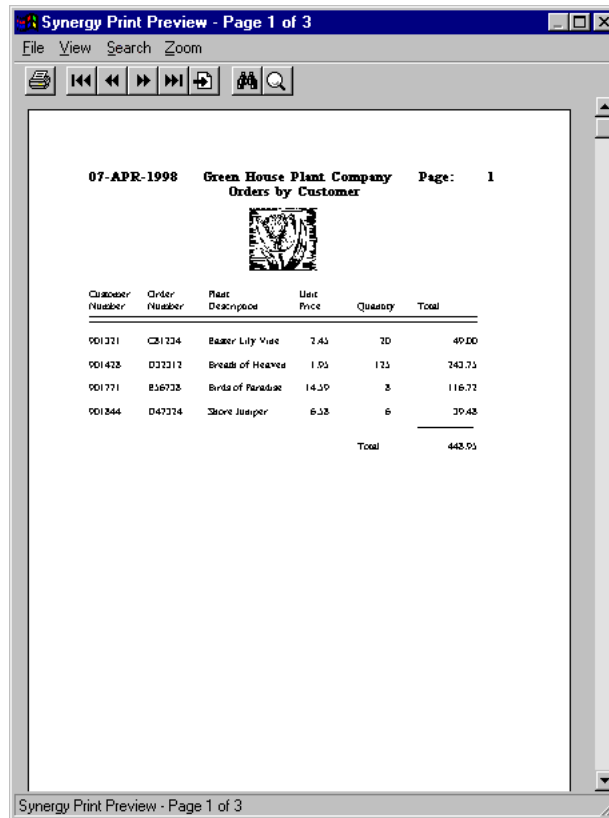


Figure 17-1. A sample print preview.

To search for text while in the previewer, select Search > Find or click the Find toolbar button. A dialog box will be displayed that prompts for the following search criteria:

Find what: Enter the text you want to search for.

Match case: Select Match case if you want your search to be case-sensitive.

Direction: Select Forward to search forward or Backward to search backward through the previewed text.

To search for subsequent instances of the specified text, select Search > Find next.



Whenever a search in the previewer highlights text, enhanced preview mode is automatically turned off in the page display. However, this does not affect the alignment of the text when it is printed.

To print directly from the previewer, select File > Print now. To display the Print dialog box, select File > Print. If a range of pages was specified (either by the user or via the DWP_SETPAGES subfunction of %WPR_PRINT), the Pages radio button will be selected in the dialog, and the associated text field will contain those page ranges. Otherwise, the All radio button will be selected.

If the application was responsible for printing the selected pages (in other words, DWP_PRINTDLG_APPPAGE was passed to DWP_PRINTDLG),

- ▶ the previewer assumes that the pages printed correspond to the specified ranges.
- ▶ if the user selects a range of pages that lies outside the ranges of pages displayed by the previewer (including selecting All), a message that those pages are not available is displayed and the user is returned to the dialog.
- ▶ the user may select any subset of the original set of page ranges, including Current page. Any subsequent execution of the DWP_PRINT subfunction of %WPR_EXECUTE will honor that range.

If the application was not responsible for printing the selected pages, the user can select any range of pages (including All) and any subsequent DWP_PRINT operation will print those pages.

If the user selects a different printer in this dialog, that printer becomes the destination for any subsequent execution of the DWP_PRINT subfunction of %WPR_EXECUTE. When this happens, the metafile may need to be regenerated. To avoid skewing the report, the metafile must also be regenerated when the page orientation or resolution is changed or when a different paper size is selected. In any of these cases, a value of DWP_PREVIEW_REGEN will be returned to tell the calling application to regenerate the metafile (which should be done prior to calling the DWP_PRINT subfunction of %WPR_EXECUTE). If the calling application does not regenerate the metafile before calling DWP_PRINT, the resulting output will probably differ from what the user expected.

Changes to copies, collation, or selection of Print to file will be applied when printing without requiring regeneration of the metafile. The Properties button in the dialog is disabled to prevent the user from changing the orientation or paper source for the report. If the user clicks the Print button in this dialog, the previewer will update the report settings with the information specified in the dialog. Then the previewer will exit and return DWP_PREVIEW_PRINT to the application.

If the user presses the Apply button on this dialog and then later presses Cancel, the settings are updated, but the previewer won't exit. The return value when the previewer exits later will be determined by the method used to exit at that time.

Examples

The example below opens a preview window titled "Print Preview." If the user chooses to print the file, the file is sent to the printer specified by the DWP_GETPRINTER subfunction of %WPR_INFO.

```
if (%wpr_execute(aPrinter, DWP_PREVIEW, file, "Print Preview", TRUE) .eq.  
    & DWP_PREVIEW_PRINT)  
    xcall wpr_execute(aPrinter, DWP_PRINT)
```

Also see ["Sample programs" on page 17-7](#).

%WPR_GETDEVICE – Retrieve printer settings

WIN

status = %WPR_GETDEVICE(*report_handle*, *capability*[, *setting*])

Return value

status

One of the following values:

true The printer supports the specified capability.

false The printer does not support the capability, or a Windows system call failed.

Arguments

report_handle

The report handle returned by the DWP_GETPRINTER subfunction of %WPR_INFO. (See [page 17-30](#).) (**n**)

capability

One of the printer capabilities listed in the Discussion section below. (**n**)

setting

(optional) Returned with the current printer setting if that capability is supported for the device. (If the capability is not supported, *setting* is undefined.) (**a** or **n**)

Discussion

If an invalid *report_handle* is passed, the runtime generates a trappable error. If the printer does not support the capability, *status* is false and %SYSERR returns 0. If a Win32 function fails, *status* is false and %SYSERR returns the status code from the last system call. Refer to the Windows **winerror.h** file or look under “Runtime error codes” on your Microsoft Developer Network CD.

Capability can be one of the following:

DWP_ORIENTATION

Gets the orientation for the document. *Setting* values are as follows:

DWPO_PORTRAIT

DWPO_LANDSCAPE

DWP_PAPERSIZE

Gets the paper size for the document. Overrides both the DWP_PAPERLENGTH and DWP_PAPERWIDTH capabilities. *Setting* values are as follows (additional values may be added as Microsoft expands its list):

DWPP_LETTER
DWPP_LETTERSMALL
DWPP_TABLOID
DWPP_LEDGER
DWPP_LEGAL
DWPP_STATEMENT
DWPP_EXECUTIVE
DWPP_A3
DWPP_A4
DWPP_A4SMALL
DWPP_A5
DWPP_B4
DWPP_B5
DWPP_FOLIO
DWPP_QUARTO
DWPP_10X14
DWPP_11X17
DWPP_NOTE
DWPP_ENV_9
DWPP_ENV_10
DWPP_ENV_11
DWPP_ENV_12
DWPP_ENV_14
DWPP_CSHEET
DWPP_DSHEET
DWPP_ESHEET
DWPP_ENV_DL
DWPP_ENV_C5
DWPP_ENV_C3
DWPP_ENV_C4
DWPP_ENV_C6
DWPP_ENV_C65
DWPP_ENV_B4
DWPP_ENV_B5
DWPP_ENV_B6
DWPP_ENV_ITALY
DWPP_ENV_MONARCH
DWPP_ENV_PERSONAL

DWPP_FANFOLD_US
DWPP_FANFOLD_STD_GERMAN
DWPP_FANFOLD_LGL_GERMAN
DWPP_ISO_B4
DWPP_JAPANESE_POSTCARD
DWPP_9X11
DWPP_10X11
DWPP_15X11
DWPP_ENV_INVITE
DWPP_RESERVED_48
DWPP_RESERVED_49
DWPP_LETTER_EXTRA
DWPP_LEGAL_EXTRA
DWPP_TABLOID_EXTRA
DWPP_A4_EXTRA
DWPP_LETTER_TRANSVERSE
DWPP_A4_TRANSVERSE
DWPP_LETTER_EXTRA_TRANSV
DWPP_A_PLUS
DWPP_B_PLUS
DWPP_LETTER_PLUS
DWPP_A4_PLUS
DWPP_A5_TRANSVERSE
DWPP_B5_TRANSVERSE
DWPP_A3_EXTRA
DWPP_A5_EXTRA
DWPP_B5_EXTRA
DWPP_A2
DWPP_A3_TRANSVERSE
DWPP_A3_EXTRA_TRANSVERSE

DWP_PAPERLENGTH

Gets the paper length in tenths of a millimeter. Overrides the setting for DWP_PAPERSIZE.

DWP_PAPERWIDTH

Gets the paper width in tenths of a millimeter. Overrides the setting for DWP_PAPERSIZE.

DWP_SCALE

Gets the scaling factor, where 100 indicates no scaling.

DWP_COPIES

Gets the number of copies to print.

DWP_QUALITY

Gets the print quality. *Setting* values are as follows:

DWPQ_HIGH
DWPQ_MEDIUM
DWPQ_LOW
DWPQ_DRAFT

DWP_COLOR

Gets the color/monochrome mode:

DWPC_COLOR
DWPC_MONOCHROME

DWP_DUPLEX

Gets the duplex mode:

DWPD_SIMPLEX
DWPD_HORIZONTAL
DWPD_VERTICAL

DWP_RESOLUTIONX

Gets the X pixel resolution (DPI).

DWP_RESOLUTIONY

Gets the Y pixel resolution (DPI).

DWP_COLLATE

Gets the collation of multiple copies. True indicates multiple copies are collated; false indicates no collation.

DWP_FORMNAME

Gets the name of the form to use. It overrides DWP_PAPERSIZE, as well as DWP_PAPERLENGTH and DWP_PAPERWIDTH.

DWP_PIXELWIDTH

Gets the printable width of the device's page, in pixels.

DWP_PIXELHEIGHT

Gets the printable height of the device's page, in pixels.

DWP_PHYSICALWIDTH

Gets the physical (printable plus nonprintable) width of the device's page, in pixels.

DWP_PHYSICALHEIGHT

Gets the physical (printable plus nonprintable) height of the device's page, in pixels.

DWP_XOFFSET

Gets the offset, in pixels, from the left edge of the device's page to the printable left edge.

DWP_YOFFSET

Gets the offset, in pixels, from the top edge of the device's page to the printable top edge.

DWP_FONT

Gets the font characteristics. *Setting* can be mapped to the following structure:

```

structure font_specs
    facename      ,D_WPRFONTNAME ;Name of typeface
    pointsize     ,i4             ;Point size
    weight        ,i4             ;Font weight (DWPFW_DONTCARE,
                                ; DWPFW_THIN, DWPFW_EXTRALIGHT,
                                ; DWPFW_LIGHT, DWPFW_NORMAL,
                                ; DWPFW_MEDIUM, DWPFW_SEMIBOLD,
                                ; DWPFW_BOLD, DWPFW_EXTRABOLD,
                                ; DWPFW_HEAVY)
    italic        ,i1             ;True if italic
    underline     ,i1             ;True if underline
    strikeout     ,i1             ;True if strikeout
    fixed         ,i1             ;True if fixed-pitch
    font_fill     ,a32            ;Filler
    orientation   ,i4             ;Character orientation in tenths of a
                                ; degree from x
    escapement    ,i4             ;Text baseline escapement in tenths of
                                ; a degree

```

Note that **orientation** and **escapement** enable you to rotate characters and text strings:

- ▶ **Orientation** specifies the angle, in tenths of a degree, between the baseline of a character and the page's horizontal axis. The default is zero, which represents the normal horizontal orientation of a character.
- ▶ **Escapement** specifies the angle, in tenths of a degree, between the baseline of a string of text and the horizontal axis of a page. The default is zero, which represents the normal horizontal orientation of a text string.

Positive values rotate text or characters counter-clockwise, and negative values rotate the text clockwise.

Generally, **orientation** and **escapement** will match, but a text string can, for example, be printed vertically while each character in the string is oriented horizontally. In this case, **orientation** would be 0 and **escapement** would be -900 (which is 90 degrees clockwise) or 2700 (which is 270 degrees counter-clockwise).

This structure is defined in **DBLDIR:winprint.def**, along with **D_WPRFONTNAME (a32)**.

DWP_TEXTCOLOR

Gets the color (as an RGB triplet) for the text. An RGB triplet can be constructed using the supplied macro (in **DBLDIR:winprint.def**):

`M_RGB (red,green,blue)`

where *red*, *green*, and *blue* are intensities (in the range 0–255) for each color component. If all three are 255, the color is white; if all are 0, the color is black.

DWP_BACKCOLOR

Gets the background color (as an RGB triplet).

DWP_PAPERSOURCE

Sets and retrieves the paper source for a printer. *Setting* is one of the following:

DWPS_UPPER	Upper bin (or the only one)
DWPS_LOWER	Lower bin
DWPS_MIDDLE	Middle bin
DWPS_MANUAL	Manual feed
DWPS_ENVELOPE	Envelope bin
DWPS_ENVMANUAL	Manual envelope feed
DWPS_AUTO	Auto-select by paper size
DWPS_TRACTOR	Tractor feed
DWPS_SMALLFMT	Small format
DWPS_LARGEfmt	Large format
DWPS_LARGECAPACITY	Large capacity bin
DWPS_CASSETTE	Cassette
DWPS_FORMSOURCE	Source for selected form
DWPS_USER	User-defined

The returned arguments is an integer value that indicates the source currently selected for the printer. If the device doesn't support setting the source, or if the specified source doesn't exist, the returned arguments is false.

DWP_SPECS

Gets all of the printer capabilities at once. *Setting* can be mapped to the following structure:

```

structure print_specs
    printer          ,D_PRINTERNAME ;Name of the printer
    orientation      ,i4             ;DWPO_PORTRAIT/DWPO_LANDSCAPE
    paper_size       ,i4             ;Paper size for next document (see above)
    paper_length     ,i4             ;Paper length in tenths of a millimeter
    paper_width      ,i4             ;Paper width in tenths of a millimeter
    scale            ,i4             ;Scaling factor
    copies           ,i4             ;Number of copies
    quality          ,i4             ;DWPQ_(HIGH, MEDIUM, LOW, DRAFT) or 0
    color            ,i4             ;DWPC_(COLOR, MONOCHROME)

```

```

duplex           ,i4      ;DWPD_(SIMPLEX, HORIZONTAL, VERTICAL)
resolution_x     ,i4      ;X pixel resolution (or 0)
resolution_y     ,i4      ;Y pixel resolution (or 0)
collate          ,i4      ;Collation true or false
formname         ,D_FORMNAME ;Name of form
pixelwidth       ,i4      ;Width of the page in pixels
pixelheight      ,i4      ;Height of the page in pixels
physicalwidth    ,i4      ;Width (total) of page in pixels
physicalheight   ,i4      ;Height (total) of page in pixels
xoffset          ,i4      ;X offset of page in pixels
yoffset          ,i4      ;Y offset of page in pixels
paper_source     ,i4      ;Location of source of paper to print on
ps_filler        ,a60     ;Print Spec filler
extended_pr_name ,D_PRINTERNAME_EX ;Extended size printer name

```

This structure is defined in **DBLDIR:winprint.def**, along with **D_PRINTERNAME (a128)**, **D_PRINTERNAME_EX (a256)**, **D_FORMNAME (a32)**, and **D_FONT (a^size(font))**. See the documentation for each of the individual capabilities for more information.

To change everything, retrieve the setting using **%WPR_GETDEVICE** with the **print_specs** structure and change what you want to change. Then call **%WPR_SETDEVICE** with the updated **print_specs** structure specified. Some printer capabilities cannot be modified. See **%WPR_SETDEVICE** for the list of modifiable capabilities.



Depending on what features and settings they support, some printer drivers may not return all information or may return incorrect information. The accuracy of the information returned by **%WPR_GETDEVICE** rests solely on the accuracy of the information supplied by the printer driver.

Examples

See “[Sample programs](#)” on page 17-7.

%WPR_INFO – Set and retrieve extended device capabilities

WIN

return = %WPR_INFO(*report_handle*, *operation*[, *arguments*])

Return value

return

Defined by the specified operation. See the Discussion section below.

Arguments

report_handle

The report handle returned by the DWP_GETPRINTER subfunction of %WPR_INFO. See the Discussion section below. (n)

operation

One of the printer operations listed in the Discussion section below. (n)

arguments

(optional) Any arguments used by the specified operation. See the Discussion section below.

Discussion

If an invalid *report_handle* is passed, the runtime generates a trappable error. If a Win32 function fails, the DWP_ subfunction returns false and %SYSERR returns the status code from the last system call. Refer to the Windows **winerror.h** file or look under “Runtime error codes” on your Microsoft Developer Network CD.

Operation can be one of the following:

DWP_PRINTDLG[, *dialog_specs*]

Pops up a printer selection dialog box that enables the user to select many of the features described under %WPR_GETDEVICE and specifies information about the dialog.

Dialog_specs is the structure provided below. This structure is defined in

DBLDIR:winprint.def. If *dialog_specs* is not passed, all structure elements are treated as 0, with the following exceptions:

- ▶ *flags*. The DWP_PRINTDLG_COLLATE flag is set or cleared based on the previous setting for the print job as specified by %WPR_SETDEVICE(DWP_COLLATE), or the default for the device if not previously specified. All other flags are treated as not set.
- ▶ *copies*. The *copies* field defaults to the previous setting for the print job as specified by %WPR_SETDEVICE(DWP_COPIES), or 1 if not previously specified.

```

structure dialog_specs      ;Structure of Print dialog info
    flags                   ;Flags to control dialog behavior:
                                ; DWP_PRINTDLG_STYLEPRINT, use "Print"
                                ; dialog instead of "Print Setup"
                                ; And one or more of the following if
                                ; DWP_PRINTDLG_STYLEPRINT is used:
                                ; DWP_PRINTDLG_SHOWTOFILE, show and
                                ; enable "Print to file" check box
                                ; DWP_PRINTDLG_TOFILE, check "Print to
                                ; file" check box
                                ; DWP_PRINTDLG_DISABLECURPAGE, disable
                                ; "Current page" radio button
                                ; DWP_PRINTDLG_DISABLEPAGENUMS, disable
                                ; "Pages" radio button and edit control
                                ; DWP_PRINTDLG_DISABLESELECT, disable
                                ; "Selection" radio button and edit
                                ; control
                                ; DWP_PRINTDLG_COLLATE, check "Collate"
                                ; check box
                                ; DWP_PRINTDLG_APPPAGE, application
                                ; processes all page selections
    pages                   ;Type of page selection to print. May
                                ; be one of the following:
                                ; DWP_PRINTDLG_ALLPAGES, all pages
                                ; (default)
                                ; DWP_PRINTDLG_CURRENT, current page
                                ; DWP_PRINTDLG_PAGES, page range(s)
                                ; DWP_PRINTDLG_SELECTION, selection
    npagerange              ;Number of page ranges
    group pagerange         ;Page ranges
        from_page          ;Beginning of a page range
        to_page            ;End of a page range
    endgroup
    copies                  ;Number of copies
    dialog_filler           ;Reserved for future use

```

You must call DWP_GETPRINTER before DWP_PRINTDLG.

DWP_PRINTDLG returns one of the following values:

DWP_PRINTDLG_CANCEL	The user either clicked Cancel or closed the dialog box.
DWP_PRINTDLG_PRINT	The user clicked OK (or Print when using DWP_STYLEPRINT) in the dialog box.
DWP_PRINTDLG_APPLY	The user clicked Apply, then Cancel in the dialog box.

All fields in the **dialog_specs** structure are both passed and returned. The values in *flags* can be combined using .BOR. to combine the desired behaviors.

If DWP_PRINTDLG_APPPAGE is specified, it is the responsibility of the calling application to submit only the pages requested in this case or to programmatically specify the ranges of pages to print. (See the DWP_SETPAGES subfunction of %WPR_PRINT on [page 17-40](#) for information about programatically setting a range of pages.) If DWP_PRINTDLG_APPPAGE is not specified, and the user selects “All” or “Pages” in the Print dialog, the application must print the entire report, and the Synergy Windows printing API automatically prints only the requested pages, both in the previewer and when printing.



DWP_PRINTDLG_APPPAGE is normally only used if an application's logical page numbers do not correspond to physical page numbers. For instance, if physical page 1 is a title page, then logical page 1 may be physical page 2, and so forth. In this type of scenario, DWP_PRINTDLG_APPPAGE would be used when DWP_PRINTDLG is called immediately after DWP_GETPRINTER, so the application knows which pages to generate. In most cases, this flag is not used at all.

If the user selects “Current page” or “Selection,” it is the responsibility of the calling application to print only the pages specified, or to programmatically specify the range of pages, regardless of whether or not DWP_PRINTDLG_APPPAGE is specified.

When passed, *pages* can be one of the following values:

If <i>pages</i> is	The selected radio button is
DWP_PRINTDLG_ALLPAGES	All
DWP_PRINTDLG_CURRENT	Current page
DWP_PRINTDLG_PAGES	Pages
DWP_PRINTDLG_SELECTION	Selection

The reverse of this table is true for the value returned in *pages*; if the user selected the All radio button, DWP_PRINTDLG_ALLPAGES is returned in *pages*, and so forth.

When passed, the entries in the *pagerange* array, up to *npagerange*, are used to populate the Pages field in the dialog. Upon return, if the user entered or left a value in the Pages field, the specified ranges are returned in the *pagerange* array, and *npagerange* is returned with the number of those ranges. A maximum of 256 independent page ranges can be entered.

If the user entered a number into the Number of copies spin-edit control, that number is returned in *copies*, even though your application is not expected to handle copies or collation. The Synergy Windows printing API will print the specified number of copies when the DWP_PRINT subfunction of %WPR_EXECUTE is called.

DWP_GETPRINTER[, *device_name*]

Allocates and returns a report handle (in both the return value and the *report_handle* argument) that you can pass to subsequent %WPR_ functions. DWP_GETPRINTER must be called before any other %WPR_ function. The *device_name* is the name of the printer. If *device_name* is not passed, it is set to the Windows default printer.

DWP_DELPRINTER[, *keep_file*]

Deletes this report handle and all associated objects created with DWP_GETPRINTER. If DWP_DELPRINTER is not called, the objects are released during runtime shutdown. If *keep_file* is true, the metafile will not be deleted by this subfunction. If *keep_file* is false, the metafile will be deleted, regardless of any previously specified argument passed to the DWP_PRINT subfunction of %WPR_EXECUTE. If *keep_file* is not specified, any previously specified *keep_file* argument on the DWP_PRINT subfunction of %WPR_EXECUTE will apply. If that argument wasn't specified, the metafile will be deleted.

DWP_DELPRINTER returns true if the operation is successful.

DWP_GETFILENAME, *filename*

Retrieves the metafile name that corresponds to the passed report handle and returns it in *filename*. If *document_name* was passed in %WPR_PRINT's DWP_BEGINJOB call, this name is returned. If *document_name* was not passed, the temporary filename is returned.

DWP_GETFILENAME returns true if successful.

DWP_GETPEN[, *pen_specs*]

Create a pen or brush for drawing lines. DWP_GETPEN without the optional structure returns a DWP_GEOMETRIC pen one pixel wide. To create thicker pens or brushes, pass *pen_specs*, which is the structure provided below. This structure is defined in **DBLDIR:winprint.def**.



Any memory handle created for the *pen_specs* structure *must* be initialized with DM_NULL. See **winprt1.dbl** (distributed in the dblexamples directory) for an example.

```

structure pen_specs
    pen_style    ,i4 ;Pen style
                    ; DWP_GEOMETRIC creates a pen with extended
                    ; attributes
                    ; DWP_COSMETIC creates a single-pixel pen
    line_style   ,i4 ;Line Style -- can contain any combination of
                    ; DWP_ALTERNATE, every other pixel (cosmetic pens
                    ; only)
                    ; DWP_SOLID, solid
                    ; DWP_DASH, dashed
                    ; DWP_DOT, dotted
                    ; DWP_DASHDOT, alternating dashes and dots
                    ; DWP_DASHDOTDOT, alternating dashes and double dots

```



```

;A pen_style value of DWP_GEOMETRIC allows for end
; caps and line joining specifications:
; DWP_ENDCAP_ROUND, round end caps
; DWP_ENDCAP_SQUARE, square end caps
; DWP_ENDCAP_FLAT, flat end caps
; DWP_JOIN_BEVEL, beveled joins
; DWP_JOIN_ROUND, round joins
line_width ,i4 ;Line width. If pen_style is DWP_GEOMETRIC, the
; width is specified in logical units. If pen_style
; is DWP_COSMETIC, the default value 1 is used
brush_style ,i4 ;If pen_style is DWP_GEOMETRIC, the brush style must
; be defined as one of the following:
; DWP_BS_HATCHED, hatched brush
; DWP_BS_HOLLOW, hollow brush
; DWP_BS_SOLID, solid brush
; If pen_style is DWP_COSMETIC, the default value
; DWP_BS_SOLID is used
brush_color ,i4 ;The brush color can be defined by M_RGB()
hatch_style ,i4 ;If pen_style is DWP_GEOMETRIC and the brush style
; is DWP_BS_HATCHED, the brush hatch can be defined
; as DWP_HS_BDIAGONAL, 45-degree upward, left-to-
; right hatch
; DWP_HS_CROSS, horizontal and vertical cross-hatch
; DWP_HS_DIAGCROSS, 45-degree crosshatch
; DWP_HS_FDIAGONAL, 5-degree downward, left-to-right
; hatch
; DWP_HS_HORIZONTAL, horizontal hatch
; DWP_HS_VERTICAL, vertical hatch
pen_filler ,a16 ;Pen filler

```

DWP_GETPEN returns a pen handle.

DWP_DELPEN, *pen_handle*

Deletes the pen created with DWP_GETPEN. If DWP_DELPEN is not called, the objects are released during runtime shutdown.

DWP_DELPEN returns true if successful or false if an invalid pen handle is passed.

DWP_X_FROM_COL, *column*

Returns the X pixel coordinate for a column in the current font on the current printer. The pixel returned for an integer column is the origin of that column. A fractional column may be specified by passing an implied-decimal value. See “DWP_GETSTATUS” on page 17-33.

DWP_Y_FROM_ROW, *row*

Returns the Y pixel coordinate for a row in the current font on the current printer. The pixel returned for an integer row is the origin of that row. A fractional row may be specified by passing an implied-decimal value. See “DWP_GETSTATUS” on page 17-33.

DWP_TEXTWIDTH[, *string*]

Returns the average character width as defined by the font if nothing is passed or the width of the passed string if *string* is passed. The return value is in pixels. See “DWP_GETSTATUS” on page 17-33.

DWP_TEXTHEIGHT[, *string*]

Returns the row height, in pixels, calculated as

height + external leading

if nothing is passed or the actual row height of the string if *string* is passed. See “DWP_GETSTATUS” on page 17-33.

DWP_LARGESTFONT, *height*

Returns the largest point size available in the current font that fits the specified *height* in pixels.

DWP_TEXTMETRICS, *text_specs*

Returns a structure containing basic information about the current font. *Text_specs* is the structure provided below. This structure is defined in **DBLDIR:winprint.def**.

```

structure text_specs
    height      ,i4 ;Character height (ascender + descender)
    ascent      ,i4 ;Character ascent (units above the base line)
    descent     ,i4 ;Character descent (units below the base line)
    intleading   ,i4 ;Leading inside height
    extleading   ,i4 ;Character leading outside the font
    avewidth     ,i4 ;Average character width (usually x)
    maxwidth     ,i4 ;Width of the widest character
    weight       ,i4 ;Font weight
    overhang     ,i4 ;Extra width per string that may be added to
                    ; synthesized fonts, such as bold or italic
    digitizedX   ,i4 ;Horizontal aspect of the device the font was
                    ; designed for
    digitizedY   ,i4 ;Vertical aspect of the device the font was
                    ; designed for
    firstchar    ,a1 ;Value of the first character defined in the font
    lastchar     ,a1 ;Value of the last character defined in the font
    defaultchar  ,a1 ;Value of the character to be substituted for
                    ; characters not in the font
    breakchar    ,a1 ;Value of the character used to define word breaks
                    ; for text justification
    italic       ,i4 ;True if font is an italic font
    underlined   ,i4 ;True if font is an underlined font
    strikeOut    ,i4 ;True if font is a strikeout font
    pitch        ,i4 ;Font pitch, technology, and family
                    ; DWP_VARIABLE_PITCH, variable pitch font
                    ; DWP_VECTOR, vector font

```

```

; DWP_TRUE_TYPE, true type font
; DWP_DEVICE, device font
charset      ,i4 ;Font character set
text_filler  ,a32 ;Text spec filler

```

DWP_GETSTATUS

Returns one or more of the following values (defined in **winprint.def**) that indicate the status of the printer:

Mnemonic	Value	The printer is...
DWPST_PAUSED	^x(1)	Paused
DWPST_PENDING_DELETION	^x(4)	About to be deleted from the user's profile
DWPST_PAPER_JAM	^x(8)	Experiencing a paper jam
DWPST_PAPER_OUT	^x(10)	Out of paper
DWPST_MANUAL_FEED	^x(20)	Awaiting manual paper feed
DWPST_PAPER_PROBLEM	^x(40)	Having a paper problem other than a jam or out of paper
DWPST_OFFLINE	^x(80)	Offline
DWPST_IO_ACTIVE	^x(100)	In an active input/output state
DWPST_BUSY	^x(200)	Busy
DWPST_PRINTING	^x(400)	Printing
DWPST_OUTPUT_BIN_FULL	^x(800)	Waiting for the output bin to be cleared
DWPST_NOT_AVAILABLE	^x(1000)	Not available for printing
DWPST_WAITING	^x(2000)	Waiting
DWPST_PROCESSING	^x(4000)	Processing a command to change settings
DWPST_INITIALIZING	^x(8000)	Initializing
DWPST_WARMING_UP	^x(10000)	Warming up
DWPST_TONER_LOW	^x(20000)	Low on toner
DWPST_NO_TONER	^x(40000)	Out of toner
DWPST_PAGE_PUNT	^x(80000)	Unable to print the current page

Mnemonic	Value	The printer is...
DWPST_USER_INTERVENTION	^x(100000)	Waiting for the user to respond to an error
DWPST_OUT_OF_MEMORY	^x(200000)	Out of memory
DWPST_DOOR_OPEN	^x(400000)	Waiting for someone to close the printer door
DWPST_SERVER_UNKNOWN	^x(800000)	Not able to report status
DWPST_POWER_SAVE	^x(1000000)	In power save mode
—	0	Not experiencing a problem
—	-1	Attempt to query printer status failed. Use %SYSERR to retrieve the sytem error code associated with this failure.

Limitations of %WPR_INFO

When using the DWP_X_FROM_COL and DWP_Y_FROM_ROW or the DWP_TEXTWIDTH and DWP_TEXTHEIGHT subfunctions to compute the distance between characters, the returned values are usually correct but may have small innaccuracies in some cases:

- ▶ The Microsoft Platform SDK warns that certain devices kern characters, which means the width of a string may not equal the sum of the widths of each of the characters.
- ▶ Some devices report incorrect information for fonts that are not device independent and scalable. Usually the fonts are reported as being slightly smaller than they are, but we have also seen reports that the fonts are larger.
- ▶ If you do not pass the *enhanced_preview* flag to the DWP_BEGINJOB subfunction of %WPR_PRINT, no intercharacter spacing information is preserved in the metafile. Instead, the natural spacing for that font in the given device is used. This mostly affects the spacing of characters in the previewer, but we have observed cases where the printed output can be affected as well. This phenomenon primarily applies to fonts that are not device independent and scalable.

Given the above information, if you must position text and graphics (for example, lines) close to one another, use the following guidelines:

- ▶ Use only device-independent, scalable fonts, such as TrueType or OpenType. A number of TrueType fixed-pitch fonts are available (Courier New, Lucida Console, Arial Monospaced, and so forth), as well as many proportional fonts.
- ▶ Pass the *enhanced_preview* flag to DWP_BEGINJOB. Doing so significantly increases the size of the metafile, but it also ensures the best scaling of fonts.

- ▶ If any text lies to the right of a line or other graphic, position that text independently. Don't rely on a given number of spaces (or other characters) to push the text far enough (or not too far) to the right.
- ▶ If possible, add a couple of extra pixels into the equation. For instance, if you are drawing a line between each row of text, use the text height plus the pen width for the row size.
- ▶ Draw lines and graphics last. When text is output, it can overwrite the entire rectangle for the characters, so lines that intersect that rectangle are erased.

Examples

See [“Sample programs”](#) on page 17-7.

%WPR_PRINT – Perform print operation to selected printer

WIN

return = %WPR_PRINT(*report_handle*, *operation*[, *arguments*])

Return value

return

Defined by the specified operation. See the Discussion section below.

Arguments

report_handle

The report handle returned by the DWP_GETPRINTER subfunction of %WPR_INFO. See [page 17-30](#). (n)

operation

One of the printer operations listed in the Discussion section below. (n)

arguments

(optional) Any arguments used by the specified operation. See the Discussion section below.

Discussion

If an invalid *report_handle* is passed, the runtime generates a trappable error. If a Win32 function fails, the DWP_ subfunction returns false and %SYSERR returns the status code from the last system call. Refer to the Windows **winerror.h** file or look under “Runtime error codes” on your Microsoft Developer Network CD.

Operation can be one of the following:

DWP_BEGINJOB, [*document_name*][, *enhanced_preview*]

Opens a document for printing. *Document_name* is the name of the metafile to create. The metafile cannot be a remote file specification using *x/Server*. Although we don’t recommend using a remote file in this situation for performance reasons, if you must use one, you will need to use a UNC path to specify it. If *document_name* is not passed, a temporary metafile is created. If the *keep_file* flag from %WPR_EXECUTE’s DWP_PRINT operation is passed and is true, the file is not deleted after printing.

If *enhanced_preview* is passed and is true, the text is output to a metafile just like any other GDI operation. This mode correctly scales fixed-pitch TrueType fonts when using the print previewer, and playing it back to a different device (the screen, for instance) scales the output along with all other GDI operations in a consistent manner. However, this enhanced preview mode creates a metafile that may be up to six times larger than when this mode is not set.

If *enhanced_preview* is not passed or if it is passed as false, text is not output to the metafile. Instead, a special comment record containing the text is written to the metafile, and this sets a flag on playback to output the text to the destination device using GDI commands. The difference is that the font is thereby scaled to the final device based on point size, rather than on how it fit into the original device context. Thus, if the devices are different in scale there will likely be a difference in the text size, due to rounding of the font point size. This difference is particularly noticeable in non-TrueType fonts, because they don't scale well anyway.

In summary, passing the flag as true ensures graphical precision of the text, whereas not passing it or passing it as false saves disk space on the metafile. For more information, see [“Print preview” on page 17-17](#).

DWP_BEGINJOB returns true if successful. We recommend that you call DWP_BEGINJOB *after* you set printer properties. Once DWP_BEGINJOB is called, you can only change characteristics within a page, such as font and color.

DWP_ENDJOB

Closes a document.

DWP_ENDJOB returns true if successful.

DWP_ADDFILE, *filename*

Sends the specified file to the current document. This subfunction starts on a new page and ends its last page. *Filename* cannot be a remote file specification using *x/Server*. Although we don't recommend using a remote file in this situation for performance reasons, if you must use one, you will need to use a UNC path to specify it.

DWP_ADDFILE returns the number of pages added.

DWP_BEGINPAGE

Starts a new page on the current document.

DWP_BEGINPAGE returns true if successful.

DWP_ENDPAGE

Ends the current page on the current document.

DWP_ENDPAGE returns true if successful.

DWP_WRITEOUT, *x*, *y*, *text*

Outputs text on the current page at the specified *x*, *y* pixel coordinates.

DWP_WRITEOUT returns the X pixel coordinate where *text* ends.

DWP_LINE, *pen_handle*, *startx*, *starty*, *endx*, *endy*

Draws a line to the specified device starting at *startx*, *starty* and ending at *endx*, *endy*. The *pen_handle* is returned by the DWP_GETPEN subfunction of %WPR_INFO. (For more information, see [page 17-30](#).)

DWP_LINE returns true if successful.

DWP_FILL, *left*, *top*, *width*, *height*, *color*[, *hatch*]

Fills a rectangular region of a report page with a color, which can be solid or hatched. Note that anything (text, graphics, etc.) that's in the fill region when %WPR_PRINT is called with DWP_FILL will be covered. If text is subsequently displayed over the filled region, the DWP_BACKCOLOR setting for %WPR_SETDEVICE determines the color of the background area of each text cell.

The region for the fill is defined with the following arguments. Note that if the rectangle coordinates are outside of the boundaries of the page, results will vary depending on the device.

- ▶ *left*, the leftmost x-axis pixel coordinate
- ▶ *top*, the topmost y-axis pixel coordinate
- ▶ *width*, the width, in pixels, of the region
- ▶ *height*, the height, in pixels, of the region

The fill color is defined by the *color* argument, which must be an RGB triplet as constructed by the M_RGB macro (in **DBLDIR:winprint.def**):

M_RGB (*red*,*green*,*blue*)

If *color* contains a non-zero value in bit 24 or higher, results will vary depending on the device.

By default the fill is solid, but you can instead fill the region with a pattern by passing one of the following hatch options as *hatch* (any other value will cause a runtime error). Note that the DWP_BACKCOLOR setting for %WPR_SETDEVICE determines the background for a hatch pattern.

DWPH_SOLID	Fill the entire rectangle with <i>color</i> .
DWPH_HORIZONTAL	Fill with horizontal hatch (----) of <i>color</i> .
DWPH_VERTICAL	Fill with vertical hatch () of <i>color</i> .
DWPH_FDIAGONAL	Fill with 45-degree downward diagonal hatch (\\) of <i>color</i> .
DWPH_BDIAGONAL	Fill with 45-degree upward diagonal hatch (///) of <i>color</i> .
DWPH_CROSS	Fill with horizontal/vertical crosshatch (++++) of <i>color</i> .
DWPH_DIAGCROSS	Fill with diagonal crosshatch (XXXX) of <i>color</i> .

The fill will not be transparent, even if *color* meets the settings passed to the %WPR_PRINT subfunction DWP_TRANSPARENCY.

DWP_FILL returns true if successful.

DWP_BITMAP, *graphic_file*, [*x*, *y*], [*cx*, *cy*][, *scale*]

Outputs a graphical image in the specified location on the current page. The following file formats are supported:

- ▶ BMP – Windows bitmap format
- ▶ GIF – Graphics Interchange Format
- ▶ JPEG – Joint Photographic Experts Group, stored in JFIF (JPEG File Interchange Format)
- ▶ Exif – Exchangeable image file
- ▶ PNG – Portable Network Graphics
- ▶ TIFF – Tag Image File Format

If the specified graphic file does not exist or is not one of the supported types, no image will be output. *Graphic_file* cannot be a remote file specification using *x/Server*. Although we don't recommend using a remote file in this situation for performance reasons, if you must use one, you will need to use a UNC path to specify it.

The optional *x* and *y* coordinates designate where the image will be placed on the page. If they are not supplied, the default position is used. The optional *cx* and *cy* arguments specify the desired height and width of the image. If they are not supplied, the actual size of the image is used. If *cx* and *cy* are passed, and *scale* is not passed or is false, the image is clipped. If *scale* is passed and is true, the image is scaled to the specified *cx*, *cy* dimensions. If you specify a size for the image that is larger than the actual size, and *scale* is not set to true, the excess space is black by default.

If a transparent color has been defined using the SYN_TRANSPARENT_COLOR environment variable, DWP_BITMAP will recognize that color as transparent. Any pixel in the image that corresponds to the designated transparent color will remain unpainted, revealing whatever else was printed in that same location on the page instead. For more information about transparent colors, see [SYN_TRANSPARENT_COLOR](#) in the “Environment Variables” chapter of *Environment Variables and System Options*.

DWP_BITMAP returns true if successful.



Because pixels are not universal, how an image appears is ultimately influenced by the printer driver. Thus, you may need to write a routine that sizes the image on the document, based on the pixel values returned by the driver with calls to the DWP_PIXELWIDTH and DWP_PIXELHEIGHT subfunctions of %WPR_GETDEVICE. Use those values in the routine to ensure the image is sized consistently. The routine must perform calculations to make the size fit to the desired portion of the page, with the *scale* arguments of DWP_BITMAP set to true. If you want to maintain the aspect ratio of the image, then scale appropriately.

Note that while most devices use the same pixel ratio horizontally and vertically, some don't. For these exceptions, get the DWP_PAPERWIDTH and DWP_PAPERLENGTH values and do some additional calculations to compute the skew ratio.

DWP_SETPAGES, *npagerange*, *pagerange*

Specifies a range of pages that can be selected for previewing and printing.

Npagerange is the number of page ranges contained in *pagerange*. *Pagerange* is an array of at least *npagerange* elements that contains the ranges of pages to print. Each entry in the array has the following form:

```
group pagerange ,i8
    from_page    ,i4
    to_page      ,i4
endgroup
```

The first page submitted to the Windows printing API (via DWP_BEGINPAGE) is treated as page 1.

DWP_SETPAGES returns true if successful or false if one or more of the specified page ranges is invalid. (An invalid page range is any range containing a negative number or where the end page is less than the start page.)

DWP_TRANSPARENCY, *transparent_color*[, *threshold*]

Specifies a transparent color to enable you to print an image over text and have the text show through, as well as a threshold for how closely to match the transparent color. Whether rendered in the previewer or the printed page, any portion of any image that qualifies as transparent will not be printed at all.

If this subfunction is not called, color transparency settings will be set to those specified by the SYN_TRANSPARENT_COLOR and SYN_TRANSPARENCY_THRESHOLD environment variables or their corresponding default values (none and 0, respectively).

Transparent_color is a color to treat as transparent. It can be one of the following values:

DWP_TRANSPARENT_NONE (-1)

No color is treated as transparent and *threshold* is ignored.

DWP_TRANSPARENT_DEFAULT (-2)

The transparent color defaults to any value specified in SYN_TRANSPARENT_COLOR, and *threshold* is ignored, reverting to any value specified in SYN_TRANSPARENCY_THRESHOLD.

rgb

The RGB triplet for the color you want to designate as transparent. This is an integer value. Only the low-order 24 bits are considered.

Threshold is an optional value that can be added to or subtracted from each red, green, and blue component of the transparent color to constitute an acceptable transparency range. It can be one of the following values:

DWP_TRANSPARENT_NONE (-1) or 0

The transparent color must match exactly within the image. (default)

n

The threshold above or below each RGB component of the transparent color to accept as within the transparency range, where *n* is an integer value between 0 and 255. Only the low-order 8 bits of this value are used and treated as unsigned.

You can call this subfunction at any time for a valid report handle and the new settings will take effect for the entire report. Only one color can be designated as transparent, and the specified color will be transparent in all images that contain it; you cannot have different settings for different images in the same report.

Note that the transparency settings do not apply to the color of text (DWP_TEXTCOLOR) or its background (DWP_BACKCOLOR). Transparency only applies to images (DWP_BITMAP).

Examples

See “Sample programs” on page 17-7.

%WPR_SETDEVICE – Set printer settings

WIN

status = %WPR_SETDEVICE(*report_handle*, *capability*, *setting*)

Return value

status

One of the following values: (**n**)

true The printer supports the specified capability.

false The printer was not set to the requested setting.

Arguments

report_handle

The report handle returned by the DWP_GETPRINTER subfunction of %WPR_INFO. (See [page 17-30](#).) (**n**)

capability

One of the following printer capabilities, which are described in the Discussion section on [page 17-20](#): (**n**)

DWP_ORIENTATION

DWP_PAPERSIZE

DWP_PAPERLENGTH

DWP_PAPERWIDTH

DWP_SCALE

DWP_COPIES

DWP_QUALITY

DWP_COLOR

DWP_DUPLEX

DWP_COLLATE

DWP_FORMNAME

DWP_FONT

DWP_TEXTCOLOR

DWP_BACKCOLOR

DWP_PAPERSOURCE

DWP_SPECS

setting

The value to which you want to set the printer capability. (**a** or **n**)

Discussion

See the Discussion section for %WPR_GETDEVICE, which begins on [page 17-20](#). For %WPR_SETDEVICE, each of the applicable capabilities listed in that section sets (rather than gets) the specified property. See the Arguments section above for a list of applicable capabilities.

Although all printer capabilities are included in the **print_specs** structure shown on [page 17-25](#) (passed by DWP_SPECS), only the following can be modified using %WPR_SETDEVICE: **orientation**, **paper_size**, **paper_length**, **paper_width**, **scale**, **copies**, **quality**, **color**, **duplex**, **collate**, **formname**, and **paper_source**. Any settings that you change must be supported by your printer driver.

For information on duplex printing, refer to the *Migrating Your Application to Windows* document, available on the Online Manuals CD.

Custom paper sizes

If your printer driver allows you to select custom paper sizes, you can do so as follows:

1. Use %WPR_GETDEVICE to get the DWP_SPECS for the device.
 2. Clear the **paper_size** member of the returned structure.
 3. Set the **paper_length** and **paper_width** members to the size of the desired custom form. (The specified paper length and paper width must match a supported paper size; you cannot make up your own paper dimensions unless your printer driver supports it.)
 4. Use %WPR_SETDEVICE with DWP_SPECS using the modified structure.
-

18

Synergy XML API

Synergy XML API 18-4

Describes the Synergy XML API, which gives you direct access to XML from within your Synergy programs. This API includes the following routines:

%XML_ATTR_COPY – Copy from one XML attribute to another	18-19
%XML_ATTR_CREATE – Create an XML attribute.....	18-20
%XML_ATTR_DELETE – Release an XML attribute	18-21
%XML_ATTR_GETNAME – Get the name of an XML attribute	18-22
%XML_ATTR_GETVALUE – Get the value of an XML attribute.....	18-23
%XML_ATTR_GETVALUEHANDLE – Return the handle for an XML attribute’s value	18-24
%XML_ATTR_SETNAME – Set the name of an XML attribute.....	18-26
%XML_ATTR_SETVALUE – Set the value of an XML attribute	18-27
%XML_ATTR_SETVALUEHANDLE – Assign a handle as an XML attribute’s value	18-28
%XML_ATTRLIST_ADD – Add an attribute to an XML attribute list.....	18-30
%XML_ATTRLIST_ADDLIST – Add a list of attributes to an XML attribute list	18-31
%XML_ATTRLIST_CLEAR – Clear XML attribute list contents	18-33
%XML_ATTRLIST_COPY – Copy attributes from one XML attribute list to another	18-34
%XML_ATTRLIST_COUNT – Count the attributes in an XML attribute list.....	18-35
%XML_ATTRLIST_CREATE – Create an XML attribute list	18-36
%XML_ATTRLIST_DELETE – Release an XML attribute list.....	18-37
%XML_ATTRLIST_FIND – Find an attribute in an XML attribute list by name.....	18-38
%XML_ATTRLIST_INSERT – Insert an attribute at a specific position in a list	18-39
%XML_ATTRLIST_ITEM – Find an attribute in an XML attribute list by position	18-41
%XML_ATTRLIST_REMOVE – Remove an attribute from an XML attribute list	18-42
%XML_DOC_ADDDECLARATION – Add a declaration to an XML document.....	18-43
%XML_DOC_CREATE – Create an XML document instance	18-45
%XML_DOC_DELETE – Release an XML document instance.....	18-46
%XML_DOC_GETDECLARATION – Get the declaration for an XML document.....	18-47

%XML_DOC_GETROOT – Return the root element of an XML document.....	18-48
%XML_DOC_REMOVEDECLARATION – Remove a declaration from an XML document.....	18-49
%XML_DOC_SETCOMMENT – Set a comment at the XML document level	18-50
%XML_DOC_SETDECLARATION – Set the declaration for an XML document.....	18-51
%XML_DOC_SETROOT – Set the root element of an XML document	18-52
%XML_DOC_TOFILE – Write contents of an XML document to a file	18-53
%XML_DOC_TOSTRING – Write contents of an XML document to an XML string	18-54
%XML_ELEM_ADDATTRIBUTES – Add attributes in a list to an XML element	18-55
%XML_ELEM_ADDCHILD – Add a child to an XML element.....	18-57
%XML_ELEM_ADDCHILDREN – Add child elements to an XML element	18-58
%XML_ELEM_ATTRIBUTES – Return attributes for an XML element	18-60
%XML_ELEM_CHILDREN – Get children of an XML element.....	18-61
%XML_ELEM_COPY – Copy from one XML element to another	18-62
%XML_ELEM_CREATE – Create an XML element instance	18-63
%XML_ELEM_DELETE – Release an XML element instance.....	18-64
%XML_ELEM_GETATTRIBUTE – Get the value of an attribute for an XML element.....	18-65
%XML_ELEM_GETELEMENTSBYNAME – Get children of an XML element by tag	18-66
%XML_ELEM_GETNAME – Get the tag name of an XML element	18-67
%XML_ELEM_GETTEXT – Get the text value of an XML element.....	18-68
%XML_ELEM_GETTEXTHANDLE – Return handle containing text for an element.....	18-69
%XML_ELEM_REMOVEATTRIBUTE – Remove an attribute from an XML element.....	18-71
%XML_ELEM_REMOVEATTRIBUTES – Remove all attributes from an XML element....	18-72
%XML_ELEM_REMOVECHILD – Remove the first child specified from an element	18-73
%XML_ELEM_REMOVECHILDREN – Remove all child elements from an element.....	18-74
%XML_ELEM_SETATTRIBUTE – Add an attribute to an XML element.....	18-75
%XML_ELEM_SETNAME – Set the tag name of an XML element	18-76
%XML_ELEM_SETTEXT – Set the text value of an XML element.....	18-77
%XML_ELEM_SETTEXTHANDLE – Assign a handle as the text for an XML element	18-78
%XML_ELEM_TOSTRING – Give the string representation for an XML element	18-80
%XML_ELEMLIST_ADD – Add an element to an XML element list.....	18-81
%XML_ELEMLIST_ADDLIST – Add a list of elements to an XML element list.....	18-82
%XML_ELEMLIST_CLEAR – Clear XML element list contents.....	18-84
%XML_ELEMLIST_COPY – Copy an XML element list to another list.....	18-85

%XML_ELEMLIST_COUNT – Count the elements in an XML element list.....	18-86
%XML_ELEMLIST_CREATE – Create an XML element list.....	18-87
%XML_ELEMLIST_DELETE – Release an XML element list.....	18-88
%XML_ELEMLIST_FIND – Find an element in an XML element list by name.....	18-89
%XML_ELEMLIST_FINDTAG – Find an element in an XML element list by tag	18-90
%XML_ELEMLIST_INSERT – Insert an XML element at a specific position in a list.....	18-91
%XML_ELEMLIST_ITEM – Find an element in an XML element list by position	18-93
%XML_ELEMLIST_REMOVE – Remove an element from an XML element list	18-94
%XML_OPTION – Set an XML option value.....	18-95
%XML_PARSER_CREATE – Create an XML parser instance	18-97
%XML_PARSER_DELETE – Release an XML parser instance.....	18-98
%XML_PARSER_ERROR – Get the last XML parser error	18-99
%XML_PARSER_PARSEFILE – Parse an XML file	18-101
%XML_PARSER_PARSESTRING – Parse an XML string.....	18-102
%XML_STRING_APPEND – Append a Synergy alpha string to an XML string.....	18-103
%XML_STRING_APPENDDYN – Append one XML string to another.....	18-104
%XML_STRING_APPENDHANDLE – Append Synergy handle contents to an XML string.....	18-105
%XML_STRING_CLEAR – Clear the contents of an XML string.....	18-106
%XML_STRING_COPY – Copy one XML string to another	18-107
%XML_STRING_CREATE – Create an XML string instance.....	18-108
%XML_STRING_DELETE – Release an XML string instance	18-109
%XML_STRING_GETHANDLE – Get the memory handle of an XML string.....	18-110
%XML_STRING_GETSIZE – Get the size of an XML string	18-111

Synergy XML API

XML (Extensible Markup Language) is a standard that specifies a syntax that enables you to create your own markup language. It wraps all data in tags that identify what each piece of data represents. As the name suggests, you can define an unlimited number of tags to define the data in the XML document. XML looks similar to HTML in that it employs the start-tag/end-tag mechanism to delimit the data.

The Synergy XML API gives you direct access to XML from within your Synergy programs. It enables you to

- ▶ parse an XML file or string into a memory-based XML document that represents a DOM tree. (See “DOM trees” on page 18-6 for more information.) Parsing the file means to disassemble the XML so that your Synergy program can process it.
- ▶ iterate through the DOM tree to access the XML data. This entails finding the root element, being able to iterate recursively through its children, and being able to get text and attributes for all these elements.
- ▶ assemble Synergy data into a memory-based XML document. This includes being able to specify an element’s children, attributes, and text values.
- ▶ write the contents of a memory-based XML document to a file or XML string.

XML documents

Each XML document, whether in memory or in a file or string, has one root element. All document content is defined by this root element and its children. An element is represented by a start tag and an end tag delimiting its text and/or children. Between the tags, each element may have zero or more attributes that further describe the element. In addition, each element may have zero or more child elements. These child elements also have their own attributes, text values, and/or children.

Here’s an example of XML describing the weather in Seattle:

```
<?xml version="1.0"?>
<weather-report>
  <date>March 25, 1998</date>
  <time>08:00</time>
  <area>
    <city>Seattle</city>
    <state>WA</state>
    <region>West Coast</region>
    <country>USA</country>
  </area>
  <measurements>
    <skies>partly cloudy</skies>
    <temperature>46</temperature>
    <wind>
```

```

        <direction>SW</direction>
        <windspeed>6</windspeed>
    </wind>
    <humidity>87</humidity>
    <visibility>10</visibility>
</measurements>
</weather-report>

```

A well-formed XML document satisfies the following rules:

- ▶ It must have a single, unique root element.
- ▶ Every element must have a start tag and an end tag.
- ▶ Element and attribute names are case sensitive and cannot contain spaces.
- ▶ Elements must be properly nested; they cannot have structural overlaps.
- ▶ Certain characters must be escaped or represented by a combination of characters.
- ▶ Attribute values must be enclosed in quotation marks.
- ▶ Empty elements have a special form to which they must adhere.



Creating or processing large (multi-megabyte) XML documents requires an even larger amount of memory than the original document. If you think there is the potential to run out of memory, you must use an `ONERROR($ERR_CATCH)` to trap the “Not enough memory for desired operation” error (`$ERR_NOMEM`). If you want to simulate an out-of-memory condition, you can set the `MAXMEMMAX` environment variable to limit the amount of memory the runtime will use.

Embedding an XML document

You can embed an XML document within another XML document using the CDATA syntax. Because the text in a CDATA node is treated as plain text by the parser, instead of being processed as XML, CDATA is used to escape text that would otherwise be recognized as markup. For example, you might define an XML element as a CDATA node if your text contains a lot of “<” and “&” characters.

A CDATA node is formatted like this:

```

<node_name>
<![CDATA[embedded XML here]]>
</node_name>

```

To access the CDATA data node from the XML document, execute `%XML_ELEM_GETTEXT` on the parent node that contains CDATA.



The XML API does not support embedding CDATA within the text stream. For example, `<mynode>my text<![CDATA[more text]]>more text</mynode>` is not supported.

DOM trees

An XML parser is used to load an XML document into the memory of your computer. Parsing the document produces a tree view of the document that can be traversed to retrieve and manipulate the XML data. This tree view of the XML document is called a DOM (Document Object Model) tree. The document's root element is the top level of the tree. This element can have one or many child elements, or nodes, that represent the branches of the tree. Using a DOM, you can create a document, navigate its structure, and add, modify, or delete its elements.

Figure 18-1 shows what the following XML document would look like as a DOM tree:

```
<structure name="mystruct" size="284" >  
  <field name="id" type="decimal" size="4" />  
  <field name="description" type="alpha" size="250" />  
  <field name="username" type="alpha" size="30" />  
</structure>
```

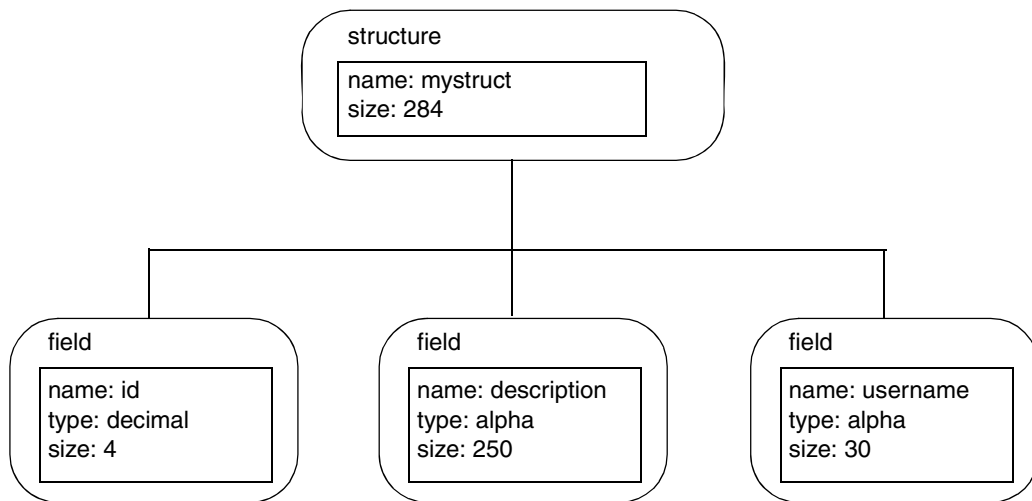


Figure 18-1. A DOM tree of an XML document.

In this example the topmost tag, **structure**, becomes the root element. This structure element has two attributes: name and size. Below it are the three child elements representing the three field tags. Each of these children in turn has three attributes: name, type, and size.

XML API routines

The Synergy XML API routines are documented on [pages 18-20](#) through [18-111](#). Each routine can be called as either a subroutine or a function; however, if you call it as a subroutine you will not be able to access its return value.

To use the XML API, you must

WIN, UNIX

- ▶ Include the **DBLDIR:synxml.def** file in your Synergy program.
 - ▶ Link the **DBLDIR:synxml.elb** library into your Synergy program.
-

VMS

- ▶ Link your program as follows:

```
link program, sys$share:synrtl/opt, sys$input:/opt
SYNXML/share
^Z
```

Alternatively, you can copy **synrtl.opt** to a local directory and add SYNXML/SHARE to your options file.

All programs that use the XML API must be built using the **-T** or **/TRIM** compiler option (which trims trailing null arguments from XCALLs and function references), and the **-X** or **/IMPLICIT** compiler option (which automatically declares all undefined function references as external ^VAL functions).

XML parser routines

Before you can parse and manipulate an XML document, you need an XML parser. The `%XML_PARSER_xxx` routines create a parser and parse an XML file or string into memory as a DOM tree. They include

```
%XML_PARSER_CREATE
%XML_PARSER_DELETE
%XML_PARSER_ERROR
%XML_PARSER_PARSEFILE
%XML_PARSER_PARSESTRING
```

Here's how you would typically use them:

1. Call `%XML_PARSER_CREATE` to create the parser.
2. Call either `%XML_PARSER_PARSEFILE` or `%XML_PARSER_PARSESTRING`, depending on whether the XML arrives as a physical file or an in-memory handle in a Synergy program. Both of these functions return an XML document that can be processed using the XML document routines (`%XML_DOC_`*xxx*).
3. Use `%XML_DOC_DELETE` to delete the document returned by `%XML_PARSER_PARSEFILE` or `%XML_PARSER_PARSESTRING`.
4. Call `%XML_PARSER_DELETE` to release the parser.

XML document routines

The %XML_DOC_xxx routines enable you to manipulate the in-memory DOM tree representation of an XML document. They include

%XML_DOC_ADDDECLARATION
%XML_DOC_CREATE
%XML_DOC_DELETE
%XML_DOC_GETDECLARATION
%XML_DOC_GETROOT
%XML_DOC_REMOVEDDECLARATION
%XML_DOC_SETCOMMENT
%XML_DOC_SETDECLARATION
%XML_DOC_SETROOT
%XML_DOC_TOFILE
%XML_DOC_TOSTRING

Here's how you would typically use them:

Once an XML document has been parsed using the %XML_PARSER_xxx routines

1. Get the document's root element using %XML_DOC_GETROOT.
2. Process the root element using the XML element routines (%XML_ELEM_xxx).
3. (Optional) Depending on the needs of your Synergy application, write the XML document to an XML file using %XML_DOC_TOFILE or to an XML string using %XML_DOC_TOSTRING.

If you are assembling Synergy data in XML format

1. Create an XML document with %XML_DOC_CREATE.
2. Designate the root element with %XML_DOC_SETROOT.
3. Process the root element (and subelements) using the XML element routines (%XML_ELEM_xxx).
4. Depending on the needs of your Synergy application, write the XML document to an XML file using %XML_DOC_TOFILE or to an XML string using %XML_DOC_TOSTRING.
5. Release the XML document using %XML_DOC_DELETE.

XML element routines

The `%XML_ELEM_`*xxx* routines enable you to manipulate an XML element. They include

`%XML_ELEM_ADDATTRIBUTES`
`%XML_ELEM_ADDCHILD`
`%XML_ELEM_ADDCHILDREN`
`%XML_ELEM_ATTRIBUTES`
`%XML_ELEM_CHILDREN`
`%XML_ELEM_COPY`
`%XML_ELEM_CREATE`
`%XML_ELEM_DELETE`
`%XML_ELEM_GETATTRIBUTE`
`%XML_ELEM_GETELEMENTSBYNAME`
`%XML_ELEM_GETNAME`
`%XML_ELEM_GETTEXT`
`%XML_ELEM_GETTEXTHANDLE`
`%XML_ELEM_REMOVEATTRIBUTE`
`%XML_ELEM_REMOVEATTRIBUTES`
`%XML_ELEM_REMOVECHILD`
`%XML_ELEM_REMOVECHILDREN`
`%XML_ELEM_SETATTRIBUTE`
`%XML_ELEM_SETNAME`
`%XML_ELEM_SETTEXT`
`%XML_ELEM_SETTEXTHANDLE`
`%XML_ELEM_TOSTRING`

You can get the following information from an element:

- ▶ An attribute value, using `%XML_ELEM_GETATTRIBUTE`
- ▶ A text value, using `%XML_ELEM_GETTEXT`
- ▶ The element's children, using either `%XML_ELEM_CHILDREN` or `%XML_ELEM_GETELEMENTSBYNAME`

You can also set an attribute value with `%XML_ELEM_SETATTRIBUTE`, or set its text value with `%XML_ELEM_SETTEXT`.

XML attribute list routines

An attribute list is simply a list of attributes for a given element.

The %XML_ATTRLIST_xxx routines enable you to manipulate XML attribute lists. They include

%XML_ATTRLIST_ADD
%XML_ATTRLIST_ADDLIST
%XML_ATTRLIST_CLEAR
%XML_ATTRLIST_COPY
%XML_ATTRLIST_COUNT
%XML_ATTRLIST_CREATE
%XML_ATTRLIST_DELETE
%XML_ATTRLIST_FIND
%XML_ATTRLIST_INSERT
%XML_ATTRLIST_ITEM
%XML_ATTRLIST_REMOVE

Usually when you retrieve information about an attribute, you can use %XML_ELEM_GETATTRIBUTE and avoid calling these functions directly. Indirectly, %XML_ELEM_GETATTRIBUTE calls %XML_ATTRLIST_FIND to find the attribute by name and then calls %XML_ATTR_GETVALUE to get the attribute's value. However, you may also choose to iterate through an element's attributes:

1. To get all of the attributes for an element, call %XML_ELEM_ATTRIBUTES, which returns an attribute list instance.
2. Call %XML_ATTRLIST_COUNT to get the number of attributes in the list.
3. Use a Synergy FOR loop to get each attribute in the list using %XML_ATTRLIST_ITEM.

XML attribute routines

The %XML_ATTR_xxx routines enable you to manipulate a single XML attribute. They include

%XML_ATTR_COPY
%XML_ATTR_CREATE
%XML_ATTR_DELETE
%XML_ATTR_GETNAME
%XML_ATTR_GETVALUE
%XML_ATTR_GETVALUEHANDLE
%XML_ATTR_SETNAME
%XML_ATTR_SETVALUE
%XML_ATTR_SETVALUEHANDLE

Use these routines to get or set the attribute's name or value. (In most cases, however, you can simply use %XML_ELEM_SETATTRIBUTE to add or update element attributes.)

XML element list routines

The %XML_ELEMLIST_xxx routines enable you to manipulate a list of XML elements. They include

%XML_ELEMLIST_ADD
%XML_ELEMLIST_ADDLIST
%XML_ELEMLIST_CLEAR
%XML_ELEMLIST_COPY
%XML_ELEMLIST_COUNT
%XML_ELEMLIST_CREATE
%XML_ELEMLIST_DELETE
%XML_ELEMLIST_FIND
%XML_ELEMLIST_FINDTAG
%XML_ELEMLIST_INSERT
%XML_ELEMLIST_ITEM
%XML_ELEMLIST_REMOVE

1. Call either %XML_ELEM_CHILDREN or %XML_ELEM_GETELEMENTSBYNAME to obtain a list of elements.
2. Call %XML_ELEMLIST_COUNT to get the number of elements in the list.
3. Use a Synergy FOR loop to iterate through the list, getting each element by calling %XML_ELEMLIST_ITEM.

You can also iterate through a list of elements looking for an element by name by calling %XML_ELEMLIST_FIND, or by tag by calling %XML_ELEMLIST_FINDTAG.

XML string routines

The %XML_STRING_xxx routines enable you to manipulate a dynamic string. This string enables the XML API to handle strings of any size, even those greater than the 65,535 limitation. You can append a regular Synergy alpha string using %XML_STRING_APPEND, append another dynamic string using %XML_STRING_APPENDDDYN, or append a handle to the XML string using %XML_STRING_APPENDHANDLE.

When processing XML data in a Synergy program, you will typically need to change the XML string into a Synergy memory handle. The %XML_STRING_GETHANDLE routine provides the handle, and the %XML_STRING_GETSIZE routine provides its size.

If XML data arrives in a Synergy program as in-memory data, such as an alpha string or a D_ADDR handle returned from the HTTP document transport API, in order to parse that data you must first create an XML string using %XML_STRING_CREATE and then call the appropriate append routine to add that data to the XML string. Then the XML string can be passed into %XML_PARSER_PARSESTRING. Likewise, going the other direction, if you have an XML document, you can call %XML_DOC_TOSTRING to first get an XML string and then call %XML_STRING_GETHANDLE to pass it into the HTTP API routines.

The XML string routines include

`%XML_STRING_APPEND`
`%XML_STRING_APPENDDDYN`
`%XML_STRING_APPENDHANDLE`
`%XML_STRING_CLEAR`
`%XML_STRING_COPY`
`%XML_STRING_CREATE`
`%XML_STRING_DELETE`
`%XML_STRING_GETHANDLE`
`%XML_STRING_GETSIZE`

XML option routine

The `%XML_OPTION` routine sets a value for an XML option.

Parsing and processing XML data

This section describes the basic steps you need to take to parse (or disassemble) an existing XML document sent to your Synergy application and process the data so your application can use it. The document can be in the form of a Synergy alpha string or a file.

1. Create a Synergy XML parser by calling `%XML_PARSER_CREATE`.
2. Parse the XML document by calling `%XML_PARSER_PARSEFILE`.
3. Get the root element by calling `%XML_DOC_GETROOT`.
4. Iterate through the DOM tree, beginning with the root element from the document created in [step 3](#). You can iterate through the tree in many different paths using `%XML_ELEM_CHILDREN`, `%XML_ELEM_GETELEMENTSBYNAME`, and the `%XML_ELEMLIST_XXX` routines.
5. Retrieve XML data by getting the element's attributes, associated text, and child attributes using `%XML_ELEM_GETATTRIBUTE`, `%XML_ELEM_GETTEXT`, and `%XML_ELEM_CHILDREN`, respectively.
6. Repeat [step 4](#) and [step 5](#) as needed for all elements in the XML document.
7. Use the XML data as desired in your application.
8. After parsing is done, release the parser by calling `%XML_PARSER_DELETE`. (You can perform this step any time after [step 2](#), when parsing is complete.)
9. After the document is no longer needed, release it by calling `%XML_DOC_DELETE`.

Example of parsing XML

The following example uses the Synergy XML API to parse an XML file. For more examples, see the CodeExchange at <http://www.synergex.com>.

```
.include "DBLDIR:synxml.def"
main
.align

record
    i            ,i4
    parser       ,XML_PARSER_TYPE
    doc          ,XML_DOC_TYPE
    root         ,XML_ELEM_TYPE
    elem         ,XML_ELEM_TYPE
    children     ,XML_ELEMLIST_TYPE
    tagName      ,a250
    attrValue    ,a250
    msg          ,a250

proc

    ; Create parser instance

    parser = %xml_parser_create()

    ; Parse the xml file

    doc = %xml_parser_parseFile(parser,"hey.xml")
    if .not.(doc)
        begin
            xcall xml_parser_error(parser,msg)
            xcall test_writeit("error:"+%atrim(msg))
            goto done
        end

    ; Get the root element

    root = %xml_doc_getRoot(doc)
    if .not.(root)
        begin
            xcall test_writeit("error:empty root")
            goto done
        end
    end
```

```

; Print out the tag name and the attribute name value

xcall xml_elem_getName(root, tagName)
xcall xml_elem_getAttribute(root, "name", attrValue)
xcall test_writeit("root:"+%atrim(tagName)+":"+%atrim(attrValue))

; Get the children of the root

children = %xml_elem_children(root)

; Iterate through the list of children

for i from 1 thru %xml_elemlist_count(children)
  begin
    elem = %xml_elemlist_item(children, i)
    if .not.(elem)
      begin
        xcall test_writeit("error:empty element for index "+%string(i))
        goto done
      end

    ; Print out the tag name and the attribute name value

    xcall xml_elem_getName(elem, tagName);
    xcall xml_elem_getAttribute(elem, "name", attrValue)
    xcall test_writeit("  child:"+%atrim(tagName)+":"+
&                                     %atrim(attrValue))
    end

done, if (parser)
  xcall xml_parser_delete(parser) ;Delete the parser instance
if (doc)
  xcall xml_doc_delete(doc)      ;Delete the XML document
  sleep(3)
endmain

; write a message to standard out

subroutine test_writeit
  msg      , a

proc
  open(1, o, 'tt:')
  writes(1, %atrim(msg))
  close 1
  xreturn
endsubroutine

```

Assembling Synergy data in XML format

This section contains the basic steps required to create a well-formed XML document using Synergy program data for use by an external system. The document may be represented as a Synergy memory handle or as a physical file on the hard disk.

1. Create the XML document by calling `%XML_DOC_CREATE`.
2. Create the root element of the XML document by calling `%XML_ELEM_CREATE`.
3. Specify that the element created in [step 2](#) is the root element of the XML document by calling `%XML_DOC_SETROOT`.
4. Name the element using `%XML_ELEM_SETNAME`, and assign any necessary attributes using `%XML_ELEM_SETATTRIBUTE`.
5. Do the following for each subelement:
 - ▶ Create each subelement using `%XML_ELEM_CREATE`.
 - ▶ Specify the subelement's name and attributes by calling `%XML_ELEM_SETNAME` and `%XML_ELEM_SETATTRIBUTE`, respectively.
 - ▶ Add each subelement to the parent element by calling `%XML_ELEM_ADDCHILD`.
6. Write the XML document into an XML file by calling `%XML_DOC_TOFILE`.
7. If you need to write the XML document to a Synergy memory handle, create an XML string by calling `%XML_DOC_TOSTRING`. Get the handle and size of the XML string using `%XML_STRING_GETHANDLE` and `%XML_STRING_GETSIZE`, respectively.
8. Release the XML document by calling `%XML_DOC_DELETE`.

Example of assembling XML

The example below shows how to build the DOM tree for the structure example on [page 18-6](#). It writes the XML to a file called **hey.xml**. For more examples, see the CodeExchange at <http://www.synergex.com>.

```
.include "DBLDIR:synxml.def"

main
.align
record
    doc          , XML_DOC_TYPE
    root         , XML_ELEM_TYPE
    f1           , XML_ELEM_TYPE
    f2           , XML_ELEM_TYPE
    f3           , XML_ELEM_TYPE
```

```
proc
    doc = %xml_doc_create
    if (doc)
    begin
        root = %xml_elem_create
        xcall xml_doc_setRoot(doc,root)
        xcall xml_elem_setName(root,"structure")
        xcall xml_elem_setAttribute(root,"name","mystruct")
        xcall xml_elem_setAttribute(root,"size","284")
        f1 = %xml_elem_create
        xcall xml_elem_setName(f1,"field")
        xcall xml_elem_setAttribute(f1,"name","id")
        xcall xml_elem_setAttribute(f1,"type","decimal")
        xcall xml_elem_setAttribute(f1,"size","4")
        xcall xml_elem_addChild(root,f1)
        f2 = %xml_elem_create
        xcall xml_elem_setName(f2,"field")
        xcall xml_elem_setAttribute(f2,"name","description")
        xcall xml_elem_setAttribute(f2,"type","alpha")
        xcall xml_elem_setAttribute(f2,"size","250")
        xcall xml_elem_addChild(root,f2)
        f3 = %xml_elem_create
        xcall xml_elem_setName(f3,"field")
        xcall xml_elem_setAttribute(f3,"name","username")
        xcall xml_elem_setAttribute(f3,"type","alpha")
        xcall xml_elem_setAttribute(f3,"size","30")
        xcall xml_elem_addChild(root,f3)
        xcall xml_doc_toFile(doc,"hey.xml")
        xcall xml_doc_delete(doc)
    end
endmain
```

Synergy XML API defines

The XML API uses the following type defines:

For this XML instance	The type define is
Attribute	XML_ATTR_TYPE
Attribute list	XML_ATTRLIST_TYPE
Document	XML_DOC_TYPE
Element	XML_ELEM_TYPE

For this XML instance	The type define is
Element list	XML_ELEMLIST_TYPE
Parser	XML_PARSER_TYPE
String	XML_STRING_TYPE

For instance, you would pass XML_ELEM_TYPE handles to XML_ELEM_xxx routines and XML_ATTR_TYPE handles to XML_ATTR_xxx routines. You would not pass an XML_ATTR_TYPE handle to XML_ELEM_GETNAME, for example. If the wrong type is passed, an “Invalid external function data type” error (\$ERR_INVEXFTYP) is generated.

The XML API also has a status define, XML_SUCCESS, that indicates an operation was successful. Many XML API functions return XML_SUCCESS on successful completion; they do not return this value if the operation wasn’t successful. We suggest you check your code as follows:

```
status = %xml_attr_setValue(attrid, "myname")
if (XML_SUCCESS .eq. status) then
    ; Operation was successful
else
    ; Operation was unsuccessful -- do error handling
```

Likewise, failure can be tested by

```
status .ne. XML_SUCCESS
```


%XML_ATTR_COPY – Copy from one XML attribute to another

status = %XML_ATTR_COPY (*destid*, *sourceid*)

Return value

status

XML_SUCCESS if the copy is successful.

Arguments

destid

The destination attribute ID. (XML_ATTR_TYPE)

sourceid

The source attribute ID. (XML_ATTR_TYPE)

Discussion

The %XML_ATTR_COPY function copies the name and value from a source XML attribute to a destination XML attribute. The element object handle in *destid* must already exist before %XML_ATTR_COPY is called; it is not created automatically.

This function is defined as ^VAL.

%XML_ATTR_CREATE – Create an XML attribute

attr = %XML_ATTR_CREATE

Return value

attr

The XML attribute instance created, or 0 if not successful. (XML_ATTR_TYPE)

Discussion

An XML attribute is a characteristic of an XML element. For example, an attribute of a field element might be name or size.

Most of the time it is easier to use %XML_ELEM_SETATTRIBUTE, which defines an attribute's name and value and assigns the attribute to an XML element, rather than %XML_ATTR_CREATE. If you create an attribute with %XML_ATTR_CREATE, you must also set its name and value with %XML_ATTR_SETNAME and %XML_ATTR_SETVALUE and then add it to an element's attribute list using %XML_ATTRLIST_ADD.

Attributes explicitly created with this routine and not added as data to a parent element will need to be deleted using %XML_ATTR_DELETE.

This function is defined as ^VAL.

%XML_ATTR_DELETE – Release an XML attribute

```
status = %XML_ATTR_DELETE (attr)
```

Return value

status

XML_SUCCESS if the attribute is successfully released.

Arguments

attr

The XML attribute instance to release. (XML_ATTR_TYPE)

Discussion

You will rarely need to call this function, because the %XML_DOC_DELETE function releases all memory for an XML document, including all attributes associated with the document. The only time you will need to release an individual attribute is when the attribute was explicitly created with %XML_ATTR_CREATE and was not added as data to a parent element (and therefore would not be released as part of a document or list).

Note that while %XML_ATTR_DELETE releases the attribute's memory, it does not automatically remove the attribute from any attribute lists of which it is a member.

This function is defined as ^VAL.

%XML_ATTR_GETNAME – Get the name of an XML attribute

status = %XML_ATTR_GETNAME (*attr*, *name*)

Return value

status

XML_SUCCESS if a nonblank attribute name is returned.

Arguments

attr

The XML attribute instance to return a name for. (**XML_ATTR_TYPE**)

name

Returned with the name of the XML attribute specified by *attr*. The maximum length is 256 characters. (**a**)

Discussion

This function is defined as ^VAL.

%XML_ATTR_GETVALUE – Get the value of an XML attribute

status = %XML_ATTR_GETVALUE (*attr*, *value*[, *length*])

Return value

status

XML_SUCCESS if a nonblank value is returned.

Arguments

attr

The XML attribute instance to get a value for. (XML_ATTR_TYPE)

value

Returned with the value of the XML attribute. The maximum length is 65,022 characters. (a)

length

(optional) Returned with the length of the returned attribute value. The minimum *length* is 1, even if *value* is blank. (n)

Discussion

This function is defined as ^VAL.



If you want to receive variable-length data in a large buffer, we recommend using the *length* argument, as this will avoid time spent clearing the remainder of the buffer (beyond *length*) with spaces.

See also

[%XML_OPTION](#) on page 18-95.

%XML_ATTR_GETVALUEHANDLE – Return the handle for an XML attribute's value

```
handle = %XML_ATTR_GETVALUEHANDLE (attr)
```

Return value

handle

The handle that contains the value for the specified attribute, or 0 if no value is assigned.
(**D_ADDR**)

Arguments

attr

The XML attribute instance to get the value handle for. (**XML_ATTR_TYPE**)

Discussion

%XML_ATTR_GETVALUEHANDLE returns the memory handle that contains the value of an XML attribute.

Since %XML_ATTR_GETVALUEHANDLE may be used instead of %XML_ATTR_GETVALUE for performance reasons, %XML_ATTR_GETVALUEHANDLE does not honor %XML_OPTION with regard to unescaping attribute values. If you use %XML_ATTR_GETVALUEHANDLE, unescaping certain characters (for example, ampersands and double quotation marks) is your responsibility. You can unescape characters using %SYN_UNESCAPE_HANDLE after calling this method. See [%SYN_UNESCAPE_HANDLE on page 9-336](#) for more information.



If you call %XML_ATTR_GETVALUEHANDLE and then do an unescape on the handle, any document that contains that attribute will be modified. Therefore, if you don't want to modify your document, you should make a copy of the handle before performing any escaping or unescaping.

This function is defined as ^VAL.

Examples

The following example returns the value for the **Starbucks grind** attribute as a handle.

```
doc      ,XML_DOC_TYPE
attr     ,XML_ATTR_TYPE
root     ,XML_ELEM_TYPE
elem     ,XML_ELEM_TYPE
attrlist ,XML_ATTRLIST_TYPE
```

```

children ,XML_ELEMLIST_TYPE
h          ,d_addr

root = %xml_doc_getroot(doc)
children = %xml_elem_children(root)

elem = %xml_elemlist_findtag(children, "Starbucks")
attrlist = %xml_elem_attributes(elem)
attr = %xml_attrlist_find(attrlist, "grind")

open(1,o, "outfile.txt")

h = %xml_attr_getvaluehandle(attr)

if (h) then
  writes(1, ^m(h))
else
  writes(1, "Unable to get value handle for attribute")
close 1

```

%XML_ATTR_SETNAME – Set the name of an XML attribute

status = %XML_ATTR_SETNAME (*attr*, *name*)

Return value

status

XML_SUCCESS if the name is set successfully.

Arguments

attr

The XML attribute instance to name. (**XML_ATTR_TYPE**)

name

The name of the attribute. The maximum length is 256 characters. (**a**)

Discussion

If you defined the attribute using %XML_ELEM_SETATTRIBUTE, you do not need to call %XML_ATTR_SETNAME to specify the attribute name. However, you can use %XML_ATTR_SETNAME to change the name of an existing XML attribute.

This function is defined as ^VAL.

%XML_ATTR_SETVALUE – Set the value of an XML attribute

status = %XML_ATTR_SETVALUE (*attr*, *value*)

Return value

status

XML_SUCCESS if the value is set successfully.

Arguments

attr

The XML attribute instance to set a value for. (**XML_ATTR_TYPE**)

value

The value of the attribute. The maximum length is 65,022 characters. (**a**)

Discussion

If you defined the attribute using %XML_ELEM_SETATTRIBUTE, you do not need to call %XML_ATTR_SETVALUE to specify the attribute's value. However, you can use %XML_ATTR_SETVALUE to change the value of an existing XML attribute

This function is defined as ^VAL.

See also

[%XML_OPTION](#) on page 18-95.

%XML_ATTR_SETVALUEHANDLE – Assign a handle as an XML attribute’s value

status = %XML_ATTR_SETVALUEHANDLE (*attr*, *handle*)

Return value

status

One of the following values:

XML_SUCCESS The value is set successfully.

XML_FAIL The value is not set successfully.

Arguments

attr

The XML attribute instance to set a value handle for. (**XML_ATTR_TYPE**)

handle

The handle of the value for the attribute. (**D_ADDR**)

Discussion

%XML_ATTR_SETVALUEHANDLE assigns a handle as the value for an XML attribute. The length of the attribute value cannot be larger than 65,502.

For performance reasons, %XML_ATTR_SETVALUEHANDLE does not honor %XML_OPTION with regard to escaping attribute values. If you use %XML_ATTR_SETVALUEHANDLE, escaping certain characters (for example, ampersands and double quotation marks) is your responsibility. You can escape characters using %SYN_ESCAPE_HANDLE before calling this method. See [%SYN_ESCAPE_HANDLE on page 9-324](#) for more information.



If the handle being set is different from the current one, you should release the old handle, if it isn't being used somewhere else, before setting the new one. If you don't, memory leaks will occur.

This function is defined as ^VAL.

Examples

The following example uses a handle to set the value of an attribute.

```

doc      ,XML_DOC_TYPE
attr     ,XML_ATTR_TYPE
root     ,XML_ELEM_TYPE
elem     ,XML_ELEM_TYPE
children ,XML_ELEMLIST_TYPE
attrlist ,XML_ATTRLIST_TYPE
h        ,d_addr

; Find the PeetsCoffee element and get the location attribute

root = %xml_doc_getroot(doc)
children = %xml_elem_children(root)
elem = %xml_elemlist_findtag(children, "PeetsCoffee")
attrlist = %xml_elem_attributes(elem)
attr = %xml_attrlist_find(attrlist, "location")

h = %mem_proc(DM_ALLOC+DM_STATIC+DM_BLANK, 200)

; Set a new value for the location attribute

^m(h) = 'Seattle, Washington, USA'

xcall xml_attr_setvaluehandle(attr, h)

```

%XML_ATTRLIST_ADD – Add an attribute to an XML attribute list

status = %XML_ATTRLIST_ADD (*attr_list*, *attr*)

Return value

status

XML_SUCCESS if the attribute is successfully added to the list.

Arguments

attr_list

The XML attribute list instance that you're adding an attribute to. (XML_ATTRLIST_TYPE)

attr

The XML attribute instance to add to the list. (XML_ATTR_TYPE)

Discussion

This function is defined as ^VAL.

%XML_ATTRLIST_ADDLIST – Add a list of attributes to an XML attribute list

status = %XML_ATTRLIST_ADDLIST(*destid*, *sourceid*)

Return value

status

One of the following values:

XML_SUCCESS Attributes are added successfully.

XML_FAIL Attributes are not added successfully.

Arguments

destid

The destination XML attribute list ID. (**XML_ATTRLIST_TYPE**)

sourceid

The source XML attribute list ID. (**XML_ATTRLIST_TYPE**)

Discussion

%XML_ATTRLIST_ADDLIST adds each of the attributes in the source XML attribute list to the end of the destination attribute list.

This function is defined as ^VAL.

Examples

The following example adds a new attribute list to another attribute list.

```

doc          ,XML_DOC_TYPE
root         ,XML_ELEM_TYPE
elem         ,XML_ELEM_TYPE
attr1        ,XML_ATTR_TYPE
attr2        ,XML_ATTR_TYPE
attrnew      ,XML_ATTRLIST_TYPE
attrorig     ,XML_ATTRLIST_TYPE
children     ,XML_ELEMLIST_TYPE

; Create a list of the attributes of element FolgersCoffee

root=%xml_doc_getroot(doc)
children = %xml_elem_children(root)
elem = %xml_elemlist_findtag(children, "FolgersCoffee")
attrorig = %xml_elem_attributes(elem)
```

```
; Create a new list of attributes

    attrnew = %xml_attrlist_create()

    attr1 = %xml_attr_create()
    xcall xml_attr_setname(attr1, "grind")
    xcall xml_attr_setvalue(attr1, "finegrind")

    attr2 = %xml_attr_create()
    xcall xml_attr_setname(attr2, "brew")
    xcall xml_attr_setvalue(attr2, "decaf")

    xcall xml_attrlist_add(attrnew, attr1)
    xcall xml_attrlist_add(attrnew, attr2)

; Append the new list to the original list

    xcall xml_attrlist_addlist(attrorig, attrnew)
```

%XML_ATTRLIST_CLEAR – Clear XML attribute list contents

status = %XML_ATTRLIST_CLEAR(*listid*)

Return value

status

XML_SUCCESS if the clear is successful.

Arguments

listid

The attribute list ID. (XML_ATTRLIST_TYPE)

Discussion

The %XML_ATTRLIST_CLEAR function clears the contents of the XML attribute list by releasing the attributes in the list, leaving an empty list.

This function is defined as ^VAL.

%XML_ATTRLIST_COPY – Copy attributes from one XML attribute list to another

status = %XML_ATTRLIST_COPY (*destid*, *sourceid*)

Return value

status

XML_SUCCESS if the copy is successful.

Arguments

destid

The destination attribute list ID. (XML_ATTRLIST_TYPE)

sourceid

The source attribute list ID. (XML_ATTRLIST_TYPE)

Discussion

The %XML_ATTRLIST_COPY function copies a list of attributes from a source XML attribute list to a destination XML attribute list. The contents of the destination list are cleared before the copy is done. The element object handle in *destid* must already exist before %XML_ATTRLIST_COPY is called; it is not created automatically.

This function is defined as ^VAL.

%XML_ATTRLIST_COUNT – Count the attributes in an XML attribute list

count = %XML_ATTRLIST_COUNT (*attr_list*)

Return value

count

The number of attributes in the list. (**i4**)

Arguments

attr_list

The XML attribute list instance whose attributes will be counted. (**XML_ATTRLIST_TYPE**)

Discussion

This function is defined as ^VAL.

%XML_ATTRLIST_CREATE – Create an XML attribute list

list = %XML_ATTRLIST_CREATE

Return value

list

The XML attribute list instance created, or 0 if not successful. (**XML_ATTRLIST_TYPE**)

Discussion

%XML_ATTRLIST_CREATE creates an empty XML attribute list. Attribute lists explicitly created with this routine and not added as data to a parent element will need to be deleted using %XML_ATTRLIST_DELETE.

This function is defined as ^VAL.

%XML_ATTRLIST_DELETE – Release an XML attribute list

```
status = %XML_ATTRLIST_DELETE(attr_list)
```

Return value

status

XML_SUCCESS if the attribute list is successfully released.

Arguments

attr_list

The XML attribute list instance to release. (XML_ATTRLIST_TYPE)

Discussion

You will rarely need to call this function, because the %XML_DOC_DELETE function releases all memory for an XML document, including all attribute lists associated with the document. The only time you will need to release an individual attribute list is when the list was explicitly created with %XML_ATTRLIST_CREATE (and therefore would not be released as part of a document or list).

This function is defined as ^VAL.

%XML_ATTRLIST_FIND – Find an attribute in an XML attribute list by name

attr = %XML_ATTRLIST_FIND(*attr_list*, *attr_name*)

Return value

attr

The XML attribute instance if found, or 0 if not found. (**XML_ATTR_TYPE**)

Arguments

attr_list

The XML attribute list instance. (**XML_ATTRLIST_TYPE**)

attr_name

The name of the attribute to find. (**a**)

Discussion

%XML_ATTRLIST_FIND looks for the first attribute with the specified name in an XML attribute list. You can alternatively retrieve an attribute by its index position using %XML_ATTRLIST_ITEM.

This function is defined as ^VAL.

%XML_ATTRLIST_INSERT – Insert an attribute at a specific position in a list

status = %XML_ATTRLIST_INSERT(*attr_list*, *attr*, *index*)

Return value

status

One of the following values:

XML_SUCCESS The attribute is inserted successfully.

XML_FAIL The attribute is not inserted successfully.

Arguments

attr_list

The destination XML attribute list instance in which to insert the attribute.
(**XML_ATTRLIST_TYPE**)

attr

The XML attribute instance to insert. (**XML_ATTR_TYPE**)

index

The index position at which to insert the attribute in the list (starting at 1). (**n**)

Discussion

%XML_ATTRLIST_INSERT inserts the specified attribute before the specified index position in an XML attribute list.

This function is defined as ^VAL.

Examples

The example below inserts a new attribute into an attribute list.

```

doc          ,XML_DOC_TYPE
root         ,XML_ELEM_TYPE
elem         ,XML_ELEM_TYPE
attr1        ,XML_ATTR_TYPE
attrlist     ,XML_ATTRLIST_TYPE
children     ,XML_ELEMLIST_TYPE

; Create a list of the attributes of element MaxwellHouse
root = %xml_doc_getroot(doc)
children = %xml_elem_children(root)
```

```
    elem = %xml_elemlist_findtag(children, "MaxwellHouse")
    attrlist = %xml_elem_attributes(elem)

; Create a new attribute

    attr1 = %xml_attr_create()
    xcall xml_attr_setname(attr1, "grind")
    xcall xml_attr_setvalue(attr1, "coursegrind")

; Insert in front of first attribute

    xcall xml_attrlist_insert(attrlist, attr1, 1)
```

%XML_ATTRLIST_ITEM – Find an attribute in an XML attribute list by position

attr = %XML_ATTRLIST_ITEM(*attr_list*, *index*)

Return value

attr

The XML attribute instance if found, or 0 if not found. (XML_ATTR_TYPE)

Arguments

attr_list

The XML attribute list instance. (XML_ATTRLIST_TYPE)

index

The index position of the attribute in the list. (n)

Discussion

%XML_ATTRLIST_ITEM looks for an attribute in an XML attribute list by its index position. You can alternatively retrieve an attribute by name using %XML_ATTRLIST_FIND.

This function is defined as ^VAL.

%XML_ATTRLIST_REMOVE – Remove an attribute from an XML attribute list

```
status = %XML_ATTRLIST_REMOVE(attr_list, attr)
```

Return value

status

XML_SUCCESS if the attribute was removed.

Arguments

attr_list

The XML attribute list instance. (XML_ATTRLIST_TYPE)

attr

The XML attribute instance to remove. (XML_ATTR_TYPE)

Discussion

%XML_ATTRLIST_REMOVE removes an attribute from the XML attribute list but does not delete the attribute. If you remove an attribute from a list and don't add it to another list, you must call %XML_ATTR_DELETE to delete the attribute and release its associated memory.

Examples

The example below assumes an XML attribute list called **alist** has an attribute whose name is **attr1**:

```
attr = %xml_attrlist_find(alist, "attr1")
if (attr)
  begin
    ret = %xml_attrlist_remove(alist, attr)
  end
```


%XML_DOC_ADDDECLARATION – Add a declaration to an XML document

status = %XML_DOC_ADDDECLARATION(*document*, *declaration_name*, *declaration*)

Return value

status

One of the following values:

XML_SUCCESS A declaration is added successfully.

XML_FAIL A declaration is not added successfully.

Arguments

document

The XML document instance to add a declaration to. (**XML_DOC_TYPE**)

declaration_name

The name of the declaration. Valid characters are alphanumeric characters, underscores, and dashes. (**a**)

declaration

The declaration text to add to the specified document. (See the Discussion below for the declaration format.) (**a**)

Discussion

%XML_DOC_ADDDECLARATION adds the specified declaration to an XML document (if it doesn't already exist in the document) by appending it to the list of declarations for that document. If a declaration with the same name is found, the existing declaration is updated.

A declaration is a processing instruction that begins with "<?" and ends with ">". Editors, browsers, and document processors use declarations to determine how a document should be processed.

Every XML document should begin with a declaration of the following form, which identifies the document as XML and specifies the version of XML being used:

```
<?xml version='version' [standalone='stand_status'] [encoding='encoding_name']?>
```

version

The current XML version number.

stand_status

One of the following:

- yes** The XML document has an internal DTD.
- no** The XML document has an external DTD. (default)

encoding_name

The encoding name of the character set being used. Common examples are UTF-8, UTF-16, ISO-10646-UCS-2, ISO-10646-UCS-4, ISO-8859-1 to ISO-8859-9, ISO-2022-JP, Shift_JIS, and EUC-JP. The default is UTF-8, or US-ASCII. The Synergy XML API only processes ASCII characters.

A *stand_status* value of **yes** tells a validating parser (one that validates a document against a DTD) to generate an error if an external DTD is encountered. A Synergy XML parser is *not* a validating parser.

For example:

```
status = %xml_doc_adddeclaration(doc, "xml", "version='1.0'
&      standalone='yes' encoding='UTF-8'")
```

adds the following full XML declaration to the document:

```
<?xml version='1.0' standalone='yes' encoding='UTF-8'?>
```

In addition to the XML version declaration, you can add declarations with other names, such as

```
status = %xml_doc_adddeclaration(mydoc, "word", "document='test.doc'")
```

which adds this processing instruction to the document:

```
<?word document="test.doc"?>
```

The maximum length for a document declaration string, including the declaration name and the opening and closing “<?” and “?>” is 250 characters. The default declaration, if none is explicitly set, is

```
<?xml version='1.0'?>
```

This function is defined as ^VAL. It supersedes %XML_DOC_SETDECLARATION.

Examples

The following call

```
status = %xml_doc_adddeclaration(doc, "xml-stylesheet", "type='text/xsl'
&      href='inprintdelivery.xsl'")
```

adds this full processing instruction to the document:

```
<?xml-stylesheet type='text/xsl' href='inprintdelivery.xsl'?>
```

%XML_DOC_CREATE – Create an XML document instance

document = %XML_DOC_CREATE

Return value

document

The XML document instance created, or 0 if not successful. (XML_DOC_TYPE)

Discussion

Calling %XML_DOC_CREATE is the first step in creating a well-formed XML document using Synergy data. %XML_DOC_CREATE not only creates a new XML document, but it also creates a root element in that document.

Documents explicitly created with this routine will need to be deleted using %XML_DOC_DELETE.

This function is defined as ^VAL.

Examples

See [“Example of assembling XML” on page 18-16.](#)

%XML_DOC_DELETE – Release an XML document instance

status = %XML_DOC_DELETE (*document*)

Return value

status

XML_SUCCESS if the document is deleted successfully.

Arguments

document

The XML document instance to delete. (XML_DOC_TYPE)

Discussion

After you're finished processing an XML document, you must release the document instance with %XML_DOC_DELETE. This routine releases all memory for an XML document, including the document itself, its root element, all of its children, and any attributes associated with these elements.

This function is defined as ^VAL.

Examples

```
xcall xml_doc_delete(document)
```

For a complete example, see [“Example of assembling XML”](#) on page 18-16.

%XML_DOC_GETDECLARATION – Get the declaration for an XML document

status = %XML_DOC_GETDECLARATION(*document*, *declaration*[, *declaration_name*])

Return value

status

One of the following values:

XML_SUCCESS A nonblank declaration string is returned.

XML_FAIL A declaration string is not returned successfully.

Arguments

document

The XML document instance to get a declaration for. (**XML_DOC_TYPE**)

declaration

Returned with the declaration string for the specified XML document instance. The maximum length is 250 characters. (**a**)

declaration_name

(optional) The name of the declaration to get. (**a**)

Discussion

A declaration begins with “<?” and ends with “?>”. If *declaration_name* is not specified, %XML_DOC_GETDECLARATION returns the first declaration for the XML document in *declaration*.

A sample declaration is as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

This function is defined as ^VAL.

See also

[%XML_DOC_ADDDECLARATION](#) on page 18-43.

%XML_DOC_GETROOT – Return the root element of an XML document

```
root = %XML_DOC_GETROOT(document)
```

Return value

root

The XML element that is the root of the XML document. (**XML_ELEM_TYPE**)

Arguments

document

The XML document instance for which to return a root element. (**XML_DOC_TYPE**)

Discussion

If you are processing XML data from a parsed document, one of the first things you should do is find the document's root element using %XML_DOC_GETROOT.

This function is defined as ^VAL.

Examples

```
root = %xml_doc_getroot(document)
```

For a more detailed example, see [“Example of parsing XML”](#) on page 18-14.

%XML_DOC_REMOVEDECLARATION – Remove a declaration from an XML document

status = %XML_DOC_REMOVEDECLARATION(*document*, *declaration_name*)

Return value

status

One of the following values:

XML_SUCCESS	The declaration is removed successfully.
XML_FAIL	The declaration is not removed successfully.

Arguments

document

The XML document instance from which to remove a declaration. (**XML_DOC_TYPE**)

declaration_name

The name of the declaration to remove. (**a**)

Discussion

This function is defined as ^VAL.



You cannot remove the **xml** declaration of the form `<?xml version='version'...?>`.

Examples

The following example removes the declaration named **xml-stylesheet** from the document (in other words, the one that starts with “`<?xml-stylesheet...?>`”).

```
status = %xml_doc_removedeclaration(docid, "xml-stylesheet")
```

%XML_DOC_SETCOMMENT – Set a comment at the XML document level

status = %XML_DOC_SETCOMMENT(*document*, *comment*)

Return value

status

XML_SUCCESS if a comment is set.

Arguments

document

The XML document instance to set a comment for. (**XML_DOC_TYPE**)

comment

The comment string to set. (**a**)

Discussion

Comments are hidden from the end user when the output document is displayed. A comment in an XML document has the format

```
<!--comment-->
```

%XML_DOC_SETCOMMENT adds the opening and closing characters (“<!--” and “-->”, respectively) for you; the *comment* argument should only include the text of the comment.

Note the following restrictions:

- ▶ Double hyphens (--) are not allowed as part of the comment text.
- ▶ Nested comments are not allowed.

This function is defined as ^VAL.

Examples

The function

```
status = %xml_doc_setcomment(document, "this is my comment")
```

sets a comment that looks like this:

```
<!--this is my comment-->
```


%XML_DOC_SETDECLARATION – Set the declaration for an XML document

%XML_DOC_SETDECLARATION is superseded by %XML_DOC_ADDDECLARATION.
See [%XML_DOC_ADDDECLARATION on page 18-43](#) for more information.

%XML_DOC_SETROOT – Set the root element of an XML document

status = %XML_DOC_SETROOT(*document*, *element*)

Return value

status

XML_SUCCESS if the root element is set successfully.

Arguments

document

The XML document instance for which to set the root element. (XML_DOC_TYPE)

element

The XML element instance to set as the root. (XML_ELEM_TYPE)

Discussion

An XML document should contain only one root element, and that element may appear only once in the document. The remainder of the page contents are contained inside this element.

Note that %XML_DOC_CREATE automatically creates a root element when it creates a new XML document.

This function is defined as ^VAL.

Examples

```
xcall xml_doc_setRoot(document, root)
```

For a more detailed example, see [“Example of assembling XML” on page 18-16](#).

%XML_DOC_TOFILE – Write contents of an XML document to a file

status = %XML_DOC_TOFILE(*document*, *filename*)

Return value

status

XML_SUCCESS if the file was written successfully.

Arguments

document

The XML document instance whose contents will be written to the specified file.
(XML_DOC_TYPE)

filename

The name of the file that will be written. (a)

Discussion

%XML_DOC_TOFILE serializes an XML document into a stream file. You can alternatively serialize an XML document into an XML string using %XML_DOC_TOSTRING.



When serializing an XML document to a file, %XML_DOC_TOFILE will not unescape any escaped text or attribute values in the XML document even if %XML_OPTION is set to SYNESCAPE_UNESCAPE.

This function is defined as ^VAL.

Examples

```
xcall xml_doc_tofile(document, "myfile.xml")
```

For a more detailed example, see “[Example of assembling XML](#)” on page 18-16.

%XML_DOC_TOSTRING – Write contents of an XML document to an XML string

xml_string = %XML_DOC_TOSTRING (*document*[, *format*])

Return value

xml_string

The XML string that contains the serialized XML document. (**XML_STRING_TYPE**)

Arguments

document

The XML document instance to serialize. (**XML_DOC_TYPE**)

format

(optional) If this argument is passed, the XML string is formatted with a line feed (LF) between the lines. (**n**)

Discussion

%XML_DOC_TOSTRING creates and returns an XML string that contains an entire XML document. You can then retrieve the Synergy memory handle for that string using %XML_STRING_GETHANDLE.

You can alternatively serialize an XML document into an XML file using %XML_DOC_TOFILE.

Strings created with %XML_DOC_TOSTRING should be deleted using %XML_STRING_DELETE when you're done with them.



When serializing an XML document to a string, %XML_DOC_TOSTRING will not unescape any escaped text or attribute values in the XML document even if %XML_OPTION is set to SYNEscape_UNESCAPE.

This function is defined as ^VAL.

Examples

See the CodeExchange at <http://www.synergex.com>.

%XML_ELEM_ADDATTRIBUTES – Add attributes in a list to an XML element

status = %XML_ELEM_ADDATTRIBUTES (*element*, *attr_list*)

Return value

status

One of the following values:

XML_SUCCESS The attributes are added successfully.

XML_FAIL The attributes are not added successfully.

Arguments

element

The XML element instance to which attributes will be added. (**XML_ELEM_TYPE**)

attr_list

The attribute list instance that contains the attributes to add. (**XML_ATTRLIST_TYPE**)

Discussion

%XML_ELEM_ADDATTRIBUTES adds each attribute in the specified list of attributes to the specified XML element. These attributes shall be appended to any existing element attributes.

This function is defined as ^VAL.

Examples

The example below adds two attributes to the first element of the root element.

```
doc          ,XML_DOC_TYPE
root         ,XML_ELEM_TYPE
elem         ,XML_ELEM_TYPE
elemlist     ,XML_ELEMLIST_TYPE
attr1        ,XML_ATTR_TYPE
attr2        ,XML_ATTR_TYPE
attrlist     ,XML_ATTRLIST_TYPE
```

```
; Get the list of all elements of root, then locate first element
```

```
root = %xml_doc_getroot(doc)
elemlist = %xml_elem_children(root)

elem = %xml_elemlist_item(elemlist, 1)
```

; Create new attributes, put them into a list, and add it to the element

```
attr1 = %xml_attr_create()
xcall xml_attr_setname(attr1, "Height")
xcall xml_attr_setvalue(attr1, "43inches")

attr2 = %xml_attr_create()
xcall xml_attr_setname(attr2, "Weight")
xcall xml_attr_setvalue(attr2, "18lbs")

attrlist = %xml_attrlist_create()
xcall xml_attrlist_add(attrlist, attr1)
xcall xml_attrlist_add(attrlist, attr2)

xcall xml_elem_addattributes(elem, attrlist)
```

%XML_ELEM_ADDCHILD – Add a child to an XML element

status = %XML_ELEM_ADDCHILD (*parent*, *child*)

Return value

status

XML_SUCCESS if the XML element child is added successfully.

Arguments

parent

The XML element instance to which a child will be added. (XML_ELEM_TYPE)

child

The child XML element instance to add to the specified parent. (XML_ELEM_TYPE)

Discussion

After you create a child XML element and specify its name and attributes, you must add it to the parent element by calling %XML_ELEM_ADDCHILD.

This function is defined as ^VAL.

Examples

The example below adds a child element called **field1** to the **root** element.

```
xcall xml_elem_addChild(root, field1)
```

Also see [“Example of assembling XML”](#) on page 18-16.

%XML_ELEM_ADDCHILDREN – Add child elements to an XML element

status = %XML_ELEM_ADDCHILDREN (*parent*, *elem_list*)

Return value

status

One of the following values:

XML_SUCCESS The child elements are added successfully.

XML_FAIL The child elements are not added successfully.

Arguments

parent

The XML element instance to which one or more children will be added.

(**XML_ELEM_TYPE**)

elem_list

A list of child elements that will be added to the parent. (**XML_ELEMLIST_TYPE**)

Discussion

%XML_ELEM_ADDCHILDREN adds each of the XML elements in the specified list of child elements to the specified parent element. These children are appended to any existing children already in the parent.

This function is defined as ^VAL.

Examples

The following example adds two new children to the root element.

```
doc          ,XML_DOC_TYPE
root         ,XML_ELEM_TYPE
elem1        ,XML_ELEM_TYPE
elem2        ,XML_ELEM_TYPE
elemlist     ,XML_ELEMLIST_TYPE

root = %xml_doc_getroot(doc)

elemlist = %xml_elemelist_create()

elem1=%xml_elem_create()
elem2=%xml_elem_create()
```



```
xcall xml_elem_setname(elem1, "JavaCity")
xcall xml_elem_setattribute(elem1, "location", "California")
xcall xml_elemlist_add(elemlist, elem1)

xcall xml_elem_setname(elem2, "SeattlesBest")
xcall xml_elem_setattribute(elem2, "location", "Washington")
xcall xml_elemlist_add(elemlist, elem2)

xcall xml_elem_addchildren(root, elemlist)
```

%XML_ELEM_ATTRIBUTES – Return attributes for an XML element

attr_list = %XML_ELEM_ATTRIBUTES (*element*)

Return value

attr_list

The attribute list instance for the specified element. (**XML_ATTRLIST_TYPE**)

Arguments

element

The XML element instance for which attributes will be returned. (**XML_ELEM_TYPE**)

Discussion

To get all of the attributes for an XML element, call %XML_ELEM_ATTRIBUTES, which returns an attribute list instance.

This function is defined as ^VAL.

%XML_ELEM_CHILDREN – Get children of an XML element

elem_list = %XML_ELEM_CHILDREN(*element*)

Return value

elem_list

A list of all child elements of the specified XML element. (XML_ELEMLIST_TYPE)

Arguments

element

The XML element instance for which to return a list of child elements.
(XML_ELEM_TYPE)

Discussion

To obtain a list of child elements for an XML element, call %XML_ELEM_CHILDREN.

This function is defined as ^VAL.

Examples

```
children = %xml_elem_children(root)
```

For a complete example, see [“Example of parsing XML” on page 18-14](#).

%XML_ELEM_COPY – Copy from one XML element to another

status = %XML_ELEM_COPY (*destid*, *sourceid*)

Return value

status

XML_SUCCESS if the copy is successful.

Arguments

destid

The destination element ID. (XML_ELEM_TYPE)

sourceid

The source element ID. (XML_ELEM_TYPE)

Discussion

The %XML_ELEM_COPY function copies the name, text, attributes, and children from one XML element to another. The element object handle in *destid* must already exist before %XML_ELEM_COPY is called; it is not created automatically.

This function is defined as ^VAL.

%XML_ELEM_CREATE – Create an XML element instance

element = %XML_ELEM_CREATE

Return value

element

The XML element instance created, or 0 if not successful. (XML_ELEM_TYPE)

Discussion

Use %XML_ELEM_CREATE to create both root elements and subelements. XML elements explicitly created with this routine and not added as data to a parent element will need to be deleted using %XML_ELEM_DELETE.

This function is defined as ^VAL.

Examples

The following example creates a root element.

```
root = %xml_elem_create
```

Also see [“Example of assembling XML” on page 18-16](#).

%XML_ELEM_DELETE – Release an XML element instance

status = %XML_ELEM_DELETE (*element*)

Return value

status

XML_SUCCESS if the element was released successfully.

Arguments

element

The XML element instance to release. (**XML_ELEM_TYPE**)

Discussion

%XML_ELEM_DELETE releases an XML element and all of its subelements and attributes. You will rarely need to call this function, because the %XML_DOC_DELETE function releases all memory for an XML document, including all elements and attributes associated with the document. The only time you will need to release an individual element is when the element was explicitly created with %XML_ELEM_CREATE and was not added as data to a parent element (and therefore would not be released as part of a document or list).

Note that while %XML_ELEM_DELETE releases the element's memory, it does not automatically remove the element from any element lists of which it is a member.

This function is defined as ^VAL.

%XML_ELEM_GETATTRIBUTE – Get the value of an attribute for an XML element

status = %XML_ELEM_GETATTRIBUTE(*element*, *attr_name*, *attr_value*)

Return value

status

XML_SUCCESS if a nonblank attribute value is returned.

Arguments

element

The XML element instance. (XML_ELEM_TYPE)

attr_name

The name of the attribute to find a value for. (a)

attr_value

Returned with the value of the specified attribute. (a)

Discussion

%XML_ELEM_GETATTRIBUTE calls %XML_ATTRLIST_FIND to find the XML attribute by name and then calls %XML_ATTR_GETVALUE to get the attribute's value.

This function is defined as ^VAL.

Examples

```
xcall xml_elem_getAttribute(root, "name", attrValue)
```

For a complete example, see [“Example of parsing XML” on page 18-14](#).

%XML_ELEM_GETELEMENTSBYNAME – Get children of an XML element by tag

elem_list = %XML_ELEM_GETELEMENTSBYNAME (*element*, *tag_name*)

Return value

elem_list

The XML element list of all children with the specified tag name directly under the specified XML element, or 0 if no children are found. (**XML_ELEMLIST_TYPE**)

Arguments

element

The XML element instance whose children will be listed. (**XML_ELEM_TYPE**)

tag_name

The tag name of the children to list. (**a**)

Discussion

%XML_ELEM_GETELEMENTSBYNAME uses a tag name to retrieve a list of child XML elements for the specified parent element. Element lists explicitly created with this routine will need to be deleted using %XML_ELEMLIST_DELETE.

This function is defined as ^VAL.

Examples

We'll use the following portion of an XML document as an example:

```
<structure name="mystruct" >
  <group name="mygroup" />
  <field name="f1" />
  <field name="f2" />
</structure>
```

Assuming **parent** is the **structure** element,

```
%xml_elem_getElementsByName(parent, "field")
```

will return a list that contains two children, **f1** and **f2**.

Also see [“Example of parsing XML” on page 18-14](#).

%XML_ELEM_GETNAME – Get the tag name of an XML element

status = %XML_ELEM_GETNAME(*element*, *tag_name*)

Return value

status

XML_SUCCESS if a nonblank name is returned.

Arguments

element

The XML element instance to get a tag name for. (XML_ELEM_TYPE)

tag_name

Returned with the tag name of the XML element as a string value. The maximum length is 256 characters. (**a**)

Discussion

Every XML element is represented by a start tag and an end tag delimiting its text and/or children.

For example, if the start and end tags are <time> and </time>, the tag name is **time**.

%XML_ELEM_GETNAME retrieves the tag name for the specified element instance.

This function is defined as ^VAL.

Examples

```
xcall xml_elem_getName(root, tagName)
```

For a complete example, see [“Example of parsing XML” on page 18-14](#).

%XML_ELEM_GETTEXT – Get the text value of an XML element

status = %XML_ELEM_GETTEXT(*element*, *text*[, *length*])

Return value

status

XML_SUCCESS if a nonblank text value is returned.

Arguments

element

The XML element instance to get a value for. (**XML_ELEM_TYPE**)

text

Returned with the text string value of the element. The maximum length is 65,534 characters. (**a**)

length

(optional) Returned with the length of the returned text. (**n**)

Discussion

%XML_ELEM_GETTEXT returns the text that appears between the specified element's opening tag and closing tag, including any embedded XML documents delimited by CDATA. Leading and trailing blanks are stripped.

This function is defined as ^VAL.



If you want to receive variable-length data in a large buffer, we recommend using the *length* argument, as this will avoid time spent clearing the remainder of the buffer (beyond *length*) with spaces.

See also

[%XML_OPTION on page 18-95](#) for information about unescaping text when calling %XML_ELEM_GETTEXT.

%XML_ELEM_GETTEXTHANDLE – Return handle containing text for an element

handle = %XML_ELEM_GETTEXTHANDLE (*element*)

Return value

handle

The handle that contains the text for the specified element, or 0 if no text is assigned.
(D_ADDR)

Arguments

element

The XML element instance to get the text handle for. (XML_ELEM_TYPE)

Discussion

%XML_ELEM_GETTEXTHANDLE returns the memory handle that contains the text for an XML attribute.

Since %XML_ELEM_GETTEXTHANDLE may be used instead of %XML_ELEM_GETTEXT for performance reasons, %XML_ELEM_GETTEXTHANDLE does not honor %XML_OPTION with regard to unescaping text. If you use %XML_ELEM_GETTEXTHANDLE, unescaping certain characters (for example, ampersands and double quotation marks) is your responsibility. You can unescape characters using %SYN_UNESCAPE_HANDLE after calling this method. See [%SYN_UNESCAPE_HANDLE on page 9-336](#) for more information.



If you call %XML_ELEM_GETTEXTHANDLE and then do an unescape on the handle, any document that contains that element will be modified. Therefore, if you don't want to modify your document, you should make a copy of the handle before performing any escaping or unescaping.

This function is defined as ^VAL.

Examples

The example below finds the element that contains the text.

```
doc      , XML_DOC_TYPE
root     , XML_ELEM_TYPE
elem     , XML_ELEM_TYPE
children , XML_ELEMLIST_TYPE
h        , d_addr
```

```
root = %xml_doc_getroot(doc)
children = %xml_elem_children(root)
elem = %xml_elemlist_findtag(children, "PeetsCoffee")

h = %xml_elem_gettexthandle(elem)

open(1, o, "outfile.txt")

if (h) then
    writes(1, ^m(h))
else
    writes(1, "Unable to get the texthandle")
close 1
```

%XML_ELEM_REMOVEATTRIBUTE – Remove an attribute from an XML element

status = %XML_ELEM_REMOVEATTRIBUTE(*element*, *attr_name*[, *attribute*])

Return value

status

XML_SUCCESS if the specified attribute is removed. (**i4**)

Arguments

element

The XML element instance whose attribute will be removed. (**XML_ELEM_TYPE**)

attr_name

The name of the attribute to remove. (**a**)

attribute

(optional) The removed attribute. (**XML_ATTR_TYPE**)

Discussion

This method will remove the first attribute with the specified name in the XML element's list of attributes. If the *attribute* argument is specified, you are responsible for deleting memory by calling %XML_ATTR_DELETE for that *attribute*. If *attribute* is not specified, %XML_ELEM_REMOVEATTRIBUTE will delete the attribute automatically.

If you need more control in deleting attributes, see [%XML_ATTRLIST_REMOVE](#) on page 18-42.

Examples

The example below assumes an XML element called **elem** has an attribute whose name is **attr1**:

```
ret=%xml_elem_removeattribute(elem, " attr1")
```

Or, if we want to keep the removed attribute in a variable called **attr**, we'd do this:

```
attr, XML_ATTR_TYPE
ret=%xml_elem_removeattribute(elem, "attr1", attr)
;If it doesn't get added to any attribute lists, delete it when finished:
xcall xml_attr_delete(attr)
```

%XML_ELEM_REMOVEATTRIBUTES – Remove all attributes from an XML element

status = %XML_ELEM_REMOVEATTRIBUTES (*element*)

Return value

status

One of the following values:

XML_SUCCESS The attributes are removed successfully.

XML_FAIL The attributes are not removed successfully.

Arguments

element

The XML element instance whose attributes will be removed. (**XML_ELEM_TYPE**)

Discussion

This function is defined as ^VAL.

Examples

The following example removes all of the attributes for the **CompanyABC** child of the root element.

```
doc          ,XML_DOC_TYPE
root         ,XML_ELEM_TYPE
elem         ,XML_ELEM_TYPE
children     ,XML_ELEMLIST_TYPE
findstr      ,a30
```

```
root = %xml_doc_getroot(doc)
children = %xml_elem_children(root)
```

```
findstr = "CompanyABC"
elem = %xml_elemlist_findtag(children, %atrim(findstr))
```

```
xcall xml_elem_removeattributes(elem)
```

%XML_ELEM_REMOVECHILD – Remove the first child specified from an element

status = %XML_ELEM_REMOVECHILD(*element*, *child_name*[, *child_id*])

Arguments

status

XML_SUCCESS if the specified child was removed. (**i4**)

element

The XML element instance of the parent element. (**XML_ELEM_TYPE**)

child_name

The tag name of the child element to remove. (**a**)

child_id

(optional) The removed child element. (**XML_ELEM_TYPE**)

Discussion

%XML_ELEM_REMOVECHILD removes the first child with the given tag name from an XML element. If the *child_id* argument is specified, you are responsible for deleting memory by calling %XML_ELEM_DELETE for that *child_id*. If *child_id* is not specified, %XML_ELEM_REMOVECHILD will delete the child automatically.

If you need more control than simply removing the first child, see [%XML_ELEMLIST_REMOVE on page 18-94](#). To remove all child elements, see [%XML_ELEM_REMOVECHILDREN on page 18-74](#).

Examples

The example below assumes an XML element called **elem** has a child whose tag name is **child1**:

```
ret=%xml_elem_removechild(elem, "child1")
```

Or, if we want to keep the removed child element in a variable called **child**, we'd do this:

```
ret = %xml_elem_removechild(elem,"child1",child)
;If it doesn't get added to any element lists, delete it when finished:
xcall xml_elem_delete(child)
```

%XML_ELEM_REMOVECHILDREN – Remove all child elements from an element

status = %XML_ELEM_REMOVECHILDREN(*element*)

Return value

status

One of the following values:

XML_SUCCESS The child elements are removed successfully.

XML_FAIL The child elements are not removed successfully.

Arguments

element

The XML element instance whose child elements will be removed. (**XML_ELEM_TYPE**)

Discussion

%XML_ELEM_REMOVECHILDREN removes all child elements from an XML element. This function is defined as ^VAL.

Examples

The example below removes all child elements from the root element.

```
doc          , XML_DOC_TYPE
root         , XML_ELEM_TYPE

root = %xml_doc_getroot(doc)

xcall xml_elem_removechildren(root)
```


%XML_ELEM_SETATTRIBUTE – Add an attribute to an XML element

status = %XML_ELEM_SETATTRIBUTE(*element*, *attr_name*, *attr_value*)

Return value

status

XML_SUCCESS if the attribute was set successfully.

Arguments

element

The XML element instance for which an attribute is being set. (**XML_ELEM_TYPE**)

attr_name

The name of the attribute to look for or set. The maximum length is 256 characters. (**a**)

attr_value

The value of the attribute. The maximum length is 65,022 characters. (**a**)

Discussion

%XML_ELEM_SETATTRIBUTE looks for an XML attribute by name. If it finds it, it updates the attribute's value. If it cannot find the attribute, it adds a new attribute to the element's attribute list using the specified name and value.

In the XML document, attributes are written in an element's start tag after the tag name, beginning with the name of the attribute, followed by an equal sign (=) and a value for that attribute. For example, in

```
<structure name="mystruct" >
```

name = "mystruct" is the attribute and its value.

This function is defined as ^VAL.

See also

[%XML_OPTION on page 18-95.](#)

Examples

```
xcall xml_elem_setAttribute(f1, "name", "id")
xcall xml_elem_setAttribute(f1, "type", "decimal")
xcall xml_elem_setAttribute(f1, "size", "4")
```

For a complete example, see [“Example of assembling XML” on page 18-16.](#)

%XML_ELEM_SETNAME – Set the tag name of an XML element

status = %XML_ELEM_SETNAME (*element*, *tag_name*)

Return value

status

XML_SUCCESS if the tag name is set successfully.

Arguments

element

The XML element instance for which a tag name will be set. (**XML_ELEM_TYPE**)

tag_name

The tag name string value to set for the specified element. The maximum length is 256 characters. (**a**)

Discussion

%XML_ELEM_SETNAME defines the tag name for the specified XML element. The tag name is the text that appears in the begin and end tags required to delimit each element. If you are assembling Synergy data in XML format, one of the first things you should do after you create the XML document is to set the name of the root element with %XML_ELEM_SETNAME.

This function is defined as ^VAL.

Examples

The example below sets the tag name for the **root** element to **weather-report**. The begin and end tags would be <weather-report> and </weather-report>, respectively.

```
xcall xml_elem_setName(root, "weather-report")
```

Also see [“Example of assembling XML” on page 18-16](#).

%XML_ELEM_SETTEXT – Set the text value of an XML element

status = %XML_ELEM_SETTEXT(*element*, *text*)

Return value

status

XML_SUCCESS if the text was set successfully.

Arguments

element

The XML element instance to set text for. (XML_ELEM_TYPE)

text

The text that will appear between the element's opening tag and closing tag. The maximum length is 65,534 characters. (a)

Discussion

%XML_ELEM_SETTEXT sets the text that appears between the XML element's opening and closing tags.

This function is defined as ^VAL.

See also

[%XML_OPTION on page 18-95](#) for information about escaping text when calling %XML_ELEM_SETTEXT.

Examples

See the CodeExchange at <http://www.synergex.com>.

%XML_ELEM_SETTEXTHANDLE – Assign a handle as the text for an XML element

status = %XML_ELEM_SETTEXTHANDLE (*element*, *handle*)

Return value

status

One of the following values:

XML_SUCCESS The handle is assigned successfully.

XML_FAIL The handle is not assigned successfully.

Arguments

element

The XML element instance to set a text handle for. (**XML_ELEM_TYPE**)

handle

The handle of the text for the XML element. (**D_ADDR**)

Discussion

The maximum length of the element text is 65,534 characters.

For performance reasons, %XML_ELEM_SETTEXTHANDLE does not honor %XML_OPTION with regard to escaping text. If you use %XML_ELEM_SETTEXTHANDLE, escaping certain characters (for example, ampersands and double quotation marks) is your responsibility. You can escape characters using %SYN_ESCAPE_HANDLE before calling this method. See [%SYN_ESCAPE_HANDLE on page 9-324](#) for more information.



If the handle being set is different from the current one, you should release the old handle, if it isn't being used somewhere else, before setting the new one. If you don't, memory leaks will occur.

This function is defined as ^VAL.

Examples

The example below uses a handle to set the text for an element.

```
doc      ,XML_DOC_TYPE
root     ,XML_ELEM_TYPE
elem     ,XML_ELEM_TYPE
children ,XML_ELEMLIST_TYPE
h        ,d_addr
```

```
; Find the element FolgersCoffee to receive the text

root = %xml_doc_getroot(doc)
children = %xml_elem_children(root)
elem = %xml_elemlist_findtag(children, "FolgersCoffee")

h = %mem_proc(DM_ALLOC+DM_STATIC+DM_BLANK, 200)
^m(h) = 'here is the string we want to add'

xcall xml_elem_settexthandle(elem, h)
```

%XML_ELEM_TOSTRING – Give the string representation for an XML element

status = %XML_ELEM_TOSTRING(*element*, *string*)

Arguments

status

XML_SUCCESS if the element was serialized. (**i4**)

element

The XML element instance for which you want a string representation.
(**XML_ELEM_TYPE**)

string

The string to append to. (**XML_STRING_TYPE**)

Discussion

String must be created with %XML_STRING_CREATE before you call %XML_ELEM_TOSTRING. %XML_ELEM_TOSTRING appends the element serialization to the end of the string.

Examples

The example below assumes you have an XML element called **elem1** that you want to append to an XML string:

```
strid, XML_STRING_TYPE

strid = %xml_string_create
ret = %xml_elem_tostring(elem1, strid)
```

%XML_ELEMLIST_ADD – Add an element to an XML element list

status = %XML_ELEMLIST_ADD(*elem_list*, *element*)

Return value

status

XML_SUCCESS if the element was added successfully.

Arguments

elem_list

The XML element list instance to which the element will be added.
 (XML_ELEMLIST_TYPE)

element

The XML element instance to add. (XML_ELEM_TYPE)

Discussion

%XML_ELEMLIST_ADD appends an element to the end of a list of XML elements.

This function is defined as ^VAL.

%XML_ELEMLIST_ADDLIST – Add a list of elements to an XML element list

status = %XML_ELEMLIST_ADDLIST(*destid*, *sourceid*)

Return value

status

One of the following values:

XML_SUCCESS The elements are added successfully.

XML_FAIL The elements are not added successfully.

Arguments

destid

The destination XML element list ID. (**XML_ELEMLIST_TYPE**)

sourceid

The source XML element list ID. (**XML_ELEMLIST_TYPE**)

Discussion

%XML_ELEMLIST_ADDLIST adds each of the elements in the source XML element list to the end of the destination XML element list.

This function is defined as ^VAL.

Examples

The example below adds a new element list to the root element's element list.

```
doc                ,XML_DOC_TYPE
root               ,XML_ELEM_TYPE
elem1              ,XML_ELEM_TYPE
elem2              ,XML_ELEM_TYPE
elemlistroot       ,XML_ELEMLIST_TYPE
elemlistnew        ,XML_ELEMLIST_TYPE

; Get a list of the children of root

root = %xml_doc_getroot(doc)
elemlistroot=%xml_elem_children(root)

; Create a new list of children

elemlistnew=%xml_elemlist_create()
```



```
elem1=%xml_elem_create()
elem2=%xml_elem_create()

xcall xml_elem_setname(elem1, "Folgers")
xcall xml_elem_setname(elem2, "MaxwellHouse")
xcall xml_elemlist_add(elemlistnew, elem1)
xcall xml_elemlist_add(elemlistnew, elem2)

; Add the new list to the existing list

xcall xml_elemlist_addlist(elemlistroot, elemlistnew)
```

%XML_ELEMLIST_CLEAR – Clear XML element list contents

status = %XML_ELEMLIST_CLEAR(*elem_list*)

Return value

status

XML_SUCCESS if the copy is successful.

Arguments

elem_list

The XML element list instance. (XML_ELEMLIST_TYPE)

Discussion

The %XML_ELEMLIST_CLEAR function clears the contents of the XML element list by releasing the elements in the list, leaving an empty list.

This function is defined as ^VAL.

%XML_ELEMLIST_COPY – Copy an XML element list to another list

status = %XML_ELEMLIST_COPY (*dest_list*, *source_list*)

Return value

status

XML_SUCCESS if the list was copied successfully.

Arguments

dest_list

The element list instance to copy to. (XML_ELEMLIST_TYPE)

source_list

The element list instance to copy from. (XML_ELEMLIST_TYPE)

Discussion

If the destination and source lists are not the same, %XML_ELEMLIST_COPY clears the destination list and copies every XML element from the source list to the destination list. The element object handle in *destid* must already exist before %XML_ELEMLIST_COPY is called; it is not created automatically.

This function is defined as ^VAL.

%XML_ELEMLIST_COUNT – Count the elements in an XML element list

count = %XML_ELEMLIST_COUNT (*elem_list*)

Return value

count

The number of elements in the list. (**i4**)

Arguments

elem_list

The XML element list instance. (**XML_ELEMLIST_TYPE**)

Discussion

%XML_ELEMLIST_COUNT returns the number of elements in the XML list. This function can be used in combination with %XML_ELEMLIST_ITEM to iterate through the element list using a Synergy FOR loop.

This function is defined as ^VAL.

Examples

```
for i from 1 thru %xml_elemlist_count(children)
```

For a complete example, see [“Example of parsing XML” on page 18-14.](#)

%XML_ELEMLIST_CREATE – Create an XML element list

elem_list = %XML_ELEMLIST_CREATE

Return value

elem_list

The XML element list instance created, or 0 if not successful. (XML_ELEMLIST_TYPE)

Discussion

XML element lists explicitly created with this routine and not added as data to a parent element will need to be deleted using %XML_ELEM_DELETE.

This function is defined as ^VAL.

%XML_ELEMLIST_DELETE – Release an XML element list

status = %XML_ELEMLIST_DELETE(*elem_list*)

Return value

status

XML_SUCCESS if the element list is successfully released.

Arguments

elem_list

The XML element list instance to release. (XML_ELEMLIST_TYPE)

Discussion

Use %XML_ELEMLIST_DELETE to release XML element lists explicitly created with %XML_ELEM_GETELEMENTSBYNAME or when the element list was explicitly created with %XML_ELEMLIST_CREATE and was not added as data to a parent element (and therefore would not be released as part of a document or list).

This function is defined as ^VAL.

%XML_ELEMLIST_FIND – Find an element in an XML element list by name

element = %XML_ELEMLIST_FIND (*elem_list*, *name*)

Return value

element

The XML element instance found, or 0 if not successful. (XML_ELEM_TYPE)

Arguments

elem_list

The XML element list instance. (XML_ELEMLIST_TYPE)

name

The value of the **name** attribute for the element you want to find. (a)

Discussion

%XML_ELEMLIST_FIND looks for the first element with the specified name in an XML element list. If you want to find subsequent elements with that name, you must manually iterate through the list.

You can alternatively retrieve an element by its tag using %XML_ELEMLIST_FINDTAG or by its index position using %XML_ELEMLIST_ITEM.

This function is defined as ^VAL.

%XML_ELEMLIST_FINDTAG – Find an element in an XML element list by tag

element = %XML_ELEMLIST_FINDTAG (*elem_list*, *tag*)

Return value

element

The XML element instance found, or 0 if not successful. (**XML_ELEM_TYPE**)

Arguments

elem_list

The XML element list instance. (**XML_ELEMLIST_TYPE**)

tag

The tag name of the element to find. (**a**)

Discussion

%XML_ELEMLIST_FINDTAG looks for the first element with the specified tag name in an XML element list. If you want to find subsequent elements with that tag name, you must manually iterate through the list.

You can alternatively retrieve an element by its name using %XML_ELEMLIST_FIND or by its index position using %XML_ELEMLIST_ITEM.

This function is defined as ^VAL.

%XML_ELEMLIST_INSERT – Insert an XML element at a specific position in a list

status = %XML_ELEMLIST_INSERT(*elem_list*, *element*, *index*)

Return value

status

One of the following values:

XML_SUCCESS The element is inserted successfully.

XML_FAIL The element is not inserted successfully.

Arguments

elem_list

The XML element list instance in which to insert the element. (**XML_ELEMLIST_TYPE**)

element

The XML element instance to insert. (**XML_ELEM_TYPE**)

index

The index position at which to insert the element in the list (starting at 1). (**n**)

Discussion

%XML_ELEMLIST_INSERT inserts the specified element before the specified index position in an XML element list.

This function is defined as ^VAL.

Examples

The following example inserts an element into the root element's element list.

```

doc          ,XML_DOC_TYPE
root         ,XML_ELEM_TYPE
elem1        ,XML_ELEM_TYPE
elemlist     ,XML_ELEMLIST_TYPE
count        ,i4

; Get a list of the children of root

root = %xml_doc_getroot(doc)
elemlist = %xml_elem_children(root)
count = %xml_elemlist_count(elemlist) ;Get the number of elements

```

```
; Insert the new element in front of the last element
```

```
elem1 = %xml_elem_create()
```

```
xcall xml_elem_setname(elem1, "FolgersCoffee")
```

```
xcall xml_elem_setAttribute(elem1, "grind", "finegrind")
```

```
xcall xml_elemlist_insert(elemlist, elem1, count)
```

%XML_ELEMLIST_ITEM – Find an element in an XML element list by position

```
element = %XML_ELEMLIST_ITEM(elem_list, index)
```

Return value

element

The XML element instance found, or 0 if not successful. (XML_ELEM_TYPE)

Arguments

elem_list

The XML element list instance. (XML_ELEMLIST_TYPE)

index

The index position of the element in the list. (n)

Discussion

%XML_ELEMLIST_ITEM looks for an XML element by its index position. You can alternatively retrieve an element by its name using %XML_ELEMLIST_FIND or by its tag using %XML_ELEMLIST_FINDTAG.

This function is defined as ^VAL.

Examples

```
elem = %xml_elemlist_item(children, i)
```

For a complete example, see [“Example of parsing XML” on page 18-14](#).

%XML_ELEMLIST_REMOVE – Remove an element from an XML element list

```
status = %XML_ELEMLIST_REMOVE(elem_list, element)
```

Arguments

status

XML_SUCCESS if the element was removed from the list. (**i4**)

elem_list

The XML element list instance. (**XML_ELEMLIST_TYPE**)

element

The XML element instance to remove from the list. (**XML_ELEM_TYPE**)

Discussion

%XML_ELEMLIST_REMOVE removes the XML element from the list but does not delete the element.

If you remove an element from a list and don't add it to another list, you must call %XML_ELEM_DELETE to delete that element and release its associated memory.

Examples

The example below assumes an XML element list called **elist** has an element whose name is **child1**.

```
child, XML_ELEM_TYPE
child = %xml_elemlist_findtag(elist, "child1")
if (child)
  begin
    ret = %xml_elemlist_remove(elist, child)
  end
```

%XML_OPTION – Set an XML option value

status = %XML_OPTION(*option*, *value*)

Return value

status

One of the following values:

XML_SUCCESS A valid value is provided and the option is set successfully.

XML_FAIL A valid value is not provided and the option is not set successfully.

Arguments

option

The XML option to set. (**a**)

value

The value for the option. (**n**)

Discussion

%XML_OPTION sets a value for the following options:

Option	Value	Behavior
ENCODE	SYNESCPE_OFF (default)	Does not escape or unescape any items.
	SYNESCPE_ESCAPE	Escapes text or attribute values according to XML escape rules (see page 9-324) when calling %XML_ELEM_SETTEXT, %XML_ELEM_SETATTRIBUTE, or %XML_ATTR_SETVALUE, and escapes the text or attribute values set by %XML_PARSER_PARSEFILE or %XML_PARSER_PARSESTRING.
	SYNESCPE_UNESCAPE	Unescapes text or attribute values when calling %XML_ELEM_GETTEXT or %XML_ATTR_GETVALUE, according to XML escape rules (see page 9-324).



When serializing an XML document to a string or a file, %XML_DOC_TOSTRING and %XML_DOC_TOFILE won't unescape any escaped text or attribute values in the XML document.

This function is defined as ^VAL.

Examples

The following example uses XML_OPTION to engage the unescape behavior for XML_ELEM_GETTEXT.

```
doc      ,XML_DOC_TYPE
root     ,XML_ELEM_TYPE
elem     ,XML_ELEM_TYPE
children ,XML_ELEMLIST_TYPE
text     ,a100

; Find the element that contains the text

root = %xml_doc_getroot(doc)
children = %xml_elem_children(root)
elem = %xml_elemlist_findtag(children, "MaxwellHouse")

; Set an option to have the text string unescaped

xcall xml_option("ENCODE", SYNEscape_UNESCAPE)
xcall xml_elem_gettext(elem, text)
```

%XML_PARSER_CREATE – Create an XML parser instance

parser = %XML_PARSER_CREATE

Return value

parser

The parser instance created, or 0 if not successful. (**XML_PARSER_TYPE**)

Discussion

Before you can parse an XML file or string, you must create an instance of the parser using %XML_PARSER_CREATE.

This function is defined as ^VAL.

Examples

See [“Example of parsing XML” on page 18-14](#).

%XML_PARSER_DELETE – Release an XML parser instance

status = %XML_PARSER_DELETE(*parser*)

Return value

status

XML_SUCCESS if the parser instance was successfully released.

Arguments

parser

The parser instance to release. (**XML_PARSER_TYPE**)

Discussion

After you're finished parsing an XML file or string, you must release the parser instance with %XML_PARSER_DELETE. You can release the parser any time after calling %XML_PARSER_PARSEFILE or %XML_PARSER_PARSESTRING.



Remember to also delete the document returned by %XML_PARSER_PARSEFILE or %XML_PARSER_PARSESTRING, using %XML_DOC_DELETE.

This function is defined as ^VAL.

Examples

```
xcall xml_parser_delete(parser)
```

For a complete example, see [“Example of parsing XML” on page 18-14](#).

%XML_PARSER_ERROR – Get the last XML parser error

status = %XML_PARSER_ERROR(*parser*, *error*)

Return value

status

XML_SUCCESS if a nonblank error message is returned.

Arguments

parser

The parser instance. (XML_PARSER_TYPE)

error

Returned with the text of the last parser error. (a)

Discussion

%XML_PARSER_ERROR obtains the last error encountered by the DOM parser when parsing an XML file or string. Possible XML errors are as follows:

Error message	Cause
<i>One of several Synergy file I/O error messages</i>	The %XML_PARSER_PARSEFILE routine had trouble opening the XML file. See the “Error Messages” chapter of <i>Synergy Language Tools</i> for details about the specific error you received.
Invalid XML. Missing closing tag <elementName>	The XML is missing the closing tag for the specified element name.
Invalid XML. Missing closing >	A tag is missing a closing “>” character.
Invalid XML. Unexpectedly encountered <	The parser encountered an opening tag character (<) while looking for a closing tag character (>).
Invalid XML. Expecting closing quote for attribute <attrib_name>	The specified attribute value is missing a closing quotation mark.
Invalid XML. Error reading text	The text value for an element is not located between an opening and a closing tag.

Error message	Cause
Invalid XML. Error reading comment	The parser cannot find the closing comment characters "-->".
Invalid XML. Error reading declaration	The parser cannot find a closing ">" character for a declaration.
Invalid XML. Error reading XML declaration	The parser cannot find closing "?>" characters for the XML declaration.

This function is defined as ^VAL.

Examples

See [“Example of parsing XML” on page 18-14.](#)

%XML_PARSER_PARSEFILE – Parse an XML file

document = %XML_PARSER_PARSEFILE (*parser*, *filename*)

Return value

document

The XML document instance created, or 0 if not successful. (XML_DOC_TYPE)

Arguments

parser

The parser instance. (XML_PARSER_TYPE)

filename

The name of the XML file to parse. (a)

Discussion

%XML_PARSER_PARSEFILE parses a well-formed XML file and returns a new XML document instance. The created document can be processed using the %XML_DOC_XXX routines. The document can be deleted using %XML_DOC_DELETE.



The maximum length of text for an element in an XML document is 65,534 characters. If any element text exceeds this limit, the parser will not return a valid value.

If the XML data to be parsed is in memory (as an alpha string or referenced by a memory handle), call %XML_PARSER_PARSESTRING instead. (The data must first be converted to an XML string using the %XML_STRING_XXX routines.)

This function is defined as ^VAL.

See also

[%XML_OPTION on page 18-95](#) for information about escaping text or attribute values when calling %XML_PARSER_PARSEFILE.

Examples

```
doc = %xml_parser_parsefile(dom, weather.xml)
```

Also see [“Example of parsing XML” on page 18-14](#).

%XML_PARSER_PARSESTRING – Parse an XML string

document = %XML_PARSER_PARSESTRING (*parser*, *xml_string*)

Return value

document

The XML document instance that was created if the parse was successful, or 0 if the parse was not successful. (**XML_DOC_TYPE**)

Arguments

parser

The parser instance returned by %XML_PARSER_CREATE. (**XML_PARSER_TYPE**)

xml_string

The XML string instance to parse, as returned by %XML_STRING_CREATE. (**XML_STRING_TYPE**)

Discussion

%XML_PARSER_PARSESTRING parses well-formed XML contained in an XML string and returns a new XML document instance. The created document can be processed using the %XML_DOC_xxx routines. The document can be deleted using %XML_DOC_DELETE.



The maximum length of text for an element in an XML document is 65,534 characters. If any element text exceeds this limit, the parser will not return a valid value.

If the XML data to be parsed is in a physical file instead of an XML string, call %XML_PARSER_PARSEFILE instead.

This function is defined as ^VAL.

See also

%XML_OPTION on page 18-95 for information about escaping text or attribute values when calling %XML_PARSER_PARSEFILE.

Examples

See the CodeExchange at <http://www.synergex.com>.

%XML_STRING_APPEND – Append a Synergy alpha string to an XML string

```
status = %XML_STRING_APPEND(xml_string, synergy_string[, length])
```

Return value

status

XML_SUCCESS if the string was appended successfully.

Arguments

xml_string

The XML string instance to which the Synergy string will be appended.
(XML_STRING_TYPE)

synergy_string

The Synergy string to append to the XML string. (a)

length

(optional) The length of the Synergy string. (n)

Discussion

When you call %XML_STRING_APPEND, a copy of the Synergy string data is added to the end of the specified XML string. You can use %XML_STRING_APPEND to append a Synergy alpha string to an XML string created with %XML_STRING_CREATE.

This function is defined as ^VAL.

Examples

Assuming we have the following:

```
string1      , a3
xmlstring    , XML_STRING_TYPE

xmlstring = %xml_string_create
xcall xml_string_append(xmlstring, "hey ")
string1 = "you"
xcall xml_string_append(xmlstring, string1)
```

the contents of **xmlstring** become “hey you”.

%XML_STRING_APPENDDYN – Append one XML string to another

status = %XML_STRING_APPENDDYN (*dest_string*, *source_string*)

Return value

status

XML_SUCCESS if the string was appended successfully.

Arguments

dest_string

The XML string instance to which the specified source string will be appended.
(XML_STRING_TYPE)

source_string

The XML string instance to append to the specified destination string.
(XML_STRING_TYPE)

Discussion

%XML_STRING_APPENDDYN appends a dynamic XML string to the end of an existing dynamic XML string.

This function is defined as ^VAL.

%XML_STRING_APPENDHANDLE – Append Synergy handle contents to an XML string

status = %XML_STRING_APPENDHANDLE (*xml_string*, *handle*, *size*)

Return value

status

XML_SUCCESS if the handle contents were appended successfully.

Arguments

xml_string

The XML string instance to which the string will be appended. (**XML_STRING_TYPE**)

handle

The Synergy handle that contains a string to be appended. (**D_ADDR**)

size

The size of the string to append. (**n**)

Discussion

%XML_STRING_APPENDHANDLE appends a handle's contents (for example, the contents of a D_ADDR handle returned from the HTTP document transport API) to the end of the specified XML string.

This function is defined as ^VAL.

Examples

See the CodeExchange at <http://www.synergex.com>.

%XML_STRING_CLEAR – Clear the contents of an XML string

status = %XML_STRING_CLEAR(*xml_string*)

Return value

status

XML_SUCCESS if the string was cleared successfully.

Arguments

xml_string

The XML string instance to clear. (**XML_STRING_TYPE**)

Discussion

Clearing the contents of an XML string is useful in a loop processing scenario.

This function is defined as ^VAL.

%XML_STRING_COPY – Copy one XML string to another

status = %XML_STRING_COPY (*destid*, *sourceid*)

Return value

status

XML_SUCCESS if the copy is successful.

Arguments

destid

The destination string ID. (XML_STRING_TYPE)

sourceid

The source string ID. (XML_STRING_TYPE)

Discussion

The element object handle in *destid* must already exist before %XML_STRING_COPY is called; it is not created automatically.

This function is defined as ^VAL.

%XML_STRING_CREATE – Create an XML string instance

xml_string = %XML_STRING_CREATE

Return value

xml_string

The XML string instance created, or 0 if not successful. (XML_STRING_TYPE)

Discussion

%XML_STRING_CREATE creates an empty XML string. This is the first step you must take before you can parse XML data that arrives in a Synergy program as in-memory data, such as an alpha string or a D_ADDR handle returned from the HTTP document transport API. You would then call one of the append routines (%XML_STRING_APPEND, %XML_STRING_APPENDDYN, or %XML_STRING_APPENDHANDLE) to add the data to the XML string you just created.

This function is defined as ^VAL.

Examples

See the CodeExchange at <http://www.synergex.com>.

%XML_STRING_DELETE – Release an XML string instance

status = %XML_STRING_DELETE(*xml_string*)

Return value

status

XML_SUCCESS if the string was released successfully.

Arguments

xml_string

The XML string instance to release. (XML_STRING_TYPE)

Discussion

Call %XML_STRING_DELETE to release an XML string when you no longer need it. For example, you can release a string after you've passed it into %XML_PARSER_PARSESTRING. Strings created with %XML_STRING_CREATE or %XML_DOC_TOSTRING should be released with %XML_STRING_DELETE.

This function is defined as ^VAL.

Examples

See the CodeExchange at <http://www.synergex.com>.

%XML_STRING_GETHANDLE – Get the memory handle of an XML string

handle = %XML_STRING_GETHANDLE (*xml_string*)

Return value

handle

The memory handle of the specified string. (**D_ADDR**)

Arguments

xml_string

The XML string instance whose handle will be returned. (**XML_STRING_TYPE**)

Discussion

When processing XML data in a Synergy program, you will probably want to change the XML string into a Synergy memory handle using %XML_STRING_GETHANDLE to get the handle and %XML_STRING_GETSIZE to get its size. You can then use the handle and size in your Synergy program as needed. (For example, you might pass them into the HTTP document transport API.)

This function is defined as ^VAL.

Examples

See the CodeExchange at <http://www.synergex.com>.

%XML_STRING_GETSIZE – Get the size of an XML string

size = %XML_STRING_GETSIZE(*xml_string*)

Return value

size

The size of the specified string. (**i4**)

Arguments

xml_string

The XML string instance whose size will be returned. (**XML_STRING_TYPE**)

Discussion

When processing XML data in a Synergy program, you will probably want to change the XML string into a Synergy memory handle using %XML_STRING_GETHANDLE to get the handle and %XML_STRING_GETSIZE to get its size. You can then use the handle and size in your Synergy program as needed. (For example, you might pass them into the HTTP document transport API.)

This function is defined as ^VAL.

Examples

See the CodeExchange at <http://www.synergex.com>.

19

Synergy Windowing API

The Synergy windowing API is not available on Synergy .NET.

General Windowing Concepts 19-2

Discusses the general concepts you need to know about the Synergy windowing API.

The Windowing Routines 19-13

Contains specification pages for the following windowing subroutines and functions:

W_AREA – Manipulate the processing and display areas	19-16
W_BRDR – Control window borders, titles, and their attributes	19-26
W_CAPTION – Retrieve or load the application window caption	19-33
W_DISP – Control window display and input functions	19-34
W_EXIT – Exit the windowing system	19-47
W_FLDS – Manipulate field sets within a window	19-48
W_INFO – Get information about the windowing system	19-59
%W_INFO – Get information about the windowing system	19-68
W_INIT – Initialize the windowing system	19-73
W_PROC – General window processing	19-75
W_RESTORE – Restore a window that was previously compressed	19-93
W_SAVE – Save a window in compressed form	19-95
W_UPDT – Update the physical screen	19-97

Error Processing in the Windowing System 19-98

Describes how to detect and prevent runtime errors in a program that uses the windowing API.

Converting Your Code to Use the Windowing API 19-101

Describes the straight conversion method of converting your application to use the windowing API.

General Windowing Concepts

The subroutines in the Synergy windowing API enable you to add windows to your applications. Menu windows, bordered displays, and pop-up help and error messages help make your applications more attractive and easier for users.

The UI Toolkit uses these low-level windowing routines as the foundation for its subroutines, which are more user-friendly. For more information about the Toolkit, refer to the [UI Toolkit Reference Manual](#) or ask your Synergy/DE account manager.

External subroutines and their functions

The windowing API is a group of external subroutines. All of the subroutines except the simplest have several possible functions, and many of those functions have their own arguments. For example, `W_BRDR`, which controls window borders, has functions that enable you to alter a border's display attributes. You can

- ▶ change the border's line drawing characters.
- ▶ set the color of the border.
- ▶ turn the border on or off.
- ▶ set the window title and its display attributes, color, and placement.
- ▶ turn the window title on or off.

Window IDs and names

Most of the window functions require a window ID argument that identifies the window you want to access. When you create a window, you must supply a numeric variable that will be loaded with the numeric window ID. The Synergy Language system then assigns a numeric ID to each window and loads the value in the numeric variable, so you can easily access each window by its variable without having to keep track of the actual number assigned to it. Although a **d4** or **i2** ID variable is usually large enough to contain the assigned number, we recommend that you use an **i4** variable for faster access. Window IDs can be as large as 1024 characters.

When you create a window, you must also supply a name for that window. For example, you might assign the name "Help" to a window that will contain help information or "Menu" to a window that will contain a menu. By assigning the window a generic name such as "Help" or "Menu," you can later search for a window by its name with the `WP_FIND` function. The name and the `WP_FIND` function are especially helpful when you chain to a new program; you can determine if the window you need exists before you create a new one.

Window I/O

The windowing system is conceptually divided into two elements: a logical screen per window and the physical screen. When you modify a window or its contents, the changes affect its logical screen, which is simply a group of memory locations to store information about the windows. The change doesn't affect the actual display, unless the window is visibly placed and you update the screen.

You can change information in a window without having the window displayed. But if you want to see these changes on the physical screen, you must first update the screen. In most situations, the windowing system's implicit screen updating capabilities automatically update the screen when it is appropriate.

Screen updates

The screen is normally updated in five ways:

- ▶ Explicitly with `W_UPDT`
- ▶ Explicitly with the `WD_UPDT` function of `W_DISP`
- ▶ Implicitly with `W_EXIT`
- ▶ Implicitly when the program accepts terminal input with the window I/O functions `WD_READS`, `WD_ACCEPT`, `WD_GETS`, and `WF_INPUT`. These functions automatically update the screen before they accept input.
- ▶ Implicitly when the `WD_BELL` function is used to sound the bell

The screen is also implicitly updated by the `WA_INSERT` and `WA_SCROLL` functions when hardware scrolling is in effect or `WD_WRITES` if a scroll is required.

A screen update simply copies the visible changes in the placed windows to the physical screen. Unless you request a screen redraw with the `WP_REDRAW` function, only the information that has changed is changed on the physical screen; information that was already displayed remains on the physical screen as long as the new information doesn't occupy the same space.

Window areas

A window has three areas:

- ▶ The total window area, which is the size of the entire window and is specified by the `WP_CREATE` function
- ▶ The display area, which is the part of the window that is visible on the screen and is a subset of the total area
- ▶ The processing area, which is the part of the window that is modified by routines such as `W_AREA` and is also a subset of the total area

When you create a window, the default display area is as large as the total window area. The border is drawn around the display area, not inside it. You can change the size of the display area by using the `WA_COPY` function to copy the processing area's parameters to the display area. You can also copy the total window area's parameters to the display area to restore the original display area size.

On creation, the processing area, like the display area, is equal to the total window area. You can change the size of a processing area with three functions: the `WA_SET` function, which explicitly sets the size of the processing area; the `WA_CHANGE` function, which changes the size of the processing area relative to the current size; and the `WA_COPY` function, which can copy the parameters of the display area or the total window area to the processing area.

Window placement and occlusion

You can occlude, or overlap, windows with other windows by placing one window over all or part of another window. The last window placed on the screen is the highest in the placement order and occludes the portion of any previously placed window that it overlaps. When a window is occluded, the part that lies beneath the top window is obscured. When the top window is moved or removed, the obscured portion is visible again.

When you change the data in an occluded window, those changes that aren't covered up are reflected after the next screen update. Changes to the occluded portion of the window are visible when the window is no longer occluded.

Overlay windows

You can overlay windows, which is similar to overlaying a Synergy Language record. The data, window size, and data attributes of the new window are the same as the overlaid window. However, you can alter all of the other window attributes, such as border, title, and placement.

Terminal I/O

Once you initialize the windowing system, all terminal I/O must pass through the system if you want the system to recognize the data. In other words, you must use the window display subroutine, `W_DISP`, to receive information from the terminal and display it in a window. The one exception is the output of nonprinting characters.

You can use the other Synergy Language statements, such as `DISPLAY` and `READS`, for terminal I/O while the windowing system is running, but the windowing system won't have access to any data obtained outside the system. We recommend that you avoid mixing window and nonwindow displays on the same screen.

Attributes

Many of the window functions enable you to specify display attributes, such as `ATTR_BOLD`, `ATTR_RVRS`, and `ATTR_BLNK`. Since various functions require the same attribute specifications, we have described them in [“Attributes” on page 19-13](#).

Colors and the color palette

To support color for the windowing subroutines, Synergy Language uses a color palette and, on Windows, its own set of color definitions.

The *color palette* is made up of 16 pairs of color settings. These pairs are called *color palette entries*, are numbered 1 through 16, and include a foreground color setting (which is used for text) and a background color setting. Most windowing subroutines that set or retrieve colors accept only color palette entry numbers (1 through 16) as arguments. For example, to instruct `W_FLDS` to change the color of a field, you must pass a color palette entry number as the *color* argument to `WF_COLOR`. (`W_FLDS`, a `WF_COLOR` function, applies the background color for the palette entry to the field’s background, and it applies the foreground color to text that appears in the field.)

On Windows, the foreground and background settings for color palette entries correspond to Synergy color definitions, and the Synergy runtime loads a default color palette and default color definitions into memory when it starts. See [“Colors and the color palette on Windows” on page 19-6](#) for more information.

On UNIX and OpenVMS, your application will have color only if you create a color palette by setting the `WNDC` environment variable. See [“The color palette on UNIX and OpenVMS” on page 19-9](#).

Note the following:

- ▶ UI Toolkit also uses the color palette and Synergy colors. For information on how UI Toolkit uses these and on UI Toolkit routines that can modify them, see [“Customizing the Look of Your Application”](#) in the “Customizing UI Toolkit” chapter of the *UI Toolkit Reference Manual*.
- ▶ The color palette and the Synergy color definitions cannot be used with the Synergy ActiveX API or the Synergy Windows Printing API. To specify colors for these, use RGB triplets. See [chapter 10, “Synergy ActiveX API,”](#) and [chapter 19, “Synergy Windowing API,”](#) for information.

Colors and the color palette on Windows

On Windows, a default color palette (with 16 default color palette entries) and a default set of Synergy color definitions (color 0 through color 511) are loaded into memory when the Synergy runtime starts. The default color palette contains the following entries:

Color palette entry number	Foreground (text) color	Background color
1	264	271
2	0	15
3	264	271
4	0	15
5 through 8	0	7
9	264	276
10 through 15	0	7
16	0	15

You can override the default color palette entries by using the PALETTE setting or the WP_PALET function for W_PROC. For more information, see [PALETTE](#) in the “Environment Variables” chapter of *Environment Variables and System Options* and WP_PALET on [page 19-87](#) of this manual.

The foreground and background settings for color palette entries correspond to Synergy color definitions, which are RGB triplets assigned to color numbers. There are three types of Synergy colors:

- ▶ User colors (color 0 through 255)
- ▶ System colors (color 256 through 285)
- ▶ Reserved colors (color 286 through 511)

User colors

Synergy colors 0 through 255 are *user colors*, which are colors you can change by using the COLOR n setting (see [COLOR \$n\$](#) in the “Environment Variables” chapter of *Environment Variables and System Options*) or by using the Renditions option for Proto or Synergy Control Panel, which uses the Toolkit routine U_EDITREND. (For information, see [U_EDITREND](#) in the “Utility Routines” chapter of the *UI Toolkit Reference Manual*.)

The following table lists the default user colors on Windows:

Default User Colors			
Color number	Hexadecimal RGB value	Decimal RGB value	Name
0	0x00,0x00,0x00	0,0,0	Black
1	0x00,0x00,0xFF	0,0,255	Blue
2	0x00,0xFF,0x00	0,255,0	Lime
3	0x00,0xFF,0xFF	0,255,255	Cyan
4	0xFF,0x00,0x00	255,0,0	Red
5	0xFF,0x00,0xFF	255,0,255	Fuchsia
6	0xFF,0xFF,0x00	255,255,0	Yellow
7	0xFF,0xFF,0xFF	255,255,255	White
8	0x80,0x80,0x80	128,128,128	Gray
9	0x00,0x00,0x80	0,0,128	Navy
10	0x00,0x80,0x00	0,128,0	Green
11	0x00,0x80,0x80	0,128,128	Teal
12	0x80,0x00,0x00	128,0,0	Maroon
13	0x80,0x00,0x80	128,0,128	Purple
14	0x80,0x80,0x00	128,128,0	Olive
15	0xC0,0xC0,0xC0	192,192,192	Silver
16 through 255	0x00,0x00,0x00	0,0,0	Black

System colors

Synergy colors 256 through 285 are *system colors*, which correspond to Windows system colors and can be changed only through Windows mechanisms (Windows Control Panel, for example). You cannot change these with the windowing API (or even with UI Toolkit routines). The following table lists the default system colors:

Default System Colors		
Color number	Mnemonic	Windows UI element
256	SYSCOLOR_SCROLLBAR	Scrollbar, background
257	SYSCOLOR_BACKGROUND	Desktop (if no wallpaper)
258	SYSCOLOR_ACTIVECAPTION	Active Title Bar, background
259	SYSCOLOR_INACTIVECAPTION	Inactive Title Bar
260	SYSCOLOR_MENU	Menu
261	SYSCOLOR_WINDOW	Window, background
262	SYSCOLOR_WINDOWFRAME	Active Window Border and Inactive Window Border
263	SYSCOLOR_MENUTEXT	Menu, font settings
264	SYSCOLOR_WINDOWTEXT	Window, font settings
265	SYSCOLOR_CAPTIONTEXT	Active Title bar, font settings
266	SYSCOLOR_ACTIVEBORDER	Active Window Border
267	SYSCOLOR_INACTIVEBORDER	Inactive Window Border
268	SYSCOLOR_APPWORKSPACE	Desktop
269	SYSCOLOR_HIGHLIGHT	Selected Items, background
270	SYSCOLOR_HIGHLIGHTTEXT	Selected Items, font settings
271	SYSCOLOR_3DFACE	3D Objects, face color
272	SYSCOLOR_3DSHADOW	3D Objects, shadow
273	SYSCOLOR_GRAYTEXT	Menu, disabled settings
274	SYSCOLOR_BTNTEXT	3D Objects, font settings

Default System Colors (Continued)		
Color number	Mnemonic	Windows UI element
275	SYS.COLOR_INACTIVECAPTIONTEXT	Inactive Title Bar, text
276	SYS.COLOR_3DHIGHLIGHT	3D Objects, highlight color
277	SYS.COLOR_3DDARKSHADOW	3D Objects, dark shadow
278	SYS.COLOR_3DLIGHT	3D Objects, light color
279	SYS.COLOR_INFOTEXT	Tool Tip, font settings
280	SYS.COLOR_INFOBK	Tool Tip, background color
281	SYS.COLOR_HOTLIGHT	Selected Items, background color
282	SYS.COLOR_GRADIENTACTIVECAPTION	Active Title Bar and Inactive Title Bar, Color2 (right-side color of gradient captions)
283	SYS.COLOR_GRADIENTINACTIVECAPT	Active Title Bar and Inactive Title Bar, Color (left-side color of gradient captions)
284	SYS.COLOR_MENUHILIGHT	Selected Items (highlight color for menus when shown flat)
285	SYS.COLOR_MENUBAR	Menu (color when shown flat)

Reserved colors

Synergy colors 286 through 511 are reserved for future Windows system colors. These are all set to black (0,0,0) and cannot be changed. (As system colors are added to Windows in the future, Synergex™ will assign these colors to the new Windows system colors.)

The color palette on UNIX and OpenVMS

You can enable color on UNIX and OpenVMS by setting the WNDC environment variable. The assigned string defines the default color palette. The format of the command varies according to your operating system and shell (see [WNDC](#) in the “Environment Variables” chapter of *Environment Variables and System Options* for details). For example, on UNIX, the command looks like this:

```
WNDC=palette1, palette2, ..., palette16
```

You can override color palette entries set by WNDC with subsequent WNDC statements or by using the WP_PALETTE function for W_PROC. For information on WP_PALET, see WP_PALET on [page 19-87](#).

On UNIX, if colors are not defined by the system's **termcap** file, use the ANSICOLOR environment variable to instruct the runtime to use its own internal ANSI color sequences. For information, see [ANSICOLOR](#) in the “Environment Variables” chapter of *Environment Variables and System Options*.

Windowing subroutines on Windows platforms

Some of the windowing subroutines may exhibit differences between character-based environments and graphical environments. Refer to the “[Statements, Functions, and Subroutines](#)” chapter of your *Professional Series Portability Guide* for information about these differences.

We recommend that you use a fixed font when using the Synergy windowing API on Windows platforms.

Windowing API and UI Toolkit

Although most of the windowing subroutines work normally within a UI Toolkit environment, some behave differently when working under UI Toolkit. Because Toolkit manipulates the windowing subsystem extensively, some windowing subsystem parameters may be reported with unexpected values.

For example, Toolkit may create additional windows, or additional fields in a window, thereby causing the metrics returned by W_INFO to be misleading. If a window is loaded with five fields in Synergy/DE 7, Toolkit will create between 10 and 15 fields: one for each data field, one for each prompt field, and possibly one for each format string that may be associated with the field. In earlier versions of Synergy/DE, the same window would have reported 5 to 15 fields, depending on the number of user-definable prompts and/or formats.

Follow these guidelines when using the Synergy windowing API with UI Toolkit:

- ▶ The number of windows in the environment is not always equal to the number of windows loaded.
- ▶ The number of fields within a window is not always the number of fields in the script file created by the IB_ routines.
- ▶ Avoid directly manipulating a field for your own purposes. Toolkit assumes that it owns the field and may not recognize your changes.
- ▶ Avoid using a user data set name that starts with “DTK_”. These are reserved for use by Toolkit.

- ▶ Avoid using `W_PROC` to place, remove, or delete windows. Doing so will circumvent some of the Toolkit logic, therefore causing unexpected results.
- ▶ If you require access to particular data from Toolkit and the windowing subroutines are not reporting the “correct” value, contact Synergy/DE Developer Support.

WIN

- ▶ When using proportional fonts, any change to color or attributes within the window text will cause justification to occur at the pixel coordinate of the logical (cell-based) column. This may produce a confusing display.
-
- ▶ Synergex reserves the right to use the windowing routines in the most appropriate way for Toolkit. Do not rely on information such as number of windows, number of fields, and current field number to remain constant. The internals of UI Toolkit are not published, and therefore, changes to the internals are not documented. This means that you will not get a “this may break your code” warning when something changes.

Window definitions

The windowing system uses a series of definitions to interpret windowing code. These definitions reside in files named **wxxx.def**, where *xxx* corresponds to the name of the windowing subroutine. For example, **warea.def** contains definitions particular to `W_AREA`, **wbrdr.def** contains definitions particular to `W_BRDR`, and so forth. These files are included with your Synergy Language distribution.

We also provide a file named **windows.def** with your Synergy Language system, which includes all of the Synergy windowing subroutine definitions. We recommend that you `.INCLUDE windows.def` rather than the individual **wxxx.def** files.

Every routine that uses windowing subroutines must `.INCLUDE` the **windows.def** file. For example:

```
.include "WND:windows.def"
```

Hardware scrolling

Hardware scrolling, which uses terminal scrolling capabilities to move data on the screen, only occurs under the following conditions:

- ▶ The display area is the same as the processing area.
- ▶ The display area is the same as the width of the screen, with or without a border.
- ▶ The window is not occluded.

UNIX

- ▶ You're on a VT-style (VT100, VT200, and so forth), xterm, or other type of terminal with the appropriate **termcap/terminfo** information to set the scrolling region and scroll up or down (for example, with **cs**, **sf**, and **sr** entries).
- ▶ The `WA_INSERT` or `WA_SCROLL` functions are used.

Vertical scrolling only requires the scrolling area to be a subset of the rows in the display area, as long as the whole display area is visible. We automatically turn on the scrolling capabilities by looking at the `TERM` environment variable when `W_INIT` is called.

Vertical scrolling is automatically turned on when `TERM` equals one of the following values:

```
vt1xx
vt2xx
vt3xx
vt4xx
vt5xx
xterm
```

where `xx` represents any character(s) you want.

User data sets

The windowing system has a user data set facility. A user data set is a one-dimensional alpha array that is associated with a window and maintained within the window data area by the windowing system. You can completely control its parameters and contents.

To create, put data into, or get data from a user data set field, use the `WF_USER`, `WF_UPUT`, and `WF_UGET` functions in the `W_FLDS` subroutine. Use the `WI_USER` function to return information about a user data set.

The Windowing Routines

This section describes each windowing routine and its subfunctions. Each windowing routine begins with “W_” (or “%W_”).



The W_INFO subroutine can be called as the intrinsic function %W_INFO. See %W_INFO on page 19-68 for more information.

Each windowing subfunction begins with “WX_,” where *X* is the first letter of the subroutine name to which the function belongs. For example, all of the functions for W_AREA begin with “WA_,” all of the functions for W_BRDR begin with “WB_,” and so forth.

Attributes

Many of the windowing routines have a function that alters the display attributes of a window element, such as the border, title, text, or processing area. After the routine-specific function name (such as WD_ATTR or WB_ATTR), all of these functions use the same attribute specifications. An attribute specification has the following format:

... WX_ATTR, *function* + *option* [+ *option* ...]

function

One of the following functions, which are described in the “Functions” section below:

ATTR_SET	Set attributes.
ATTR_CHNG	Change existing attributes.
ATTR_LOAD	Set listed attributes, ignore “skip” attributes, and clear the others.
ATTR_CLR	Clear attributes.

option

One of the following attributes:

ATTR_BOLD	Boldface
ATTR_BLNK	Blinking
ATTR_ITAL	Italic
ATTR_RVRS	Reverse video
ATTR_UNDR	Underline
ATTR_ALL	All of the above attributes

For `ATTR_LOAD`, *option* can also be one of the following skip attributes:

SKIP_BLNK	Skip blinking.
SKIP_BOLD	Skip boldface.
SKIP_ITAL	Skip italic.
SKIP_RVRS	Skip reverse video.
SKIP_UNDR	Skip underline.

For example, to set reverse video in a window border, use `W_BRDR` with the `WB_ATTR` function, the `ATTR_SET` function, and the `ATTR_RVRS` option. Your statement should look something like this:

```
xcall w_brdr(wndw_1, WB_ATTR, ATTR_SET + ATTR_RVRS)
```

To display some text in boldface type, you might use the statement:

```
xcall w_disp(wndw_1, WD_ATTR, ATTR_LOAD + ATTR_BOLD, "Displayed in bold")
```

All input characters that are echoed have the current default display attributes (including color) of the window in which the input is occurring.

You can specify as many attributes as you like, as long as they are separated by plus signs.

Because italic is the Windows implementation of blinking (and blinking is the UNIX and OpenVMS implementation of italic), `ATTR_BLNK` is synonymous with `ATTR_ITAL`, and `SKIP_BLNK` is synonymous with `SKIP_ITAL`.

Functions

ATTR_SET

`ATTR_SET` turns on the attribute options that follow. You can set more than one attribute at a time. If you specify an attribute that's already set, it remains set. Any attribute not specified remains unchanged. For example:

```
WD_ATTR, ATTR_SET + ATTR_RVRS + ATTR_UNDR
```

causes displayed text to include the underline and reverse video attributes.

ATTR_CHNG

`ATTR_CHNG` toggles the attributes that are listed but has no effect on the attributes that aren't listed. If the attribute is set, it is turned off; if it is not set, it is turned on. For example, if the text was previously displayed in boldface but now you want it to blink, use this function:

```
WD_ATTR, ATTR_CHNG + ATTR_BOLD + ATTR_BLNK
```

ATTR_LOAD

ATTR_LOAD sets the listed attributes, ignores any skip attributes, and clears or resets the attributes that aren't listed. For example, if you aren't sure what the current title attributes are, but you know you want them to be reverse video, you could use the following function to set ATTR_RVRS and clear all other attribute options:

```
WB_TATTR, ATTR_LOAD + ATTR_RVRS
```

The following example turns on reverse video, turns off boldface and blinking, and does nothing to the underline attribute.

```
WA_ATTR, ATTR_LOAD + ATTR_RVRS + SKIP_UNDR
```

ATTR_CLR

ATTR_CLR turns off the attribute options that follow. You can clear more than one attribute at a time. If you specify an attribute that is already cleared, it remains cleared. Any attribute that is not specified remains unchanged. For example:

```
WB_ATTR, ATTR_CLR + ATTR_ALL
```

clears all the border attributes.

W_AREA – Manipulate the processing and display areas

```
xcall W_AREA(id, function[, ...])
```

Arguments

id

A variable that contains the window ID. The ID must be the same one that was returned by the WP_CREATE function (see [page 19-77](#)). (n)

function

One or more of the following functions and their arguments, which are described in the “Functions” section below:

WA_ATTR	Alter the processing area’s attributes.
WA_BOX	Draw a box within the processing area.
WA_CHANGE	Change the size and position of the processing area relative to the current values.
WA_COLOR	Set the color of the processing area.
WA_COPY	Copy parameters to or from the display or processing area settings.
WA_CORNER	Position the current window to one corner of the processing area.
WA_FILL	Fill the processing area with characters.
WA_INSERT	Insert a line at the top or bottom of the scrolling area.
WA_SCROLL	Scroll the contents of the processing area.
WA_SET	Set the size and position of the processing area.

Discussion

W_AREA is used to modify the size, position, and contents of a window’s processing and display areas. The initial default processing and display areas are the same size as the window.

Functions

WA_ATTR, *attributes*

WA_ATTR alters the attributes of the processing area's contents. See [“Attributes” on page 19-13](#) for a description of the standard window attributes.

Examples

In this example, we fill the processing area with letters. Then we set the processing area so its upper-left corner is at position (3,5) within the window and its dimensions are 6 rows by 14 columns. The WA_BOX function draws a box around the area. Note that the line-drawing characters are part of the area, which means that if we want to change the attributes of the area's contents without changing the attributes of the box, we must shrink the area. The WA_CHANGE function moves the upper-left corner of the area by 1 row and 1 column and reduces the horizontal and vertical dimensions by 2. We also change the area's attributes to reverse video.

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 10, 22)
    xcall w_area(wndw_1, WA_FILL,
&        "abcdefghijklmnopqrstuv")
    xcall w_proc(WP_PLACE, wndw_1, 11, 28)
    xcall w_area(wndw_1, WA_SET, 3, 5, 6, 14, WA_BOX,
&        WA_CHANGE, 1, 1, -2, -2)
    call pause
    xcall w_area(wndw_1, WA_ATTR, ATTR_SET + ATTR_RVRS)
    call pause
    xcall w_exit
    stop
pause,
    xcall w_updt
    sleep 1
    return
end
```

WA_BOX

WA_BOX draws a box at the edge of the processing area. The line-drawing characters of the box are included in the area; they are not outside the area.

Examples

In the routine below, we set an area and draw a box around it before setting the next area and boxing it. Notice that we can overlap areas.

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 16, 40,
&                WP_PLACE, wndw_1, 8, 20)
    xcall w_area(wndw_1, WA_SET, 1, 1, 8, 20, WA_BOX)
    xcall w_area(wndw_1, WA_SET, 1, 21, 8, 20, WA_BOX)
    xcall w_area(wndw_1, WA_SET, 9, 1, 8, 20, WA_BOX)
    xcall w_area(wndw_1, WA_SET, 9, 21, 8, 20, WA_BOX)
    xcall w_area(wndw_1, WA_SET, 5, 11, 8, 20, WA_BOX)
    xcall w_exit
    stop
end
```

WA_CHANGE, row_change, col_change, height_change, width_change

WA_CHANGE makes relative changes to the size of the processing area. *Row_change*, *col_change*, *height_change*, and *width_change* are numeric expressions that are added to the row, column, height, and width values of the current processing area. If the resulting area doesn't lie entirely within the window, it is automatically limited so that it does fit inside the window.

Examples

The program below fills the entire area with asterisks, then sets the area to a smaller area and boxes it. WA_CHANGE sets a new area that doesn't retain the attributes of the previous area.

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4
```



```

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 10, 22)
    xcall w_area(wndw_1, WA_FILL, "**")
    xcall w_proc(WP_PLACE, wndw_1, 11, 28)
    xcall w_area(wndw_1, WA_SET, 3, 5, 6, 14, WA_BOX)
    call pause
    xcall w_area(wndw_1, WA_CHANGE, 1, 1, -2, -2, WA_FILL, " ")
    call pause
    xcall w_exit
    stop

pause,
    xcall w_updt
    sleep 1
    return
end

```

WA_COLOR, *color*

WA_COLOR resets the color for the processing area to *color*, which is a numeric expression that specifies a palette number between 1 and 16. See [“Colors and the color palette” on page 19-5](#) for more information about color.

Examples

The following routine fills a window with asterisks, then resets the processing area and boxes it. The WA_COLOR function changes the color from the default color to the background and character colors specified in palette number 2.

```

.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_PALET, 2, 1, 3)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 10, 22)
    xcall w_area(wndw_1, WA_FILL, "**")
    xcall w_proc(WP_PLACE, wndw_1, 11, 28)
    xcall w_area(wndw_1, WA_SET, 3, 5, 6, 14, WA_BOX)
    call pause
    xcall w_area(wndw_1, WA_COLOR, 2)
    call pause

```

```

        xcall w_exit
        stop

    pause,
        xcall w_updt
        sleep 1
        return
    end

```

WA_COPY, *option*

WA_COPY copies parameters to or from the row, column, height, and width settings of the display or processing area, where *option* is one of the options listed below:

WAC_WTOP	Copy window area size settings to the processing area settings.
WAC_DTOP	Copy display area size settings to the processing area settings.
WAC_WTOD	Copy window area size settings to the display area settings.
WAC_PTOD	Copy processing area size settings to the display area settings.

This function doesn't copy one area's data or attributes to another area, only the size and placement settings. For example, if we use the option WAC_PTOD, the processing area's settings are copied to the display area's settings, which means that the display area is changed so that it is the same size and in the same position as the processing area. The option WAC_WTOP copies the window area's settings to the processing area, which means that the processing area becomes the same size and is placed in the same position as the window area.

Examples

In the example below, we split the window into four quadrants plus a center section and fill each area with numbers. Then we put a second window behind the first window and fill it with asterisks. (To put the second window behind the first placed window, we place the second window first.)

Next, we set a processing area of 8 rows by 20 columns and use the option WAC_PTOD to copy those settings to the display area. The effect of this option is that the display area shrinks to the size of the processing area, and consequently, the area displays only one quadrant, plus the "overlapping" number 5s. We repeat this function, setting the display area to be equal to the different processing areas.

At the end of the program, we use the WAC_WTOD option, which copies the window area's settings to the display area. This action enlarges the display area to its original size so that all of **wndw_1** is displayed again.

```

.define MAXWINS,    15
.define WNDCHNL,    1
.include "WND:windows.def"

```

```

record
    wndw_1          ,i4
    wndw_2          ,i4
proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 16, 40)
    xcall w_area(wndw_1, WA_SET, 1, 1, 8, 20,
&                WA_FILL, "1", WA_SET, 1, 21, 8, 20,
&                WA_FILL, "2", WA_SET, 9, 1, 8, 20,
&                WA_FILL, "3", WA_SET, 9, 21, 8, 20,
&                WA_FILL, "4", WA_SET, 5, 11, 8, 20,
&                WA_FILL, "5")
    xcall w_proc(WP_CREATE, wndw_2, "window2", 20, 44)
    xcall w_area(wndw_2, WA_FILL, "**")
    xcall w_proc(WP_PLACE, wndw_2, 4, 36, WP_PLACE, wndw_1, 6, 38)
    call pause
    xcall w_area(wndw_1, WA_SET, 1, 1, 8, 20, WA_COPY, WAC_PTOD)
    call pause
    xcall w_area(wndw_1, WA_SET, 1, 21, 8, 20, WA_COPY, WAC_PTOD)
    call pause
    xcall w_area(wndw_1, WA_SET, 9, 1, 8, 20, WA_COPY, WAC_PTOD)
    call pause
    xcall w_area(wndw_1, WA_COPY, WAC_WTOD)
    call pause
    xcall w_exit
    stop

pause,
    xcall w_updt
    sleep 1
    return
end

```

WA_CORNER, *option*

WA_CORNER moves the current window position to one of the processing area's corners. This feature is useful, for example, if you want to add text to the top of the processing area. You could specify that you want to add text beginning at the upper-left corner, without having to know the exact row and column position within that processing area. *Option* is one of the options listed below:

WAC_ULC	Position to upper left.
WAC_URC	Position to upper right.
WAC_LLC	Position to lower left.
WAC_LRC	Position to lower right.

Examples

In the following routine, we create a window and fill it with asterisks. Then we set a smaller processing area and box it. To fill the processing area with another character without wiping out the box, we need to change the processing area and then fill it. Next, we use the `WA_CORNER` function to position the current window position to the upper-left corner and then the upper-right corner before displaying text. Notice that when you position to the upper-right corner and want to display text, you must move the display position backwards (in this case, eight columns back) to accommodate the text.

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 11, 51)
    xcall w_area(wndw_1, WA_FILL, "*")
    xcall w_proc(WP_PLACE, wndw_1, 12, 12)
    xcall w_area(wndw_1, WA_SET, 3, 10, 7, 35, WA_BOX,
&          WA_CHANGE, 1, 1, -2, -2, WA_FILL, " ",
&          WA_CORNER, WAC_ULC)
    xcall w_disp(wndw_1, "top left")
    xcall w_updt
    xcall w_area(wndw_1, WA_CORNER, WAC_URC)
    xcall w_disp(wndw_1, WD_MOVE, 0, -8, "top right")
    xcall w_exit
    stop
end
```

WA_FILL, *alpha_string*

`WA_FILL` fills the processing area with a string of characters specified by *alpha_string*. To clear a processing area, use a blank character as the *alpha_string* (`WA_FILL, " "`). The characters are written using the window's current default attributes.

Examples

In the following example, we set a processing area and fill it before setting a new processing area and filling it. When we're done, we clear the processing area that was set.

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4
```

```

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 16, 40)
    xcall w_proc(WP_PLACE, wndw_1, 8, 20)
    xcall w_area(wndw_1, WA_SET, 1, 1, 8, 20,
&                WA_FILL, "1", WA_SET, 1, 21, 8, 20,
&                WA_FILL, "2", WA_SET, 9, 1, 8, 20,
&                WA_FILL, "3", WA_SET, 9, 21, 8, 20,
&                WA_FILL, "4", WA_SET, 5, 11, 8, 20,
&                WA_FILL, "5")
    xcall w_updt
    sleep 1
    xcall w_area(wndw_1, WA_FILL, " ")
    xcall w_exit
    stop
end

```

WA_INSERT, *placement, line*

WA_INSERT scrolls the processing area up or down one line and inserts a line at the bottom or top of the scrolling area, where *placement* is one of the following options:

- WAI_TOP** Insert a line at the top of the scrolling area.
- WAI_BOTTOM** Insert a line at the bottom of the scrolling area.

Line is an alpha expression whose value is inserted in the scrolling area.

Hardware scrolling only occurs when the WA_INSERT or WA_SCROLL functions are used. Both of these commands do an implicit screen update when hardware scrolling is active.

WA_SCROLL, *direction, count*

WA_SCROLL scrolls the contents of the processing area. *Direction* defines the direction of the text's movement relative to the window, and the value of *count* indicates how many rows or columns to scroll. When you scroll up or down, *count* represents the number of rows. When you scroll left or right, *count* represents the number of columns. The *direction* options are listed below:

- WAS_UP** Scroll from bottom to top.
- WAS_DOWN** Scroll from top to bottom.
- WAS_LEFT** Scroll from right to left.
- WAS_RIGHT** Scroll from left to right.

Hardware scrolling only occurs when the WA_INSERT or WA_SCROLL functions are used. Both of these commands do an implicit screen update when hardware scrolling is active.

UNIX

For non-VT-type terminals, the **termcap/terminfo** bases are accessed for the sequences to set the scrolling region and scroll up or scroll down. If these sequences are not present, hardware scrolling is not enabled.

Examples

In the example below, we use hardware scrolling to scroll an area within an area. Note that after the last **WA_SCROLL** function, when we've returned to the original position, we "lose" the characters that have scrolled out of the area.

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 16, 40)
    xcall w_area(wndw_1, WA_FILL, "1234567890")
    xcall w_proc(WP_PLACE, wndw_1, 8, 20)
    call pause
    xcall w_area(wndw_1, WA_SCROLL, WAS_UP, 3)
    call pause
    xcall w_area(wndw_1, WA_SCROLL, WAS_DOWN, 6)
    call pause
    xcall w_area(wndw_1, WA_SCROLL, WAS_UP, 3)
    call pause
    xcall w_exit
    stop

pause,
    xcall w_updt
    sleep 1
    return
end
```

WA_SET, *row, column, height, width*

WA_SET defines the processing area. *Row, column, height, width* are numeric expressions that define the position and size of the processing area. The resulting area must lie entirely within the window; if the area you define exceeds the bounds of the window, the system corrects for it by moving the area or truncating it.

You can have only one processing area at a time. If, for example, you want to set five different processing areas and fill them with different characters, as we do in the example for the function WA_FILL, you must set the first processing area and fill it before resetting the processing area.

Examples

See the WA_FILL example on [page 19-22](#), which sets several processing areas.

W_BRDR – Control window borders, titles, and their attributes

`xcall W_BRDR(id, function[, ...])`

Arguments

id

A variable that contains the window ID. (**n**)

function

One or more of the following functions and their arguments, which are described in the “Functions” section below:

- WB_ATTR** Alter the window border’s attributes.
- WB_CHAR** Change the border’s line-drawing characters.
- WB_COLOR** Set the border’s display color.

WIN ---

- WB_DRAGOFF** Turn the drag bar off.
 - WB_DRAGON** Turn the drag bar on.
-
- WB_NOCELL** Display border if it does not require a full character cell for display.
 - WB_OFF** Turn the border off.
 - WB_ON** Turn the border on.
 - WB_PARTIAL** Turn the border on and enable partial borders.

WIN ---

- WB_SBOFF** Disable scroll bars.
 - WB_SBON** Enable scroll bars.
-

UNIX, VMS ---

- WB_TATTR** Alter the title’s attributes.
-
- WB_TCOLOR** Set the title’s display color.
 - WB_TITLE** Set the window’s title.
 - WB_TOFF** Turn off the title display.

WB_TON	Turn on the title display.
WB_TPOS	Position the title.

Discussion

W_BRDR turns a border on or off and modifies window titles and their attributes.

When a window has a border, the border is drawn around the display area and is not considered part of the window's content. If the display area changes, the border is redrawn accordingly. If the screen doesn't have enough room for the border, the border won't be drawn, unless you specify the WB_PARTIAL function to enable partial border displays. (See [page 19-30](#).)

Functions

WB_ATTR, *attributes*

WB_ATTR alters the window border's display attributes. See "[Attributes](#)" on [page 19-13](#) for a description of the standard window attributes.

WB_ATTR is not available in graphical environments. We recommend that you not use this feature if your software will be run on Windows.

Examples

In the example below, we first display the border in reverse video. With the ATTR_CHNG function, we add blinking, so that the attributes are then reverse video and blinking. The ATTR_LOAD function loads the named options (in this case, reverse video) and turns off any options that aren't named (which means that it clears blinking and leaves the reverse video attribute displayed). Finally, we use ATTR_CLR to clear the reverse video.

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 8, 40)
    xcall w_area(wndw_1, WA_FILL, "*")
    xcall w_proc(WP_PLACE, wndw_1, 11, 21)
    call pause
    xcall w_brdr(wndw_1, WB_ATTR, ATTR_SET + ATTR_RVRS)
    call pause
    xcall w_brdr(wndw_1, WB_ATTR, ATTR_CHNG + ATTR_BLNK)
    call pause
    xcall w_brdr(wndw_1, WB_ATTR, ATTR_LOAD + ATTR_RVRS)
```

```
call pause
xcall w_brdr(wndw_1, WB_ATTR, ATTR_CLR + ATTR_RVRS)
call pause
xcall w_exit
stop

pause,
xcall w_updt
sleep 1
return
end
```

WB_CHAR, *horiz_line, vert_line, upper_left, upper_right, lower_left, lower_right*
WB_CHAR, *alpha_expression*

WB_CHAR changes the border’s line-drawing characters. Each argument is an alpha or numeric expression that contains the character for its position: horizontal line, vertical line, upper-left corner, upper-right corner, lower-left corner, and lower-right corner, respectively.

WB_CHAR is not available in graphical environments. We recommend that you not use this feature if your software will be run on Windows.

If a WB_CHAR argument is a numeric expression, the value is interpreted as a value in the table below.

Decimal value	ASCII character
1	Horizontal line
2	Vertical line
3	Upper-left corner
4	Top tee
5	Upper-right corner
6	Left tee
7	Crossing
8	Right tee
9	Lower-left corner
10	Bottom tee

Decimal value	ASCII character
11	Lower-right corner
12–31	System specific
32–255	Corresponding ASCII character

If a WB_CHAR argument is an alpha expression, the value itself is the line-drawing characters. For example, the string “-|++++” defines *horiz_line* as “-”, *vert_line* as “|”, and all the remaining arguments as “+”.

Examples

In the following routine, we first change the line-drawing characters to -, |, +, +, +, and +. (In other words, the corners are plus signs, the horizontal lines are dashes, and the vertical lines are vertical bars.) Next, we change the line-drawing characters back to lines and corners. The decimal values correspond to ASCII line-drawing characters, which are listed in the table above.

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 8, 40)
    xcall w_area(wndw_1, WA_FILL, "1234567890")
    xcall w_proc(WP_PLACE, wndw_1, 11, 21)
    call pause
    xcall w_brdr(wndw_1, WB_CHAR, "-|++++")
    call pause
    xcall w_brdr(wndw_1, WB_CHAR, 1,2,3,5,9,11)
    xcall w_exit
    stop

pause,
    xcall w_updt
    sleep 1
    return
end
```

WB_COLOR, *color*

WB_COLOR sets the window's border display color to *color*, which is a numeric expression that contains a palette number between 1 and 16. See “Colors and the color palette” on page 19-5 for more information about color.

WB_COLOR is not available in graphical environments. We recommend that you not use this feature if your software will be run on Windows.

WB_DRAGOFF

WB_DRAGOFF turns off the drag bar if the border is on. This function is only available on Windows.

WB_DRAGON

WB_DRAGON turns on the drag bar if the border is on. This function is only available on Windows.

WB_NOCELL

WB_NOCELL designates that the window should only have a border if the border does not require a full character cell for display. In other words, the window will have a border in Windows, but not in any other supported environments. You can use this function instead of WB_ON and WB_OFF. (WB_NOCELL and WB_ON/WB_OFF are mutually exclusive.)

WB_OFF

WB_OFF turns off the border, including partial borders. If the placement position was adjusted using WP_PLACE to accommodate a border, the window is shifted back to its original position. (See WP_PLACE on page 19-87 for more information about placing a window's display area.)

WB_ON

WB_ON turns on the border (default). It does *not* override the WB_PARTIAL function; if you want WB_ON functionality after WB_PARTIAL has been set, you must first turn off the border with WB_OFF and then set WB_ON.

WB_PARTIAL

Like WB_ON, WB_PARTIAL turns the border on for the specified window, but it also enables partial border displays if an entire border doesn't fit on the screen. For example, if your window is 20 rows by 80 columns (on a 24 x 80 screen) and WB_PARTIAL is set, Synergy Language still displays the top and bottom borders. Likewise, if your window is too tall for the border to be displayed on the screen, the side borders are still displayed if WB_PARTIAL is set.

WB_PARTIAL is not available in graphical environments. We recommend that you not use this feature if your software will be run on Windows.

window, **WB_SBOFF**[, *scroll_bar*]

WB_SBOFF disables scroll bars in Windows environments, which means the specified scroll bars are not displayed at all. *Window* is the ID of the window and *scroll_bar* is one of the following optional scroll bar identifiers:

WBS_HORZ	Horizontal scroll bar
WBS_VERT	Vertical scroll bar
WBS_BOTH	Both scroll bars (default)

This qualifier is ignored on UNIX and OpenVMS.

window, **WB_SBON**[, *scroll_bar*]

WB_SBON enables the specified scroll bars in Windows environments, where *window* is the ID of the window and *scroll_bar* is one of the following optional scroll bar identifiers:

WBS_HORZ	Horizontal scroll bar
WBS_VERT	Vertical scroll bar
WBS_BOTH	Both scroll bars (default)

When scroll bars are enabled, room is left for them within the window's frame even when the scroll bars are not displayed. Scroll bars are only displayed if the window's display area is smaller than the window area. This qualifier is ignored on UNIX and OpenVMS.

WB_TATTR, *attributes*

WB_TATTR alters the title's display attributes. See [“Attributes” on page 19-13](#) for more information on standard window attributes.

WB_TATTR is not available in graphical environments. We recommend that you not use this feature if your software will be run on Windows.

Examples

In the following example, we first make sure that the border is on by using the WB_ON function, and then we set the border's attributes to reverse video. We then specify a title for the window (the contents of the variable **choice**) and change the attributes of the title to bold. Finally, we position the title in the center of the top border.

```
xcall w_brdr(wndw_1, WB_ON, WB_ATTR, ATTR_SET + ATTR_RVRS,
&          WB_TITLE, choice, WB_TATTR, ATTR_CHNG + ATTR_BOLD,
&          WB_TPOS, WBT_TOP, WBT_CENTER)
```

WB_TCOLOR, *color*

WB_TCOLOR sets the title's display color to *color*, which is a numeric expression that contains a palette number between 1 and 16. See [“Colors and the color palette” on page 19-5](#) for more information about color.

WB_TCOLOR is not available in graphical environments. We recommend that you not use this feature if your software will be run on Windows.

WB_TITLE, *title*

WB_TITLE sets the title for the window, where *title* is an alpha expression or literal that contains the window title. Note that on Windows, WB_TITLE updates the tab text if the window is part of a tab set. (On UNIX and OpenVMS, you must remove and re-add a window to change the text for its tab.)

WB_TOFF

WB_TOFF turns off the title display in the border.

WB_TON

WB_TON turns on the title display in the border.

WB_TPOS, *border, position*

WB_TPOS indicates in which border and at what position within the border the title should be displayed. The following *border* options are available:

WBT_TOP	Top border
WBT_BOTTOM	Bottom border
WBT_LEFT	Left border
WBT_RIGHT	Right border

The following *position* options are available:

<i>numeric expr</i>	Actual position
WBT_BEG	Beginning of the border (left or top)
WBT_END	End of the border (right or bottom)
WBT_CENTER	Centered in the border

The default title position is WBT_TOP and WBT_BEG (top border, left side).

WB_TPOS is not available in graphical environments. We recommend that you not use this feature if your software will be run on Windows.

W_CAPTION – Retrieve or load the application window caption

WIN

xcall W_CAPTION (*function*, *text*)

Arguments

function

One of the following functions:

WC_SET Set new caption (title).

WC_GET Get current caption (title).

text

The caption text to be set or returned (up to 78 characters long). (**a**)

Discussion

This subroutine retrieves or loads the application window caption. We recommend, however, that you use the D_CAPTION argument to the UI Toolkit E_SECT subroutine instead.

W_DISP – Control window display and input functions

```
xcall W_DISP(id, function[, ...])
```

Arguments

id

A variable that contains the window ID. (**n**)

function

One or more of the following functions and their arguments, which are described in the “Functions” section below:

<i>alpha_expr</i>	Print the specified text to the window.
WD_ACCEPT	Receive a character from the terminal.
WD_ANSI	Treat next argument as a string with embedded escape sequences.
WD_ATTR	Modify attributes for the display area.
WD_BELL	Update the screen and ring the terminal bell.
WD_CLEAR	Clear the window to blanks.
WD_CLR	Perform clear processing similar to \$SCR_CLR.
WD_COLOR	Set the current display color.
WD_FIELD	Send a field to the window.
WD_GETS	Get input from the terminal.
WD_HERASE	Erase horizontal lines.
WD_HLINE	Draw a horizontal line.
WD_MOVE	Change the current window position relatively.
WD_POS	Change the current window position absolutely.
WD_READS	Read data from the terminal into the window.
WD_REPEAT	Set the repeat count for the next WD_FIELD function.
WD_UPDT	Update the physical screen.
WD_VERASE	Erase vertical lines.
WD_VLINE	Draw a vertical line.
WD_WAIT	Wait the specified time for input.
WD_WRITES	Display next argument at current position.

Discussion

W_DISP controls window display and input functions.

If the window is placed and visible, W_DISP functions do not change the screen display until the screen is updated with a WD_READS, WD_GETS, WD_ACCEPT, WD_BELL, WD_UPDT, or WF_INPUT function or until you call W_UPDT. (The screen is also updated by WA_INSERT or WA_SCROLL when hardware scrolling is in effect.)

The ACCHR subroutine affects the windowing system for terminating input.

Functions

alpha_expr

This function displays the expression's text to the window at the current window position. The current column position is altered by the length of the alpha field until the field reaches the window's boundary. In other words, if you display the five-character message "Hello" starting at position (1,1), the current window position changes to (1,6) after "Hello" is displayed.

Any text that exceeds the window's boundaries is ignored. The text has the current display attributes for the window, which you can alter with the WD_ATTR function. (See WD_ATTR on [page 19-37](#) for more information.)

Examples

The example below positions the first character at row 4, column 2 within the display area and displays a message beginning at that position.

```
xcall w_disp(wndw_1, WD_POS, 4, 2, "Welcome to Synergy Language!")
```

WD_ACCEPT, *field*

WD_ACCEPT receives a character from the terminal without an input terminator and stores it in an alpha or numeric field.

This function is similar to the Synergy Language ACCEPT statement. The flags set with the FLAGS subroutine have the same effect on the WD_ACCEPT function as they do on the ACCEPT statement. However, the windowing system sets runtime option flags #4 and #10 for input, and it does not reset them after the input has been completed.

Another difference is that when using WD_ACCEPT, you can't trap errors with an end-of-file label or an I/O error list. To trap errors, use an ONERROR statement before the call to W_DISP. Like WD_GETS, WD_READS, and WD_BELL, this function automatically updates the screen.

Examples

The example below displays a list of choices and stores the user's response in a field called **select**. It also traps for an EOF error.

```
onerror ($ERR_EOF) eof
xcall w_disp(wndw_1, WD_POS, 2, 2, "1. View all records",
&          WD_POS, 4, 2, "2. Select record by name",
&          WD_POS, 6, 2, "3. Select record by company",
&          WD_POS, 8, 2, "Choice?",
&          WD_ACCEPT, select)
offerror
```

WD_ANSI

WD_ANSI causes the next argument to be treated as a string that contains embedded escape sequences. These sequences are decoded to appropriate W_UPDT. The following sequences are currently supported (where *n* is a number):

<esc>[<i>n</i> ; <i>nf</i>	Position cursor.
<esc>[<i>n</i> ; <i>nH</i>	Position cursor.
<esc>[1	Set protect attribute.
<esc>[2	Clear protect attribute.
<esc>[<i>nm</i>	Set/clear attributes (<i>n</i> is optional and defaults to 0).
<esc>[J	Clear to end of screen.
<esc>[0J	Clear to end of screen.
<esc>[1J	Clear to beginning of screen.
<esc>[2J	Clear screen.
<esc>[K	Clear to end of line.
<esc>[0K	Clear to end of line.
<esc>[1K	Clear to beginning of line
<esc>[2K	Clear line.
<esc>[<i>nA</i>	Move cursor up <i>n</i> rows (<i>n</i> is optional and defaults to 1).
<esc>[<i>nB</i>	Move cursor down <i>n</i> rows (<i>n</i> is optional and defaults to 1).
<esc>[<i>nC</i>	Cursor left <i>n</i> characters (<i>n</i> is optional and defaults to 1).
<esc>[<i>nD</i>	Move cursor right <i>n</i> characters (<i>n</i> is optional and defaults to 1).
^N	Interpret lowercase characters as graphics.
^O	Do not interpret lowercase characters as graphics.

^I	Move cursor right to next tab stop (every eight characters).
CR	Move cursor to LHS of screen.
LF	Move cursor down one line.
BS	Move cursor left one character.
BELL	Sound the terminal bell.
NULL	Character ignored.

WD_ATTR, attributes

WD_ATTR sets and alters attributes for subsequent display. See [“Attributes” on page 19-13](#) for information on standard window attributes.

Examples

The example below sets the reverse video attribute.

```
xcall w_disp(wndw_1, WD_ATTR, ATTR_SET + ATTR_RVRS,
&          "Displayed in reverse video")
```

WD_BELL

WD_BELL updates the screen just like WD_UPDT and then issues a bell character (0x07) to the terminal. The bell character is external to the window being referenced.

WIN

The computer issues a beep. It does *not* send a ^G to the screen.



Use WD_BELL for a system-independent method of ringing the terminal bell.

WD_CLEAR

WD_CLEAR clears the window to blanks with the current display attributes. After the window is cleared, the current position in the window returns to row 1, column 1.

Examples

In the example below, if an ONERROR statement traps the **eof** label, we clear the window's display area with WD_CLEAR and display the “End of File” message.

```
eof,
xcall w_disp(wndw_1, WD_CLEAR, WD_POS, 5, 2, "End of File")
```

WD_CLR, *option*

WD_CLR performs clear processing with the current display attributes. *Option* defines the type of clearing that is to occur, and all of the clear options reference the current position within the window, without modifying that position. *Option* can be one of the following options:

WDC_WND	Clear the entire window.
WDC_BOW	Clear from the beginning of the window to the current position.
WDC_EOW	Clear from the current position to the end of the window.
WDC_LIN	Clear the entire line that contains the current position.
WDC_BOL	Clear from the beginning of the line to the current position.
WDC_EOL	Clear from the current position to the end of the line.

WD_COLOR, *color*

WD_COLOR sets the current display color to *color*, which is a numeric expression that contains a palette number between 1 and 16. See “[Colors and the color palette](#)” on page 19-5 for more information about color.

WD_FIELD, *expression*

WD_FIELD displays a field of text to the window, where *expression* is the text to be displayed. If *expression* is alpha, its result is displayed, and the text has the current display attributes for the window. If *expression* is numeric, its value is converted to the corresponding ASCII character (according to the Synergy Language ASCII character set), and that character is displayed.

This function updates the current column position by the length of the field until the column position reaches the window’s boundary. Any text that exceeds the window’s boundaries is ignored.

You can display the same field for a specified number of repetitions by using the WD_REPEAT function. (See WD_REPEAT on [page 19-43](#).)

Examples

In the example below, we use three display expressions (**abc**, **xyz**, and **hello**) whose lengths are defined by the length of the strings displayed in the fields. The WD_REPEAT function sets the repeat count for the WD_FIELD’s text immediately after it, which in this case is **xyz**. When we display **hello**, it is only displayed once.

```
.define MAXWINS,    15
.define WNDCHNL,    1
.include "WND:windows.def"
record
    wndw_1          ,i4

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
```

```

xcall w_proc(WP_CREATE, wndw_1, "window1", 8, 40)
xcall w_area(wndw_1, WA_FILL, ".")
xcall w_proc(WP_PLACE, wndw_1, 11, 21)
xcall w_disp(wndw_1, WD_FIELD, "abc", WD_UPDT)
xcall w_disp(wndw_1, WD_REPEAT, 3, WD_FIELD, "xyz", WD_UPDT)
xcall w_disp(wndw_1, WD_FIELD, "hello", WD_UPDT)
xcall w_exit
stop
end

```

WD_GETS, *alpha_field*

WD_GETS obtains input from the terminal at the current window position. *Alpha_field* is the field in which to store the input, and it defines the maximum size of the input field. The window area receiving the data must be visible on the screen, or else you'll receive a window error #11, "Input field not completely visible." Like WD_ACCEPT, WD_READS, and WD_BELL, this function automatically updates the screen.

As you enter characters, the echoed characters overwrite any existing characters displayed in the field. If you delete characters during input, the original characters and display attributes are restored.

The WD_GETS function is similar to the standard Synergy Language GETS statement. The flags set with the FLAGS subroutine have the same effect on the WD_GETS function as they do on the GETS statement. However, the windowing system sets runtime option flags #4 and #10 for input, and it does not reset them after the input has been completed.

Another difference is that when using WD_GETS, you can't trap errors with an I/O error list or end-of-file label. To trap errors, use the ONERROR and OFFERROR statements.

Examples

The example below positions the display to the upper-left corner of the processing area with the WA_CORNER function, then moves the display down 2 rows and over 4 columns before requesting input. To obtain the input, this program uses WD_GETS and stores the response in an **a5** field.

```

.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4
    a5_fld              ,a5

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 11, 51)
    xcall w_proc(WP_PLACE, wndw_1, 12, 12)

```

```

xcall w_area(wndw_1, WA_CORNER, WAC_ULC)
xcall w_disp(wndw_1, WD_MOVE, 2, 4, "Input: ", WD_GETS, a5_fld)
xcall w_exit
stop
end

```

WD_HERASE, *row*, *column*, *length*

WD_HERASE erases horizontal lines. *Row*, *column*, and *length* are numeric expressions, where *row* and *column* designate the point at which WD_HERASE should begin erasing, and *length* designates the number of columns that should be erased. WD_HERASE removes the crossing vertex character from any vertical lines that are “uncrossed.” It ignores any nonlinear characters within the range of the erasure and doesn’t modify display attributes. The current window position is not affected.

WD_HLINE, *row*, *column*, *length*

WD_HLINE draws a horizontal line. *Row*, *column*, and *length* are numeric expressions, where *row* and *column* mark the starting point of the line, and *length* indicates how many columns wide the line is. Horizontal lines are drawn from left to right. The current window position is not affected. Intersected vertical line characters are changed to tees or crosses as appropriate.

Examples

In this example, we use WD_HLINE to draw two horizontal lines: the first one starts at row 3, column 3 and is 36 columns wide, and the second starts at row 6, column 3 and is also 36 columns wide. Then we draw two vertical lines with WD_VLINE: the first starts at row 3 column 16 and is 5 rows long, and the second starts at row 3 column 34 and is 4 rows long.

```

.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 8, 40,
&                WP_PLACE, wndw_1, 14, 21)
    xcall w_updt
    sleep 1
    xcall w_disp(wndw_1, WD_HLINE, 3, 3, 36, WD_HLINE, 6, 3, 36)
    sleep 1
    xcall w_disp(wndw_1, WD_VLINE, 3, 16, 5)
    sleep 1
    xcall w_disp(wndw_1, WD_VLINE, 3, 34, 4)
    sleep 1
    xcall w_proc(WP_REMOVE, wndw_1)

```

```

        xcall w_exit
        stop
    end

```

WD_MOVE, *row_change*, *column_change*

WD_MOVE changes the current window input position by adding the value of *row_change* and *column_change* to the current row and column values. *Row_change* and *column_change* are numeric expressions, where *row_change* is the number of rows the input position should move and *column_change* is the number of columns the input position should move. The resulting row and column position must be within the window. If the position is not within the window, the system automatically shifts the position to the values closest to the ones you specified. For example, if you are at row 1, column 1 in a window of 11 rows by 50 columns and you specify a row change of 5 and a column change of 50, the position changes to (6,50).

If you want to move the input position up or to the left, use a negative value for *row_change* or *column_change*, respectively.

Examples

In the following routine, we use WD_MOVE to alter the display position by moving it first to the right 23 columns, then down 6 rows and 12 columns to the left, and finally 40 columns to the left.

```

.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 7, 40)
    xcall w_area(wndw_1, WA_FILL, ".")
    xcall w_proc(WP_PLACE, wndw_1, 11, 21)
    xcall w_updt
    sleep 1
    xcall w_disp(wndw_1, "top left")
    xcall w_disp(wndw_1, WD_MOVE, 0, 23, "top right")
    xcall w_disp(wndw_1, WD_MOVE, 6, -12, "bottom right")
    xcall w_disp(wndw_1, WD_MOVE, 0, -40, "bottom left")
    xcall w_exit
    stop
end

```

WD_POS, *row*, *column*

WD_POS changes the current window input position. *Row* and *column* are numeric expressions that indicate where the window input position should be placed. The specified row and column must be within the window. If the position is not within the window, the system automatically shifts the position to the values closest to the ones you specified. For example, if you create a window of 11 rows by 50 columns and specify a position of row 5, column 60, the position is changed to (5,50).

If you specify a *row* or *column* value of 0, the current value for that argument is used.

Examples

The following example is similar to the WD_MOVE example, except that the WD_POS function requires you to specify the absolute values of the row and column positions, not the relative values.

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              , i4

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 7, 40)
    xcall w_area(wndw_1, WA_FILL, ".")
    xcall w_proc(WP_PLACE, wndw_1, 11, 21)
    xcall w_updt
    sleep 1
    xcall w_disp(wndw_1, WD_POS, 7, 1, "bottom left")
    xcall w_disp(wndw_1, WD_POS, 1, 32, "top right")
    xcall w_exit
    stop
end
```

WD_READS, *alpha_field*

WD_READS reads data from the terminal into the window at the current window position. *Alpha_field* is the field in which to store the input, and it defines the maximum size of the input field. The window area receiving the data must be visible on the screen, or else you'll receive a window error #11, "Input field not completely visible." Termination and activation characters are not echoed, but nonprinting characters echo as a period (.).

When a character is deleted during WD_READS processing, the character that was originally at the window position prior to the input is restored, along with its display attributes. (In such a situation, the paint character is meaningless, and it is ignored during deletions.)

Like WD_ACCEPT, WD_GETS, and WD_BELL, the WD_READS function automatically updates the screen.

The WD_READS function is similar to the Synergy Language READS statement. The flags set with the FLAGS subroutine have the same effect on the WD_READS function as they do on the READS statement. However, the windowing system sets runtime option flags #4 and #10 for input, and it does not reset them after the input has been completed.

Another difference is that when using WD_READS, you can't trap errors with an end-of-file label or an I/O error list. To trap errors, you must include an ONERROR statement before the call to W_DISP.

WIN

A limitation of WD_READS on Windows is that it does not terminate input when the field is full. You must press ENTER after input is complete, even if you set runtime option flag #9. WD_GETS has the same functionality as WD_READS on Windows.

Examples

This example positions the display to the upper-left corner with the WA_CORNER function, then moves the display down 2 rows and over 4 columns before requesting input. To obtain the input, this program uses WD_READS and stores the response in **a5_fld**.

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4
    a5_fld              ,a5

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 11, 51)
    xcall w_proc(WP_PLACE, wndw_1, 12, 12)
    xcall w_area(wndw_1, WA_CORNER, WAC_ULC)
    xcall w_disp(wndw_1, WD_MOVE, 2, 4, "Input: ", WD_READS, a5_fld)
    xcall w_exit
    stop
end
```

WD_REPEAT, *count*

WD_REPEAT sets the repeat count for the next WD_FIELD function. *Count* is a numeric expression that is the repeat count. The field supplied in the next WD_FIELD function is output the number of times specified in *count*.

The WD_REPEAT function works for only one WD_FIELD function; you must specify a count for each WD_FIELD function that has a repeating field.

Examples

In the following routine, we use three display expressions: **abc**, **xyz**, and **hello**. The WD_REPEAT function sets the repeat count for the display field defined immediately after it, in this case, **xyz**. When we display **hello**, it is only displayed once.

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 8, 40)
    xcall w_area(wndw_1, WA_FILL, ".")
    xcall w_proc(WP_PLACE, wndw_1, 11, 21)
    xcall w_disp(wndw_1, WD_FIELD, "abc")
    xcall w_disp(wndw_1, WD_REPEAT, 3, WD_FIELD, "xyz")
    xcall w_disp(wndw_1, WD_FIELD, "hello")
    xcall w_exit
    stop
end
```

WD_UPDT

WD_UPDT updates the physical screen just like the W_UPDT subroutine. It eliminates additional subroutine overhead by eliminating the need to call W_UPDT. Specify WD_UPDT as the last argument in your W_DISP argument list.

Examples

The following example displays the message “top right” beginning at row 1, column 32 within the window and updates the screen.

```
xcall w_disp(wndw_1, WD_POS, 1, 32, "top right", WD_UPDT)
```

WD_VERASE, *row*, *column*, *length*

WD_VERASE erases vertical lines. *Row*, *column*, and *length* are numeric expressions, where *row* and *column* are the point at which WD_VERASE should begin erasing, and *length* is the number of rows that should be erased. WD_VERASE removes the crossing vertex character from any horizontal lines that are “uncrossed.” It ignores any nonlinear characters within the range of the erasure and doesn’t modify display attributes. The current window position is not affected.

WD_VLINE, *row*, *column*, *length*

WD_VLINE draws a vertical line. *Row*, *column*, and *length* are numeric expressions, where *row* and *column* mark the starting point of the line, and *length* indicates how many rows deep the line is. Vertical lines are drawn from top to bottom. The current window position is not affected. Intersected horizontal line characters are changed to tees or crosses as appropriate.

Examples

In the example below, we use WD_HLINE to draw two horizontal lines: the first one starts at row 3, column 3, and is 36 columns wide, and the second starts at row 6, column 3, and is also 36 columns wide. Then we draw two vertical lines with WD_VLINE: the first starts at row 3, column 16 and is 5 rows long, and the second starts at row 3, column 34 and is 4 rows long.

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4

proc
    open (WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 8, 40,
&                WP_PLACE, wndw_1, 14, 21)
    xcall w_updt
    sleep 1
    xcall w_disp(wndw_1, WD_HLINE, 3, 3, 36, WD_HLINE, 6, 3, 36)
    sleep 1
    xcall w_disp(wndw_1, WD_VLINE, 3, 16, 5)
    sleep 1
    xcall w_disp(wndw_1, WD_VLINE, 3, 34, 4)
    sleep 1
    xcall w_exit
    stop
end
```

WD_WAIT, *wait_time*

WD_WAIT waits the specified number of seconds for input before generating a “Terminal input operation timeout” (\$ERR_TIMEOUT) error. *Wait_time* can be one of the following identifiers or corresponding values:

Q_NOWAIT or 0	Don't wait if I/O processing can't be completed.
Q_WAIT or -1	Wait until I/O processing is completed.
<i>n</i>	Wait up to <i>n</i> seconds for I/O processing to be completed.

The **Q_** identifiers are defined by the compiler.

You should specify the WD_WAIT function *before* you specify your WD_READS, WD_GETS, or WD_ACCEPT function.

Examples

In the example below, Synergy Language waits five seconds for input before generating an error.

```
xcall w_disp(wndw_1, WD_WAIT, 5, WD_READS, rec)
```

WD_WRITES, *field*

WD_WRITES displays the next argument at the current position and wraps, if necessary. The position in the window is left at column 1 of the next row in the processing area. If the bottom of the processing area is encountered, the screen scrolls up. The *field* argument is an alpha expression.

W_EXIT – Exit the windowing system

```
xcall W_EXIT
```

Discussion

W_EXIT stops the windowing system. It deletes all defined windows and releases the window memory allocated to the system. It also clears display attributes that are set in the windowing system.

W_EXIT updates the screen before it deletes the windows so that the final states of all placed windows are displayed.

The terminal channel that was passed to W_INIT must still be open when you call W_EXIT.

After a W_EXIT, you must reinitialize the windowing system by calling W_INIT if you want to use the windowing system again.

Examples

In the following routine, we call the W_PROC subroutine and remove the window with WP_REMOVE. Then we exit the windowing system with W_EXIT, which automatically updates the screen. (Without the WP_REMOVE function, the window display remains on the screen after we exit the windowing system.)

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    window              ,i4
    choice              ,a7      , "Choice"

proc
    open(WNDCHNL, o, "tt:")
    xcall flags(7004020, 1)
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, window, choice, 11, 50,
&                WP_PLACE, window, 1, 1)
    xcall w_area(window, WA_FILL, "&")
    xcall w_updt
    sleep 1
    xcall w_proc(WP_REMOVE, window)
    xcall w_exit
    stop
end
```

W_FLDS – Manipulate field sets within a window

```
xcall W_FLDS (id, function[, ...])
```

Arguments

id

A variable that contains the window ID. The ID must be the same as the one specified in the WP_CREATE function. (See [page 19-77.](#)) (n)

function

One or more of the following functions and their arguments, which are discussed in “[Window field set functions](#)” on [page 19-49](#) and “[User data set functions](#)” on [page 19-57](#):

WF_ATTR	Alter character attributes in the field.
WF_COLOR	Set the window field’s color.
WF_CREATE	Create a set of window fields.
WF_FILL	Fill a window field with a specified character.
WF_GET	Get data from a window field.
WF_INPUT	Perform window field input.
WF_PARAM	Return the position and length of a window field.
WF_POS	Set the current window position.
WF_PUT	Put data in a window field.
WF_SET	Set the position and length of a window field.
WF_UGET	Get data from a user data set window field.
WF_UPUT	Put data into a user data set window field.
WF_USER	Create a user data set.
WF_USERID	Change the active user data set.
WF_WAIT	Set input time limit.

Discussion

W_FLDS manipulates a set of fields in a window.

W_FLDS enables you to retrieve data from, load data into, and modify the attributes of regions in a window, without affecting any other portion of the window. We call these regions *window fields*. Window fields are one-row-high regions characterized by their length and position in the window. A window field has no relation to the data fields you define in the data division of your program.

With W_FLDS you can create a single window field or window field sets. Each window field has a height of one row and a length ranging from one column to the width of the window. The length, position, data, and attributes of a window field, or an entire window field set, can be modified by W_FLDS. These changes become visible with the next screen update (if the window is placed and the window field is not occluded).

Window field set functions

Many of the W_FLDS functions require a *field_number* argument, which can be one of the following values:

<i>number</i>	A numeric expression that references the window field specified by <i>number</i> .
WFN_ALL	References all fields.
WFN_FIRST	References the first field.
WFN_LAST	References the last field.
WFN_NEXT	References the next field.
WFN_CURR	References the current field.
WFN_PREV	References the previous field.

If *number* is specified, it wraps around the last field. For example, if you create 15 fields, a *field_number* of 18 references field number 3.

For all functions except WF_FILL, WF_ATTR, and WF_COLOR, the value WFN_ALL is the same as WFN_FIRST.

If you specify the current field (WFN_CURR) and the last field was WFN_ALL, all fields are affected if the function allows it. If the function doesn't allow it, WFN_CURR is the same as WFN_FIRST.

If you use WFN_NEXT and the last field number was WFN_ALL, WFN_NEXT is the same as WFN_FIRST.

If you specify WFN_PREV and the last field specified was WFN_ALL, WFN_PREV is the same as WFN_LAST.

WF_ATTR, *field_number*, *attributes*

WF_ATTR alters the attributes of the characters in the window starting at the row and column of the specified window field for the length of the field. Possible values for the *field_number* argument are listed above. If *field_number* is WFN_ALL, the attributes of all fields are altered. See [“Attributes” on page 19-13](#) for a description of standard window attributes.

WF_COLOR, *field_number*, *color*

WF_COLOR alters the window field's color starting at the row and column of the specified window field for the length of the field. See [“Window field set functions” on page 19-49](#) for possible values for the *field_number* argument. If *field_number* is WFN_ALL, the color affects all fields. *Color* is a decimal variable that contains a palette number between 1 and 16. See [“Colors and the color palette” on page 19-5](#) for more information about color.

WF_CREATE, *num_fields*

WF_CREATE sets a number of window fields and a default length and width, where *num_fields* is the number of fields to create. The first window field's parameter is set to the first row of the processing area; the second window field's parameter is set to the second row of the processing area; and so forth. If the number of window fields is greater than the number of rows, the extra fields are set to the last row. (You can change the parameters of the individual window fields with the WF_SET function.)

The WF_CREATE function replaces any existing window field set, so that the number of created fields is not cumulative. For example, if you create a set of three window fields with one WF_CREATE function and then you create a set of five window fields with a second WF_CREATE function, you get a total of five window fields, not eight.

Examples

In the following example we create three fields with WF_CREATE and then fill them with the letter “a.” Notice that the field occupies the width of the total area; for shorter fields, you must set the field length with WF_SET.

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 11, 50)
    xcall w_area(wndw_1, WA_FILL, ".")
    xcall w_proc(WP_PLACE, wndw_1, 12, 15)
    xcall w_updt
    sleep 1
    xcall w_flds(wndw_1, WF_CREATE, 3,
&                WF_FILL, WFN_ALL, "a")
    xcall w_exit
    stop
end
```


WF_FILL, *field_number*, *character*

WF_FILL outputs the first character in *character* to the window starting at the row and column of the specified window field for the length of the field. The output characters have the default display attributes. See “[Window field set functions](#)” on page 19-49 for possible values for the *field_number* argument. If *field_number* is WFN_ALL, all fields are filled with the specified character.

Examples

In this subroutine, we first create four fields and fill them all with the letter “a.” Next, we create five fields. (Because the second WF_CREATE function overrides any previous WF_CREATE function, we have a total of five fields, not nine fields.) We fill all the fields with the letter “b” before creating six fields and filling them with the letter “c.” Since the processing area is too small to hold six fields, fields 5 and 6 are both set to the bottom row.

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 11, 50)
    xcall w_area(wndw_1, WA_SET, 5, 11, 5, 30, WA_FILL, ".")
    xcall w_proc(WP_PLACE, wndw_1, 12, 15)
    call pause
    xcall w_flds(wndw_1, WF_CREATE, 4, WF_FILL, WFN_ALL, "a")
    call pause
    xcall w_flds(wndw_1, WF_CREATE, 5, WF_FILL, WFN_ALL, "b")
    call pause
    xcall w_flds(wndw_1, WF_CREATE, 6, WF_FILL, WFN_ALL, "c")
    call pause
    xcall w_exit
    stop

pause,
    xcall w_updt
    sleep 1
    return
end
```

WF_GET, *field_number*, *alpha_field*

WF_GET transfers data from a window, starting at the row and column of the specified window field for the length of the field. This function does not obtain input from the terminal. WF_GET and WF_PUT only transfer the contents of window fields. See WF_PUT on page 19-55. For a list of possible values for the *field_number* argument, see “[Window field set functions](#)” on page 19-49.

Examples

The following routine creates four fields and then accepts input from the user. It uses WF_GET to transfer the window data, starting at the row and column of the window field for the length of the field, and store it in **beta**. The field **Beta** contains whatever the user types in, along with dots, because the field in this example is as wide as the processing area, which has been filled with dots.

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1             ,i4
    alpha              ,a20
    beta               ,a20

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 11, 20)
    xcall w_area(wndw_1, WA_FILL, ".")
    xcall w_proc(WP_PLACE, wndw_1, 12, 15)
    xcall w_flds(wndw_1, WF_CREATE, 4)
    xcall w_disp(wndw_1, WD_POS, 1, 1, WD_READS, alpha)
    xcall w_flds(wndw_1, WF_GET, WFN_NEXT, beta)
    xcall w_exit
    stop
end
```

WF_INPUT, *field_number*, *mode*, *length*, *curr_pos*, *min_pos*, *status*

WF_INPUT performs terminal input using a window field, where *field_number* is the window field to use, *mode* defines the properties of the input operation, *length* is the length of the input data, *curr_pos* is the current position within the field, *min_pos* is the minimum position that is allowed within the field, and *status* is the resulting status of the operation. See [“Window field set functions” on page 19-49](#) for possible values for the *field_number* argument. WF_INPUT performs an automatic W_UPDT.

Mode can be one or more of the following options, with multiple modes combined using .bor. or a vertical bar (!):

WFI_INSRT	Perform input in insert mode.
WFI_NODEL	Terminate when the delete character (rubout) is entered.
WFI_PRELOAD	Use entering <i>status</i> value as the first input character.
WFI_TFULL	Terminate when the input field is full.
WFI_TOVRFL	Terminate when the input field overflows.
WFI_TOCCCL	Terminate when the cursor is occluded.

WFI_TUNDFL Terminate when input field underflows.

WIN

WFI_GUI Return menu ID in extra argument.

Length specifies the number of characters that are already in the window field on entry, and is set to the number of characters within the field on exit. The valid values for *length* are between zero and the size of the window field.

Curr_pos specifies the current cursor position on entry, and is set to the final cursor position on exit. The valid values for *curr_pos* are between one and the size of the field plus one. *Curr_pos* cannot be greater than one more than *length* on entry.

Min_pos is the minimum cursor position that is to be allowed within the window field. The valid values are the same as for *curr_pos*. *Min_pos* cannot be greater than the entering value of *curr_pos*.

If you set WFI_PRELOAD, the value of *status* on entry is automatically processed as the ASCII value of the first character to be input; on exit, the value is set to one of the following options:

0	A terminating character was entered.
WFI_SEID	GUI menu selection ID was returned.
WFI_SFULL	<i>Mode</i> included WFI_TFULL and the input field is exactly full.
WFI_SOVRFL	<i>Mode</i> included WFI_TOVRFL; the input field is full; and an additional, nonterminating character was typed.
WFI_SUNDFL	<i>Mode</i> included WFI_TUNDFL, <i>curr_pos</i> is equal to <i>min_pos</i> , and a delete character was typed.
WFI_SNOTPL	The window containing the input field is not placed, and no operation has occurred.
WFI_SOCCL	The cursor or field is, or has become, occluded.
WFI_SINVIS	The window's display area does not include the input field, and no operation has occurred.
WFI_STMOUT	Operation was terminated on input time-out.

If *mode* includes WFI_NODEL, all activation characters are considered terminating characters; however, if *mode* does not include WFI_NODEL, BACKSPACE and RUBOUT (ASCII characters 8 and 127) are considered to be delete characters.

If *mode* does not contain WFI_TOCCCL and the window field is not completely within the window's display area or occluded in any way by any other placed window, no input occurs and WFI_SOCCL *status* is set. However, if *mode* contains WFI_TOCCCL, only the current cursor position must be visible within the window field, and a WFI_SOCCL termination occurs only when the cursor is no longer visible (either initially or during input processing).

If *length* is nonzero on entry, the windowing system assumes that *length* characters are currently displayed within the window field.

If there is room within the field and a nonterminating character is typed, the following occurs if WFI_INSRT is set:

- ▶ If *curr_pos* is less than or equal to *length*, the text is redisplayed within the window field from position *curr_pos* through the end of the field minus one, one position to the right.
- ▶ The typed character is displayed at *curr_pos* within the field, with the window's current display attributes.
- ▶ *Curr_pos* and *length* are incremented.

However, if WFI_INSRT is not set, the following occurs:

- ▶ The typed character is displayed at *curr_pos* within the field, with the window's current display attributes.
- ▶ *Length* is incremented if it is less than *curr_pos*.
- ▶ *Curr_pos* is incremented.

If there is not room within the field, *mode* does not contain WFI_TFULL or WFI_TOVRFL, and a nonterminating character is typed, the terminal bell rings and the character is ignored.

If *curr_pos* is greater than *min_pos*, *mode* does not include WFI_NODEL, and a delete character is typed, the following occurs if WFI_INSRT is set:

- ▶ If *curr_pos* is less than or equal to *length*, the text is redisplayed within the window field from positions *curr_pos* through the end of the field, one position to the left.
- ▶ If *length* is greater than its entering value, the "recovered" character is redisplayed at the field's end; otherwise, a blank is displayed with the window's current attributes.
- ▶ *Curr_pos* and *length* are decremented.

However, if WFI_INSRT is not set, the following occurs:

- ▶ A blank is displayed at *curr_pos* within the field, with the window's current display attributes.
- ▶ *Length* is decremented if it is less than *curr_pos*.
- ▶ *Curr_pos* is decremented.

If *curr_pos* is equal to *min_pos*, *mode* does not include WFI_NODEL or WFI_TUNDFL, and a delete character is typed, the terminal bell rings and the character is ignored.

If the returned status is zero or WFI_SOVRFL, %RDTRM returns the ASCII value of the character that caused the termination. If *mode* contained WFI_TOCCCL and the returned status is WFI_SOCCL, %RDTRM is either zero, if the cursor was not initially visible, or the character that caused the cursor to be invisible during input processing.

Setting flag #5 in the FLAGS subroutine causes no echoing to occur during input. On UNIX and OpenVMS, the characters are stored in the window's character array. If the area of the screen containing the input is caused to be redrawn, the nonechoed characters are displayed. Following the WF_INPUT, we suggest you perform a WF_GET to retrieve the input followed by a WF_PUT of new characters.

WF_PARAM, *field_number*, *row*, *column*, *length*

WF_PARAM returns the row, column, and length of the specified field. See [“Window field set functions” on page 19-49](#) for possible values for the *field_number* argument.

Examples

This subroutine positions to a field in a window and accepts input for the length of the field. It assumes the window system was initialized in an earlier routine. The subroutine uses WF_PARAM to obtain the position and length of the field.

```
subroutine fld_input
    wndid          ,n
    fld_num        ,n
    data           ,a
    .include "WND:windows.def"
record
    frow           ,i4
    fcol           ,i4
    flen           ,i4

proc
    xcall w_flds(wndid, WF_PARAM, fld_num, frow, fcol, flen)
    xcall w_disp(wndid, WD_POS, frow, fcol, WD_READS, data(1:flen))
    return
endsubroutine
```

WF_POS, *field_number*

WF_POS sets the current window position to the row and column of the specified field. See [“Window field set functions” on page 19-49](#) for possible values for the *field_number* argument.

WF_PUT, *field_number*, *alpha_field*

WF_PUT transfers data from the alpha field to the window, starting at the row and column of the specified window field for the length of the field. The output data has the current default display attributes. The data won't be displayed until the window is updated. If the alpha field is longer than the window field, only the characters that fit are loaded into the field. See [“Window field set functions” on page 19-49](#) for possible values for the *field_number* argument.

Examples

The following example uses the WF_PUT function to display selections. Notice that if you haven't used a field yet, WFN_NEXT is the same as WFN_FIRST.

```

.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 11, 50)
    xcall w_area(wndw_1, WA_FILL, ".")
    xcall w_proc(WP_PLACE, wndw_1, 12, 15)
    xcall w_flds(wndw_1, WF_CREATE, 4,
&                WF_PUT, WFN_NEXT, "Selection one",
&                WF_PUT, WFN_NEXT, "Selection two",
&                WF_PUT, WFN_NEXT, "Selection three",
&                WF_PUT, WFN_NEXT, "Last selection")
    call pause
    xcall w_exit
    stop

pause,
    xcall w_updt
    sleep 1
    return
end

```

WF_SET, *field_number*, *row*, *column*, *length*

WF_SET redefines the row, column, and length of the specified window field. Possible values for the *field_number* argument are listed in [“Window field set functions” on page 19-49](#). *Row* and *column* specify the position where the field begins within the window. *Length* indicates the length of the field. If *length* is an alpha string, the field is as long as that string, and it contains the alpha string. If *length* is numeric, it indicates the number of characters in the field.

If *length* is a numeric expression, the length of the window field is set equal to *length*. If *length* is an alpha expression, the length is set to the length of the expression and a WF_PUT of the text is executed.

If the field is too long for the window, it is truncated so that it fits.

Examples

In the example below, we set four window fields, using the length of the alpha field we want to display to define the length of the field. Notice that at the first WF_SET function, we use WFN_NEXT for the field number when we intend to set the first field. When you’ve created fields that aren’t yet displayed, WFN_NEXT is the same as WFN_FIRST. When you’re on the last field, WFN_NEXT is also the same as WFN_FIRST.

```

.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 11, 50)
    xcall w_area(wndw_1, WA_FILL, ".")
    xcall w_proc(WP_PLACE, wndw_1, 12, 15)
    xcall w_flds(wndw_1, WF_CREATE, 4,
&                WF_SET, WFN_NEXT, 6, 29, "Selection one",
&                WF_SET, WFN_NEXT, 4, 15, "Selection two",
&                WF_SET, WFN_NEXT, 8, 10, "Selection three",
&                WF_SET, WFN_NEXT, 5, 12, "Last selection",
&                WF_ATTR, WFN_NEXT, ATTR_SET + ATTR_RVRS)
    call pause
    xcall w_exit
    stop

pause,
    xcall w_updt
    sleep 1
    return
end

```

User data set functions

A user data set is information that you can associate with a window for your own usage. You completely control and maintain the parameters and contents of this information. Conceptually, a user data set is a one-dimensional alpha array. You use the `WF_USER` function to create the array, `WF_UGET` to read elements to the array, and `WF_UPUT` to write elements from the array.

WF_UGET, *element_num*, *variable*

`WF_UGET` reads an element of the user data set array, where *element_num* is the element of the user data set array to read and *variable* is a Synergy Language variable that receives the data. The data is moved to *variable* according to the rules for moving alpha data. (See [“Assignment Statements” on page 3-26](#) for more information about moving alpha data.)

WF_UPUT, *element_num*, *data*

`WF_UPUT` writes an element of the user data set array, where *element_num* is the element of the user data set array to write and *data* is the data to be placed in the field entry. *Data* is moved according to the rules for moving alpha data. (See [“Assignment Statements” on page 3-26](#) for more information about moving alpha data.)

WF_USER, *dimension, size***WF_USER**, *set_name, dimension, size*

WF_USER creates an alpha array that is a user data set, in one of two forms. In both forms, *dimension* is the number of elements in array, *size* is the size of each element, and the user data set is initialized to blanks. If the first argument of WF_USER is alphanumeric, nonnull, and nonblank, the windowing system assumes the three-argument form of this function. In all other cases, the two-argument form is assumed.

The two-argument form of WF_USER creates an unnamed user data set for the window, which is considered the default user data set. If a default data set already exists for the window, the existing data set is first removed.

The three-argument form of WF_USER creates a new user data set with the name *set_name*. If a data set already exists with that name, it is first removed. However, you can have multiple user data sets with different names. After the user data set is created, it automatically becomes the “active” data set for the window. (The active user data set is referenced by WF_UGET and WF_UPUT.)

In either form of WF_USER, if *dimension* or *size* is less than or equal to zero, the existing user data set (that is, either the default set or the set referenced by *set_name*) is deleted.

WF_USERID, *set_name*

WF_USERID changes the active user data set for a window, where *set_name* is the alphanumeric name of the user data set you want to activate. (The active user data set is referenced by WF_UGET and WF_UPUT.) If *set_name* is nonnull or nonblank, it must contain the name of an existing user data set. If *set_name* is null or blank, the existing default (unnamed) user data set is activated.



In the %W_INFO intrinsic function, you can use the WIF_USERID function to test for user data sets and/or make them active. Also, the WIF_UFLDS and WIF_UFLDSIZ functions of %W_INFO references the active data set of a window. (See [%W_INFO](#) on page 19-68 for more information.)

WF_WAIT, *wait_time*

WF_WAIT specifies a time-out value for a subsequent WF_INPUT function in the same call to W_FLDS. *Wait_time* can be one of the following identifiers or corresponding values:

Q_NOWAIT or 0	Don't wait if I/O processing can't be completed.
Q_WAIT or -1	Wait until I/O processing is completed.
<i>n</i>	Wait up to <i>n</i> seconds for I/O processing to be completed.

The **Q_** identifiers are defined by the compiler.

W_INFO – Get information about the windowing system

```
xcall W_INFO (function[, ...])
```

Arguments

function

One of the following functions and their arguments, which are discussed in the “Functions” section below:

WI_ACTIVE	Return the IDs of the active windows.
WI_AREAS	Return area information for a window.
WI_ATTRS	Return a window’s attributes.
WI_BCHR	Return the border-drawing character for a window.
WI_NAME	Retrieve the name of a window.
WI_PALET	Return the foreground/background and character color codes for the current palette entries.
WI_PLACED	Return the number of placed windows and their window IDs.
WI_SYSTEM	Return general information about the windowing system.
WI_TITLE	Return title information for a window.
WI_USER	Return the parameters of a user data set.
WI_VPLACE	Return virtual placement list.
WI_WINDOW	Return general information about a window.
WI_XFR	Transfer the current processing area’s attributes and data to or from a variable.

Discussion

W_INFO enables you to obtain various types of information about the windowing system. In the “Functions” section, we describe the type and size of the fields you must create to receive the information, and we supply a typical record definition for the return value arguments.

The return value arguments must be numeric, which means you can’t access the array by naming the record as the argument. (Records are alpha.) Instead, supply the name of the first array element as the argument in the W_INFO function. (The function descriptions include examples of data areas in which to store returned information.)



You can call the W_INFO subroutine as the intrinsic function %W_INFO. See [%W_INFO](#) on [page 19-68](#) for a description of this function.

Functions

WI_ACTIVE, *return_values*

WI_ACTIVE returns the IDs of the active windows, where *return_values* is the first element of a numeric array to be loaded with the number of active windows and the window IDs in numerical order. The size of each array element must be large enough to contain the maximum window ID, and the number of array elements should be one greater than the maximum number of windows specified in the W_INIT statement.

Examples

The example below indicates the entire array in the WI_ACTIVE function by naming the first element of the array (in this case, **numwins**).

```
.include "WND:windows.def"
.define WNDCHNL,      1
.define MAXWINS,     10
record
    numwins            ,i4
    ids                ,MAXWINS i4

proc
    xcall w_init(0, WNDCHNL, MAXWINS)
    .
    .
    .
    xcall w_info(WI_ACTIVE, numwins)
```

WI_AREAS, *id*, *return_values*

WI_AREAS returns area information for the window specified by *id*. *Return_values* is an **8d3** array loaded with the following information: window row, window column, number of rows, and number of columns in the window's current display area, followed by the window row, window column, number of rows, and number of columns in the current processing area.

Examples

```
record
    retvals            ,8d3
    disp_orig_row      ,d3 @alpha
    disp_orig_col      ,d3 @alpha + 3
    disp_num_row       ,d3 @alpha + 6
    disp_num_col       ,d3 @alpha + 9
    area_orig_row      ,d3 @alpha + 12
    area_orig_col      ,d3 @alpha + 15
    area_num_row       ,d3 @alpha + 18
    area_num_col       ,d3 @alpha + 21
```

```

.
.
.
xcall w_info(WI_AREAS, wndid, retvals)

```

WI_ATTRS, *id*, *return_values*

WI_ATTRS returns the specified window's attributes. *Return_values* is a **6d4** array loaded with the following information: the window attribute code, the window palette number, the border attribute code, the border palette number, the title attribute code, and the title palette number. An attribute code is a four-digit number, with each digit representing an attribute. If you want to save a window and restore its attributes later, you can use ATTR_LOAD + **wnd_attr** (the variable into which you have stored the attribute code).

Examples

```

record
    wndid          ,i4
    retvals        ,6d4
    wnd_attr       ,d4 @retvals
    wnd_pal        ,d4 @retvals + 4
    brd_attr       ,d4 @retvals + 8
    brd_pal        ,d4 @retvals + 12
    ttl_attr       ,d4 @retvals + 16
    ttl_pal        ,d4 @retvals + 20
.
.
.
xcall w_info(WI_ATTRS, wndid, retvals)

```

WI_BCHR, *id*, *return_values*

WI_BCHR returns the border drawing characters for the specified window. *Return_values* is a **6d3** array loaded with the following information: the display codes for the horizontal line, vertical line, upper-left corner, upper-right corner, lower-left corner, and lower-right corner.

Examples

```

record
    wndid          ,i4
    border         ,6d3
    hline_chr      ,d3 @border
    vline_chr      ,d3 @border + 3
    ulc_chr        ,d3 @border + 6
    urc_chr        ,d3 @border + 9
    llc_chr        ,d3 @border + 12
    lrc_chr        ,d3 @border + 15

```

```

.
.
.
xcall w_info(WI_BCHR, wndid, retvals)

```

WI_NAME, *id*, *name*

WI_NAME retrieves the *name* of a window specified by *id*. You can also get the name of a window using the WI_WINDOW subfunction; however, this subfunction also returns 11 other pieces of information in a numeric array. WI_NAME simplifies the process of retrieving just the name.

Examples

```
xcall w_info(WI_NAME, wndid, name)
```

WI_PALET, *return_values*

WI_PALET returns the background and character color codes for the current palette entries. *Return_values* is a **32d3** array loaded with the system-dependent background and character colors for each of the 16 palette entries. The first 16 elements are loaded with the background color codes for the 16 palette entries, and the last 16 elements are loaded with the character color codes. See [“Colors and the color palette” on page 19-5](#) for more information about color.

Examples

```

record
    retvals    32d3
    background ,16d3 @paletcodes
    characters ,16d3 @paletcodes + 48
.
.
.
xcall w_info(WI_PALET, retvals)

```

WI_PLACED, *return_values*

WI_PLACED returns the number of placed windows and their window IDs, in placement order. *Return_values* is a numeric array loaded first with the number of placed windows and then with the window IDs in placement order. The size of each array element must be large enough to contain the maximum window ID, and the number of array elements should be one greater than the maximum number of windows specified in the W_INIT statement.

Examples

The example below indicates the entire array in the WI_PLACED function by naming the first element of the array (in this case, **numwins**).

```

.include "WND:windows.def"
.define WNDCHNL,      1
.define MAXWINS,      10

```

```

record
    numwins          ,i4
    ids              ,MAXWINS i4

proc
    xcall w_init(0, WNDCHNL, MAXWINS)
    .
    .
    .
    xcall w_info(WI_PLACED, numwins)

```

WI_SYSTEM, *arg_1*, *arg_2*, *return_values*

WI_SYSTEM returns general information about the windowing system. *Arg_1* is a **d5** field that was at one time loaded with the size, in bytes, of the windowing system's memory pool. *Arg_2* is also a **d5** field that was at one time loaded with the number of free bytes in the memory pool. Because the windowing subroutines no longer require a window memory pool, however, both *arg_1* and *arg_2* now return the value of the W_INIT subroutine's first argument. *Return_values* is an **8d3** array loaded with the following information:

- ▶ Maximum number of windows
- ▶ Maximum number of screen rows
- ▶ Maximum number of screen columns
- ▶ Current number of screen rows
- ▶ Current number of screen columns
- ▶ Current screen row
- ▶ Current screen column
- ▶ Terminal channel

Examples

```

record
    arg_1            ,d5
    arg_2            ,d5
    retvals          ,8d3
    maxwins          ,d3 @retvals
    maxrows          ,d3 @retvals + 3
    maxcols          ,d3 @retvals + 6
    curr_rows        ,d3 @retvals + 9
    curr_cols        ,d3 @retvals + 12
    updt_row         ,d3 @retvals + 15
    updt_col         ,d3 @retvals + 18
    ttchn            ,d3 @retvals + 21
    .
    .
    .
    xcall w_info(WI_SYSTEM, arg_1, arg_2, retvals)

```

WI_TITLE, *id*, *title*, *return_values*

WI_TITLE returns title information for the window specified by *id*. *Return_values* is a **2d3** array loaded with the following information: a code signifying which border contains the title and a code signifying the position of the title within that border. If you want to save a window and restore the title information later, you can use the variables in which the codes are stored to restore the title with the same border and position.

Examples

```
record
    wndid          ,i4
    title          ,a50
    retvals        ,2d3
    border         ,d3 @retvals
    pos           ,d3 @retvals + 3
    .
    .
    .
    xcall w_info(WI_TITLE, wndid, title, retvals)
```

WI_USER, *id*, *array*

WI_USER returns the parameters of the current user data set, where *id* is the ID of the window to reference and *array* is the first element of a decimal array that is to receive the parameters. The first element of *array* is loaded with the number of entries, the second element is loaded with the length of each entry, and the third element receives the current field of the user data set. If no user data set is defined, all values are zero. (To establish the current user data set, use the WF_USERID function of W_FLDS, or the WIF_USERID function of %W_INFO.)

WI_VPLACE, *return_values*

WI_VPLACE returns the number of virtually placed windows and their window IDs. If a window is virtually placed, it means that it is physically placed if its parent window is also physically placed, and the converse. Windows without parents do not distinguish between virtual and physical placement; the two are identical.

While WI_PLACED returns the IDs of all windows (whether children of another window or not), that are physically visible on the screen, WI_VPLACE returns the IDs of all windows that are logically placed. This includes the list returned by WI_PLACED, as well as child windows that have been placed but have a parent (or ancestor) that is not placed.

WI_WINDOW, id, name, return_values

WI_WINDOW returns general information for the specified window. *Name* is returned with the window's name, which can be up to 15 alpha characters long. *Return_values* is an **11d4** array loaded with the following information: the window's number of rows, number of columns, placement row, placement column, status flags, overlay window ID, title length, number of fields in its field set, current field number in the field set, current row, and current column.

The status flags are represented by a **d4** value, with 1 signifying true and 0 signifying false. The rightmost digit is 1 if the window is currently placed, the next digit is 1 if its border is currently on, and the next digit is 1 if a title is being displayed.

WIN

The leftmost digit is *always* 0.

UNIX, VMS

The leftmost digit is 1 if the window is currently occluded by another window.

The overlay window ID is zero if the window is not an overlay window; otherwise it is the lowest numerical window ID that this window overlays.

Examples

```
record
    wndid          ,i4
    name           ,a15
    retvals        ,11d4
    numrows        ,d4 @retvals
    numcols        ,d4 @retvals + 4
    placerow       ,d4 @retvals + 8
    placecol       ,d4 @retvals + 12
    flags          ,d4 @retvals + 16
    overid         ,d4 @retvals + 20
    titlelgth      ,d4 @retvals + 24
    numflds        ,d4 @retvals + 28
    curr_fld       ,d4 @retvals + 32
    curr_row       ,d4 @retvals + 36
    curr_col       ,d4 @retvals + 40
    .
    .
    .
    xcall w_info(WI_WINDOW, wndid, name, retvals)
```

WI_XFR, *id*, *option*, *info*

WI_XFR transfers the current processing area's attributes or data to or from a variable, where *id* is the ID of the window whose processing area should be transferred. *Option* is one of the following:

WIX_AGET	Transfer the area's attributes to <i>info</i> .
WIX_DGET	Transfer the area's data to <i>info</i> .
WIX_APUT	Transfer the attributes from <i>info</i> to the current area.
WIX_DPUT	Transfer the data from <i>info</i> to the current area.
WIX_SAGET	Transfer the screen attributes to <i>info</i> .
WIX_SDGET	Transfer the screen data to <i>info</i> .

For WIX_APUT and WIX_DPUT, the destination processing area must have the same dimensions as the source processing area. For WIX_SAGET and WIX_SDGET, the *id* argument is unused, but it is still required.

WIN

WIX_SDGET does not work on Windows. The data for each Synergy window is kept only within that window, and no global screen map is maintained.

Info is the alpha variable to which window information is saved and from which windows are restored. *Info* must be large enough to hold the contents of the current processing area; for each option, the length of the variable should equal the number of rows multiplied by the number of columns.

With the WI_XFR function, a program could save a processing area's character attributes and data to disk. Later, a different program could read the information into a processing area with the same dimensions as the original processing area.



If you are using UI Toolkit, always set the processing area (using XCALL W_AREA) before you invoke any WI_XFR options, because a Toolkit routine may modify the processing area.

Examples

In the following example, we use WIX_AGET and WIX_DGET to save the contents of a processing area before filling the processing area with dots. Then we use WIX_APUT and WIX_DPUT to restore the original data and attributes to the processing area.

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4
    attr                 ,a238
    data                 ,a238

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 11, 50)
    xcall w_area(wndw_1, WA_FILL, "1234567890")
    xcall w_proc(WP_PLACE, wndw_1, 12, 15)
    xcall w_area(wndw_1, WA_SET, 3, 9, 7, 34, WA_BOX,
&                WA_CHANGE, 1, 1, -2, -2, WA_FILL, "x")
    call pause
    xcall w_area(wndw_1, WA_SET, 3, 9, 7, 34)
    xcall w_info(WI_XFR, wndw_1, WIX_AGET, attr)
    xcall w_info(WI_XFR, wndw_1, WIX_DGET, data)
    xcall w_area(wndw_1, WA_COPY, WAC_WTOP, WA_FILL, ".")
    call pause
    xcall w_area(wndw_1, WA_SET, 3, 9, 7, 34)
    xcall w_info(WI_XFR, wndw_1, WIX_APUT, attr)
    xcall w_info(WI_XFR, wndw_1, WIX_DPUT, data)
    call pause
    xcall w_exit
    stop

pause,
    xcall w_updt
    sleep 1
    return
end
```

%W_INFO – Get information about the windowing system

`%W_INFO (wif_code1)`

`%W_INFO (wif_code2, window_id)`

`%W_INFO (wif_code3, window_id[, dataset_id])`

Arguments

wif_code1

Returned with a value based on one of the following codes defined in **winfo.def: (n)**

WIF_MAXWND	Maximum number of windows
WIF_SROWS	Current number of screen rows
WIF_SCOLS	Current number of screen columns
WIF_TTCHNL	Terminal channel number
WIF_TMAR	Top margin value
WIF_BMAR	Bottom margin value
WIF_LMAR	Left margin value
WIF_RMAR	Right margin value
WIF_CAPTURE	ID of the window that has capture of the cursor

wif_code2

Returned with a value based on one of the following codes defined in **winfo.def: (n)**

WIF_PLCFLG	Window is placed flag
WIF_BDRFLG	Border is enabled flag
WIF_TTLFLG	Window has a title flag
WIF_OCLFLG	Window is occluded flag
WIF_NOMFLG	Ignore margins on placement flag
WIF_ROWS	Number of window rows
WIF_COLS	Number of window columns
WIF_PLCROW	Requested placement row
WIF_PLCCOL	Requested placement column
WIF_ACTPROW	Actual placement row

WIF_ACTPCOL	Actual placement column
WIF_ROWPOS	Current position row
WIF_COLPOS	Current position column
WIF_PROW	Processing area upper-left row
WIF_PCOL	Processing area upper-left column
WIF_PROWSIZ	Processing area number of rows
WIF_PCOLSIZ	Processing area number of columns
WIF_DROW	Display area upper-left row
WIF_DCOL	Display area upper-left column
WIF_DROWSIZ	Display area number of rows
WIF_DCOLSIZ	Display area number of columns
WIF_FLDS	Number of window fields
WIF_CURFLD	Current window field
WIF_UFLDS	Number of user data set fields
WIF_UFLDSIZ	Size of each user data set field
WIF_HWND	System-specific window handle
WIF_UNDER	ID of window under the specified window
WIF_DRAGBAR	Window has a dragbar flag
WIF_CHILD	ID of the first child window of <i>window_id</i>
WIF_PARENT	ID of the parent of <i>window_id</i>
WIF_NEXT	ID of the next sibling of <i>window_id</i>
WIF_CURSOR	ID of the cursor associated with <i>window_id</i>
WIF_POSX	Horizontal position of <i>window_id</i> in pixels
WIF_POSY	Vertical position of <i>window_id</i> in pixels
WIF_NORMALX	Normal (restored) horizontal position of <i>window_id</i> in pixels
WIF_NORMALY	Normal (restored) vertical position of <i>window_id</i> in pixels
WIF_STATE	State of <i>window_id</i>

wif_code3

Returned with a value based on the following codes defined in **winfo.def**: (n)

WIF_USERID User data set ID exists flag

WIF_USRMEM Memory handle to a user data set in a window

window_id

The ID of the window for which you want to get information. (n)

dataset_id

(optional) The ID of the user data set for which you want to get information. (a)

Discussion

The %W_INFO intrinsic function returns various types of information about the windowing system. This function can be called in one of three forms.

If a window ID is required, *window_id* must be valid, zero, or not passed. If zero or not passed, the last window ID that is referenced is used.

Status flags are returned as true or false.



%W_INFO is identical in function to the W_INFO subroutine. (See [W_INFO](#) on page 19-59 for more information.) The “WIF_” codes, however, are valid only with %W_INFO, not with the W_INFO subroutine.

The WIF_CAPTURE subfunction returns the ID of the window that currently has capture of the cursor. If no window has capture, or if capture has been established on a non-Synergy window (for example, a DLL function call), WIF_CAPTURE returns WID_NONE for the window ID.

A macro is provided in **windows.def** to simplify usage:

```
.define W_HOURLASS_IS_ON
&      (%w_info(WIF_CAPTURE) .eq. WID_CONTAINER) .and.
&      (%w_info(WIF_CURSOR, WID_CONTAINER) .eq. WPCS_WAIT))
```

which can be used as follows:

```
if (W_HOURLASS_IS_ON)
  xcall w_proc(WP_HOURLASS_OFF)
```

The WIF_HWND subfunction retrieves system-specific window handles. With an increased demand for interoperability, especially in the Windows environments, comes the need to determine the system-specific window handle for a Synergy window. This handle can be passed to a DLL or C-interface routine to perform system-specific actions on the window. For instance, if you wanted to display bitmap graphics in a Synergy window, you could write a DLL routine to manage the image, but let the rest of the window management functions be handled by the Synergy runtime.

There are two special values for *window_id*:

WIFH_CONTAINER. The container window for the Synergy application

WIFH_APP The application window

Synergy applications always have these two windows. The container window comprises the border and caption of the application and can contain the menu, toolbars, application window, and status windows (information line and footer). The application window only contains the space in which Synergy windows can be placed. Thus, the application window is a child of the container window, and all Synergy windows are children of the application window. Remember that anything you change in a Synergy window might conflict with internals of the Synergy runtime; to avoid problems use good subclassing techniques.

The WIF_UNDER subfunction enables you to descend through the placement list to find the next window under a specified window. The return value is the ID of the next window that is partly or fully occluded by the specified window, or zero if no such window exists.

The WIF_DRAGBAR subfunction enables your application to determine the present state of the dragbar for a window. On Windows, the existence of the dragbar (or caption) is independent of whether or not a title is on. For compatibility of window libraries and application code, we also maintain the dragbar state on UNIX and OpenVMS.

The WIF_CHILD subfunction is used to retrieve the first child of a window. You can then use WIF_NEXT to loop through the remaining children. If no children exist for the specified window, this subfunction returns zero.

The WIF_PARENT subfunction returns the ID of the parent window for a specified window. If the window does not have a parent, zero is returned.

The WIF_NEXT subfunction returns the ID of the next window whose parent is the same window as the specified window. If no such window exists, zero is returned. The order of the siblings is the same as the order in which WP_CHILD was called to establish each relationship.

The WIF_CURSOR subfunction returns the ID of the cursor specified for a Synergy window. The returned cursor ID corresponds to the value passed to a prior call to W_PROC(WP_CURSOR, WP_SET). If no cursor has been specified for the window, WPCS_ARROW is returned.



If WPCS_HANDLE is returned, no additional information is available for retrieving the handle to the cursor.

Refer to the example for WIF_CAPTURE on [page 19-70](#).

The WIF_POSX and WIF_POSY subfunctions return the current pixel location of the top left corner of a window. If the window is a Synergy window, the position is relative to the client area of the application window (or relative to the tab, if the window belongs to a tab set).

The WIF_NORMALX and WIF_NORMALY subfunctions return the normal position of the top left corner of a window, which is the restored position for the window. These subfunctions work for a Synergy window as well as for the application container. For a Synergy window, the returned coordinates are relative to the client area of the window's parent (usually the application window). The difference between WIF_POSX/WIF_POSY and WIF_NORMALX/WIF_NORMALY is that the former return the current position of the window with no regard to its current state, which means they might return negative values for a maximized window or an unreasonably large y coordinate for a minimized window.

The WIF_STATE subfunction returns one of the following:

WPS_NORMAL	Window is normal.
WPS_MAXIMIZED	Window is maximized.
WPS_MINIMIZED	Window is minimized.
WPS_HIDDEN	Window is hidden.

Window_id is either a Synergy window ID or

WID_CONTAINER The application container window.

The WIF_USRMEM subfunction returns a Synergy Language memory handle to a user data set in a Synergy window. The handle is global and may not be freed or resized; if you attempt this, the error message: "Invalid memory handle" is generated.

If *dataset_id* is not passed, the most recently accessed user data set for this window is implied (see W_FLDS [WF_USERID] on [page 19-58](#)). After the call to WIF_USRMEM, the specified user data set becomes the current one (for subsequent calls to W_FLDS and W_INFO).

The returned handle may be used with the ^M casting operator to read or write data in the window's user data set, using a STRUCTURE.

W_INIT – Initialize the windowing system

```
xcall W_INIT(arg_1, channel, max_windows[, #rows, #cols][, init_size])
```

Arguments

arg_1

A numeric expression that at one time contained the number of bytes of memory allocated for window processing. Because the windowing subroutines no longer require a window memory pool, however, this argument is no longer used. *Arg_1* must be numeric and present, but it is ignored. **(n)**

channel

An expression that contains the channel you have opened for terminal output. **(n)**

max_windows

An expression that contains the maximum number of windows that can be active simultaneously. This argument has no upper limit. It causes four bytes to be reserved in memory for each window. For example, if the maximum number of windows is 15, 60 bytes is reserved in memory. **(n)**

#rows

(optional) An expression specifying the initial number of rows in the display screen. **(n)**

#cols

(optional) An expression specifying the initial number of columns in the display screen. **(n)**

init_size

(optional) An expression that is returned with a value of zero if the windowing system has not been previously initialized or with *arg_1* if the windowing system has already been initialized. **(n)**

Discussion

W_INIT initializes the windowing system, so you must call it before any other window processing occurs.

In a bound program, you don't need to initialize the windowing system more than once, but the same channel must be open to the terminal in each program. If *init_size* is passed and the windowing system has already been initialized, the windowing system is not reinitialized.

If *#rows* and *#cols* are not specified, the screen size equals the default for the specific window environment.

W_INIT interprets the DBL_CHRSET environment variable, if it exists, to set an alternate language character set.



If a program using windowing subroutines chains to another program using windowing subroutines that call W_INIT again, your window set-up is lost. You must set up your windowing system in each program.

Examples

In the example below, we declare WNDCHNL and MAXWINS as constants with the .DEFINE statement. The channel argument (WNDCHNL) in the W_INIT call must be the same as the channel opened to the terminal in the OPEN statement. The program tests whether the windowing system has been initialized, and if it hasn't, the program creates its windows.

```
.define MAXWINS,      5
.define WNDCHNL,      1
.include "WND:windows.def"
record
    init                ,d5

proc
    open(WNDCHNL, o, "tt:")
    xcall flags(7004020, 1)
    xcall w_init(1, WNDCHNL, MAXWINS,,init)
    if (.not.init)
        xcall create_wnds
    .
    .
    .
```


W_PROC – General window processing

`xcall W_PROC (function[, ...])`

Arguments

function

One or more of the following functions and their arguments, which are discussed in the “Functions” section below:

WP_CHILD	Define a parent-child relationship.
WP_COPYBLK	Copy blocks of text and attribute data between windows.
WP_CREATE	Create a window.
WP_CURSOR	Turn the cursor on or off and set its position.
WP_DELETE	Delete a window.
WP_FIND	Search for a window with the specified name or number.
WP_MARGIN	Manipulate screen margins.
WP_MOVE	Move a window relative to the most recent placement.
WP_OPTION	Control hardware scrolling.
WP_OVERLAY	Overlay a window.
WP_PALET	Change color characteristics.
WP_PLACE	Place a window’s display area on the screen.
WP_POSITION	Set the application window’s position on the screen.
WP_REDRAW	Set a flag so the screen is redrawn on the next screen update.
WP_REMOVE	Remove a window from the screen.
WP_RESIZE	Resize a window or the screen.
WP_SIZE	Change the size of a Synergy window.
WP_STATE	Set the state for a window.
WP_TOFRONT	Bring a window to the front of the placement order without moving it.

Discussion

W_PROC is the general window processing subroutine with the following available functions.

Functions

WP_CHILD, *child_id*, *parent_id*

WP_CHILD declares a parent-child relationship between two windows. When a window is the child of another window, its existence depends in every way upon the parent window:

- ▶ The placement of the child window is relative to the screen placement position of the parent window (as opposed to being relative to the screen origin), which means that placement coordinates may be negative.
- ▶ If a parent window is moved, all of its child windows move accordingly.
- ▶ If a parent window is removed from the screen, all of its child windows are removed from the screen.
- ▶ If a parent window is deleted, all of its child windows are deleted.
- ▶ If a removed parent is placed, the child windows are physically placed if they had been explicitly (logically) placed before. Placing a child window when the parent is removed does not affect the screen.

To terminate a parent-child relationship, specify zero as the parent ID. When a parent-child relationship is terminated, the child window is removed from the screen.

When WP_CHILD is performed, the window is removed. You can make the window reappear at the previous placement coordinates using

```
xcall u_window(D_REMOVE, child, D_MOVE, child, 0, 0)
```

or, if not logged with the Toolkit, using

```
xcall w_proc(WP_REMOVE, child, WP_MOVE, child, 0, 0)
```

WP_CPYBLK, *from_id*, *to_id*

WP_CPYBLK copies blocks of text and attribute data between windows relative to the processing areas. *From_id* is the ID of the window from which you want to copy text and attributes, and *to_id* is the ID of the window to which you want to copy this data. The text and attributes within the processing area of *from_id* are copied to the processing area of *to_id*. The data is copied left- and top-justified, with truncation and over nonattribute blanks.

When copying text and attributes between windows, the window data areas must be different. If *from_id* and *to_id* specify the same window ID, or if both IDs are for overlay windows with the same data, the windowing system ignores the copy request made by WP_CPYBLK.

WP_CREATE, *id*, *name*, *height*, *width*

WP_CREATE creates a window, identifying it with an ID and a name, and defines its size in rows and columns. After you've initialized the windowing system with **W_INIT**, the **WP_CREATE** function is the first one you need to use to make and display a window. The window is initially blank.

Id is a numeric field that is returned with the window's ID. The system automatically assigns a number to the variable you specify. This way, the system keeps track of the actual number of the window. In other functions acting upon this window, you simply supply the argument named here.

Name is an alpha expression with a maximum of 15 characters that contains the window's name. This argument is useful when you're not sure of the window ID. For example, you might have a standard help window that you want to use in many programs. When you chain to a new program, you may not know the window ID, but you can use the name in the **WP_FIND** function to search for and identify the window you need. If a null string is specified for *name*, a unique window name is automatically generated in the form "**_W_nnn**", where *nnn* is the ID of the window that is being created.

Height is a numeric expression that contains the number of rows deep the window will be. *Width* is a numeric expression that contains the number of columns wide the window will be. If this field contains a value of zero, for either *height* or *width*, the windowing system defaults to the screen height and width.

Examples

In the following example, we create a window whose ID is stored in **window** and whose name is stored in **choice**. (Notice that we initialize the field **choice** with the name "Choices," but we don't furnish the numeric ID for the window. The system automatically assigns a number ID to the field **window**.) The window we've created is 11 rows deep and 50 columns wide.

From now on, we can access this window with the ID **window**. If the ID is unknown by another routine, we can use the name in the **WP_FIND** function to determine if a "Choices" window exists.

```
.define WNDCHNL,      1
.define MAXWINS,     4
.include "WND:windows.def"
record
    window            ,i4
    choice            ,a7      , "Choices"

proc
    open(WNDCHNL, o, "tt:")
    xcall flags(7004020, 1)
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, window, choice, 11, 50)
```

WP_CURSOR, *option*

WP_CURSOR sets the cursor for a window, captures the cursor to a window, or sets the cursor's position following a call to W_UPDT or W_EXIT. When the windowing system is initialized, the cursor display is on and the position is row 1, column 1.

The WP_CURSOR position does not apply to terminal input. On terminal input, the cursor is positioned to wherever the input is requested on the screen.

Option is one of the following options:

WPC_CAPTURE	Capture the cursor to a window.
WPC_LVCSR	Leave the cursor at the current position on a W_UPDT.
WPC_OFF	The cursor is off.
WPC_ON	The cursor is displayed. (default)
WPC_POS	The cursor is at the specified row and column position on the screen. (The syntax for WPC_POS is described below.)
WPC_SET	Set the cursor for a window.

WPC_CAPTURE captures the cursor to a specified window, giving the window exclusive control of the cursor. It has the following syntax:

WPC_CAPTURE, *window_id*

Window_id is either the ID of a window or one of the following:

WID_CONTAINER	The container window for the Synergy application (including the border and caption)
WID_APP	The application window (excluding the border and caption)
WID_NONE	No window

Establishing capture to a window releases capture from any window that previously held it.

WID_NONE specifies that no window has capture of the cursor.

WPC_POS positions the cursor at the screen coordinates, not the window coordinates. It has the following syntax:

WPC_POS, *screen_row*, *screen_col*

Screen_row and *screen_col* are numeric expressions.



Using WPC_POS turns off the WPC_LVCSR option.

WPC_SET determines what the cursor looks like as it passes over the specified window. It has the following syntax:

WPC_SET, *window_id*, *cursor*

Window_id is either the ID of a Synergy window or one of the following:

WID_CONTAINER The container window for the Synergy application (including the border and caption)

WID_APP The application window (excluding the border and caption)

Cursor is one of the following cursor identifiers:

WPCS_ARROW Normal arrow cursor

WPCS_IBEAM I-beam (text caret)

WPCS_WAIT Hourglass

WPCS_CROSS Cross hair

WPCS_UPARROW Vertical arrow

WPCS_SIZENWSE Diagonal sizing arrows (\)

WPCS_SIZENESW Diagonal sizing arrows (/)

WPCS_SIZEWE Horizontal sizing arrows

WPCS_SIZENS Vertical sizing arrows

WPCS_SIZEALL Crossed sizing arrows (+)

WPCS_NO Slashed circle

WPCS_APPSTARTING

Arrow with hourglass

WPCS_HELP “What’s this?” cursor

or the special form

WPCS_HANDLE An externally loaded cursor handle



Modal window calls such as U_MESSAGE, %U_MSGBOX, and U_ABORT force the cursor to revert to a normal arrow.

The following macros are available in **windows.def**.

```
.define WP_HOURLGLASS_ON
&      WP_CURSOR, WPC_CAPTURE, WID_CONTAINER
&      WP_CURSOR, WPC_SET, WID_CONTAINER, WPCS_WAIT

.define WP_HOURLGLASS_OFF
&      WP_CURSOR, WPC_SET, WID_CONTAINER, WPCS_ARROW,
&      WP_CURSOR, WPC_CAPTURE, WID_NONE
```

These macros enable the following usage:

```
xcall w_proc(WP_HOURLGLASS_ON)
.
.
.
xcall w_proc(WP_HOURLGLASS_OFF)
```

Additionally, to determine what settings are currently in effect, use the WIF_CURSOR and WIF_CAPTURE subfunctions of %W_INFO or the W_HOURLGLASS_IS_ON macro. (See [pages 19-68](#) and [19-70](#) for more information.)

Examples

In this example, we reposition the cursor so that it appears at the end of the displayed message, instead of at the upper-left corner of the screen.

```
wait,
    xcall w_proc(WP_CREATE, window, choice, 1, 15,
&      WP_PLACE, window, 24, 32)
    xcall w_brdr(window, WB_OFF)
    xcall w_disp(window, WD_ATTR, ATTR_SET + ATTR_BLNK + ATTR_BOLD,
&      "Please wait...")

;Position cursor after the dots

    xcall w_proc(WP_CURSOR, WPC_POS, 24, 46)
    xcall w_updt
    return
```

WP_DELETE, *id*

WP_DELETE deletes the window with the specified ID, where *id* is a numeric expression. When the window is deleted, memory is released, and the window ID is no longer active.

WP_DELETE is *not* the same as WP_REMOVE. WP_REMOVE removes a window from the screen, but the window still exists, so it can be modified, placed again, and displayed. When a window is deleted, however, it is gone.

Examples

The example below deletes the window with the ID **wndw_4**.

```
xcall w_proc(WP_DELETE, wndw_4)
```

WP_FIND, *return_id*, *id_field*

WP_FIND searches for a window with the name or number that you supply in *id_field*. If the window exists, *return_id* (which is a numeric variable) is loaded with the window's ID so you can identify it in other functions. If the window doesn't exist, *return_id* is loaded with a zero.

Id_field is either an alpha or numeric expression. If you are searching for a window by name, specify the name as alpha. If you know the actual window ID, specify it as numeric.

Examples

The following code would appear at the beginning of a subroutine that was called from another program. We'll assume we don't know the ID of the menu window and use WP_FIND to determine the ID. Then we test the value of the return ID field (in this case, **ret_id**). If the value is zero, the window does not exist, and we can create a menu window. If the value is not zero, we know the window exists, and we can use the **ret_id** field as the window ID in other window functions.

```
subroutine menu
.include "WND:windows.def"
record
    win3          ,i4
    ret_id        ,i4
    menu          ,a*      , "Menu"

proc
    xcall w_proc(WP_FIND, ret_id, menu)
    if (.not.ret_id) then
        xcall w_proc(WP_CREATE, win3, menu, 11, 50)
    else
        xcall w_proc(WP_PLACE, ret_id, 1, 1)
```

WP_MARGIN, **WPM_SET**, *top*, *bottom*, *left*, *right***WP_MARGIN**, *flag*, *id*

WP_MARGIN sets the screen margins. This function has two forms. If the first argument is WPM_SET (which is defined in **wproc.def**), WP_MARGIN sets the screen margins, where *top* is the number rows for the top margin, *bottom* is the number of rows for the bottom margin, *left* is the number of columns for the left margin, and *right* is the number of columns for the right margin.

In the two-argument form of WP_MARGIN, *flag* is either WPM_ON or WPM_OFF (both defined in **wproc.def**). If you use this form, WP_MARGIN turns margin-sensitive screen placement on or off for the window specified by *id*. When margin-sensitive screen placement is turned on (WPM_ON), the specified window is placed relative to the current screen margins.



All screen margins are initially set to zero with margin-sensitive placement turned on.

If a window is placed and you change the screen margins or the margin-sensitive state of the window, the windowing system automatically shifts the placement of windows on the physical screen.

Examples

The following example changes the top screen margin to five rows, the bottom margin to one row, and the left and right margins to three columns each.

```
xcall w_proc(WP_MARGIN, WPM_SET, 5, 1, 3, 3)
```

Using this example, if you place a window at (1,1) and margin-sensitive placement is turned on, the window is physically placed at (6,4) on the screen.

WP_MOVE, *id*, *row_change*, *column_change*

WP_MOVE places the window's display area relative to its most recent row and column position. If the window hasn't been placed yet, the window's default point of origin is row 1, column 1.

Id is a numeric expression that contains the window ID.

Row_change is a numeric expression that contains the number of rows you want to move the window up or down. (A negative value moves the window up; a positive value moves the window down.)

Column_change is a numeric expression that contains the number of columns you want to move the window left or right. (A negative value moves the window to the left; a positive value moves the window to the right.)

The position of the window is modified by WP_MOVE. If the window is not yet placed, WP_MOVE places the window at a new position at the top of the placement order. (See [“Window placement and occlusion” on page 19-4](#) for more information about window placement.) If the window is already placed, placement order is maintained and the window is shifted to the new position.

WP_MOVE adjusts the window position relative to the current position. WP_PLACE, however, sets the window position regardless of the current position.

Examples

The following example shifts a window around the screen, first moving it 50 columns to the right, and then moving it 6 rows down and 25 columns to the left. Notice that this program does not use the WP_PLACE function; WP_MOVE places the window first, and the row and column change is calculated with a default window origin of row 1, column 1.

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 10, 30)
    xcall w_area(wndw_1, WA_FILL, "1234567890")
    xcall w_brdr(wndw_1, WB_OFF)
    xcall w_proc(WP_MOVE, wndw_1, 0, 50)
    call pause
    xcall w_proc(WP_MOVE, wndw_1, 6, -25)
    call pause
    xcall w_exit
    stop

pause,
    xcall w_updt
    sleep 1
    return
end
```



If several windows are placed on the screen and you want to take the window at the bottom of the pile and place it on top, it's easiest to remove the window first, and then move it with no row or column change. Since WP_MOVE actually places the window, you don't need the WP_PLACE function; simply use WP_MOVE with no change of row or column position, and the window is placed on the top of the pile, in its original position. The example below illustrates this technique:

```
xcall w_proc(WP_REMOVE, window, WP_MOVE, window, 0, 0)
```

WP_OPTION, *option*

WP_OPTION controls hardware scrolling, which uses the capabilities of the terminal to do the scrolling.

Option is one of the following options:

- WPO_JUMP** Emit VT100 escape sequences to set jump scroll.
- WPO_SMOOTH** Emit VT100 escape sequences to set smooth scroll.
- WPO_NOSCROLL** Explicitly turn off any scrolling region and hardware scrolling.
- WPO_VSCROLL** Turn on vertical scrolling if possible, depending on the setting of TERM.
- WPO_KEEPRDR** Prevent window borders from being turned off because of the size of the window (on Windows platforms only).
- WPO_HIDEPRDR** Restore the default border behavior after WPO_KEEPRDR has been set.
- WPO_COLORMAP** Set one of the following attribute color mapping options:
 - WPOC_ON** Turn on attribute color mapping.
 - WPOC_OFF** Turn off attribute color mapping.
 - WPOC_ULINE** Turn on underline attribute color mapping.

With WPO_JUMP or WPO_SMOOTH on UNIX or OpenVMS, if the terminal type is neither VT nor XTERM, no escape sequences are sent to the terminal.

WPO_VSCROLL turns on hardware scrolling, which only occurs under the conditions listed on [page 19-11](#).

When turned on, the attributes BOLD, UNDERLINE, REVERSE and their combinations are mapped to the default 16 color palettes as defined in “[Colors and the color palette](#)” on [page 19-5](#). These colors can be set up using WNDC or using the WP_PALET function of W_PROC.

The protected attribute selects the alternate mappings 9–16. This is to map the fact that most color terminals provide a different color set for protected characters.

Mappings are as follows:

Attribute combination	Palette
None	1
Bold	2
Underline	3
Bold Underline	4
Inverse	5
Inverse Bold	6
Inverse Underline	7
Inverse Underline Bold	8
Protected	The above palette mappings + 8

Blink uses the palette as defined above but also blinks the character (or uses italic on Windows). Color support with WD_COLOR is ignored when the WPO_COLORMAP mode is on. Protected attributes are ignored when in normal mode.

The color mapping is interpreted at runtime in the routine that decodes attributes. This routine is only used when screen updates occur. This does not affect individual windows, rather the screen output.

If WNDC is not set, WPO_COLORMAP is ignored. If WPO_COLORMAP is turned off, the protected attribute is ignored.

If a particular palette, which will be used by the attribute mapping, is not set, the mapping defaults to palette 1.

If WNDC is set on a monochrome terminal, the color sequences are still output. It is the terminal's job to ignore sequences that it does not support (in fact, this is the ANSI standard for escape sequences).

The user will set WNDC and can then use WPO_COLORMAP as a toggle. Any screen update (rather than individual window) from that point on interprets the attributes as per the setting. If the whole screen is to be updated after changing WPO_COLORMAP, it is suggested that the W_PROC function WP_REDRAW be used.

When WPO_COLORMAP is on, WD_ANSI recognizes the protected mode escape sequences.



When using the protected attribute you should *not* turn off WPO_COLORMAP unless all windows have been removed. The protected attribute conflicts with the color settings when WPO_COLORMAP is off.

WP_OVERLAY, *existing_window*, *id*, *name*

WP_OVERLAY overlays an existing window. A window overlay is similar to a Synergy Language record overlay: the data, window size, and data attributes of the new window are the same as the overlaid window. You can change all other window attributes (such as border, title, and placement).

Existing_window is a numeric or alpha expression that contains the ID or name of the original window.

Id is a numeric variable that contains the window ID of the new overlay window.

Name is an alpha expression that contains the name of the overlay window.

Examples

The following program creates an overlay window, **wndw_2**, which overlays **wndw_1**. When we first overlay **wndw_2**, it contains the same data as **wndw_1**. When we fill **wndw_2** with different data, the data changes in **wndw_1** as well, because overlaid windows have the same window size, data, and data attributes as the window that overlays them. Notice that we can place **wndw_2** separately from **wndw_1**; we can also alter other window attributes, such as border.

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              ,i4
    wndw_2              ,i4

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 8, 20)
    xcall w_area(wndw_1, WA_FILL, "1")
    xcall w_proc(WP_PLACE, wndw_1, 10, 5)
    call pause
    xcall w_proc(WP_OVERLAY, wndw_1, wndw_2, "window2",
&                WP_PLACE, wndw_2, 15, 31)
    call pause
    xcall w_area(wndw_2, WA_FILL, "2")
    xcall w_exit
    stop
```

```

pause,
    xcall w_updt
    sleep 1
    return
end

```

WP_PALET, *color, background, foreground*

WP_PALET changes the color characteristics of the color palette entry stored in the *color* argument.

Color is a numeric expression that contains a color palette number between 1 and 16. The colors that correspond to the palette numbers are system specific. See [“Colors and the color palette” on page 19-5](#) for more information about color.

Background is a three-digit numeric expression that contains the number of the new background color. *Foreground* is a three-digit numeric expression that contains the number of the new foreground color. The *foreground* and *background* numbers can be between 0 and 511 on Windows and between 0 and 255 on UNIX and OpenVMS; the significance of the values is system specific. (On Windows, these values are Synergy Language colors, which can be defaults set by the Synergy runtime or overrides set by the *COLORn* environment variables or previous calls to W_PROC. On UNIX and OpenVMS, the way they are assigned varies.)

WP_PLACE, *id, row, column*

WP_PLACE places a window’s display area on the screen with the upper-left corner at the row and column designated by *row* and *column*. *Id* is a numeric expression that contains the ID of the window whose display area is to be placed. You must place a window on the screen before you can display it.

The position of the window is specified by WP_PLACE. If the window is not yet placed, WP_PLACE places the window at a new position at the top of the placement order. (See [“Window placement and occlusion” on page 19-4](#) for more information about window placement.) If the window is already placed, placement order is maintained and the window is shifted to the new position.

Placing a window that was previously hidden restores the window to its previous state, which by default is normal.

Examples

In this example, we create three windows, fill them with the digits “1,” “2,” and “3,” and place them. Notice that the window that is placed first is on the bottom of the pile.

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1            ,i4
    wndw_2            ,i4
    wndw_3            ,i4

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 8, 20)
    xcall w_area(wndw_1, WA_FILL, "1")
    xcall w_proc(WP_PLACE, wndw_1, 10, 14)
    call pause
    xcall w_proc(WP_CREATE, wndw_2, "window2", 5, 30)
    xcall w_area(wndw_2, WA_FILL, "2")
    xcall w_proc(WP_PLACE, wndw_2, 18, 31)
    call pause
    xcall w_proc(WP_CREATE, wndw_3, "window3", 9, 10)
    xcall w_area(wndw_3, WA_FILL, "3")
    xcall w_proc(WP_PLACE, wndw_3, 12, 26)
    call pause
    xcall w_exit
stop

pause,
    xcall w_updt
    sleep 1
    return
end
```

WP_POSITION, *id*, *x*, *y*

WP_POSITION sets the position of the top left corner of a window on the screen.

Id is either the ID of a window or the following:

WID_CONTAINER The application container window

When *id* is the ID of a nonapplication container window, *x* is the horizontal pixel position on the screen relative to the client area of the window that owns this window, and *y* is the vertical pixel position relative to the client area of the window that owns this window (which is usually the application window).

When *id* is WID_CONTAINER, *x* is the horizontal pixel position on the screen relative to the upper left corner of the screen, and *y* is the vertical position relative to the upper left corner of the screen.

See also

[%W_INFO on page 19-68](#) for information about retrieving a window's position.

Examples

The example below positions the application window at the X, Y coordinates 150, 100.

```
xcall w_proc(WP_POSITION, WID_CONTAINER, 150, 100)
```

WP_REDRAW

WP_REDRAW sets an internal flag so that the entire screen is cleared and completely redrawn on the next screen update. Even if you have used WP_REDRAW more than once prior to a W_UPDT, the screen is only updated once.

This operation is useful when the screen has been changed due to circumstances beyond the windowing system's control, such as the SPAWN subroutine.

WP_REMOVE, *id*

WP_REMOVE removes a window from the screen, where *id* is a numeric expression that contains the ID of the window you want removed. When you remove a window, you can still access and modify it, but it won't be displayed until it is replaced. If you try to remove a window that hasn't been placed, this function is ignored; it won't cause an error.

Examples

In the following program, notice that after we remove the window, we can still alter it (in this case, by filling it with asterisks). Also notice that the WP_MOVE function replaces the window where it was originally situated.

```
.define MAXWINS,      15
.define WNDCHNL,      1
.include "WND:windows.def"
record
    wndw_1              , i4

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)
    xcall w_proc(WP_CREATE, wndw_1, "window1", 8, 40)
    xcall w_area(wndw_1, WA_FILL, "1234567890")
    xcall w_brdr(wndw_1, WB_OFF)
    xcall w_proc(WP_PLACE, wndw_1, 11, 21)
    call pause
    xcall w_proc(WP_REMOVE, wndw_1)
    xcall w_area(wndw_1, WA_FILL, "*")
    xcall w_proc(WP_MOVE, wndw_1, 0, 0)
```

```

        call pause
        xcall w_exit
        stop

pause,
        xcall w_updt
        sleep 1
        return
end

```

WP_RESIZE, *window_id*, *row*, *column*

WP_RESIZE either resizes the specified window or resizes the screen to the specified size and on UNIX and OpenVMS, emits a VT100 escape sequence to change between 80- and 132-column modes as appropriate.

Window_id is one of the following:

- id* Resize the specified window ID.
- WPR_SCREEN** Resize the screen. WPR_SCREEN is defined in **wproc.def**.

Row and *column* are numeric expressions, where *row* is the number of rows deep that you want the window to be and *column* is the number of columns wide that you want the window to be. You can resize a screen to more than 24 rows.

WIN

If you change the container from a value of less than 132 to 132 or greater, the font size does not change unless you have specified the FONT_ALTERNATE initialization setting in the **synergy.ini** or **synuser.ini** file or used the %U_WNDFONT function with the D_ALTFONT132 option to provide a new font or point size.

UNIX, VMS

The number of columns is currently limited to either 80 or 132. (If you resize the number of columns to a value less than 80, it defaults to 80. Resizing to a value greater than 80 defaults to 132.) This constraint does not apply to Windows. If the terminal type is neither VTxxx nor XTERM, no escape sequences are sent to the terminal.

When you resize a window, the processing and display areas are automatically reset to the new size. If the new window size causes any entries in a window field set to fall outside the window, the windowing system redefines the position of those entries to row 1, column 1, and the length to the number of columns in the window. If you increase the number of rows or columns when you resize a window, the new areas are initialized to blanks with no attributes.

Before using WP_RESIZE to resize the screen, make sure that all windows placed with WP_PLACE will fit into the new display area. If they won't all fit, a window manager error (\$ERR_WNDERR) is generated.

WP_SIZE, *id*, *cx*, *cy*

WIN

WP_SIZE changes the size of a Synergy window.

Id is either the ID of a Synergy window or the following:

WID_CONTAINER The application container window

Cx, *cy* are the horizontal and vertical pixel sizes, respectively, for the outside of the window (including any border, caption, and so forth).

This subfunction has no effect on UNIX and OpenVMS, although it generates an error if an invalid window ID is passed.

The window is sized to the specified outer dimensions exactly as if the user had dragged the border to that size (although you can also use this subfunction on windows that do not have a size method or border). If the window has a size method, that method is invoked if the pixel size changes in either direction.

WP_STATE, *id*, *state*

WP_STATE modifies the state of a window (minimized, maximized, hidden, or normal).

Id is either the ID of the window whose state you want to modify or the following:

WID_CONTAINER The application container window

State is one of the following:

WPS_NORMAL Restore the window to its normal state.

WPS_MAXIMIZED Maximize the window.

WPS_MINIMIZED Minimize the window.

WPS_HIDDEN Hide the window.



WPS_HIDDEN on a Synergy window is equivalent to WP_REMOVE. Likewise, WPS_NORMAL on a Synergy window executes a WP_MOVE, *id*, 0, 0.

See also

[%W_INFO](#) on page 19-68 for information about retrieving a window's state.

Examples

The example below sets the application window state to be minimized.

```
xcall w_proc(WP_STATE, WID_CONTAINER, WPS_MINIMIZED)
```

This next example hides the window specified by **wndid**.

```
xcall w_proc(WP_STATE, wndid, WPS_HIDDEN)
```

WP_TOFRONT, *id*

WP_TOFRONT argument brings a window to the front of the display without changing the window's position (unless the window was not previously placed).

Id is a numeric expression that contains the window ID.

Examples

The example below brings **wndw_3** to the front of the placement order.

```
xcall w_proc(WP_TOFRONT, wndw_3)
```

W_RESTORE – Restore a window that was previously compressed

```
xcall W_RESTORE (func_status, id, buffer[, options[, name]])
```

Arguments

func_status

Passed as the value CTL_BEGIN on the initial call to W_SAVE, and is returned with the one of the following status values: **(n)**

- CTL_BEGIN (1)** Start the restore operation.
- CTL_MORE (2)** Another call to W_RESTORE is required.
- CTL_DONE (0)** Restore operation is complete.

id

Returned with the ID of the restored window. **(n)**

buffer

Contains the information to use for the restore function. **(a)**

options

(optional) One or more of the following options, combined using .bor. or a vertical bar (|): **(n)**

- OPT_NODATA** Don't restore saved window's text and attributes.
- OPT_NOTITLE** Don't restore saved window's title.
- OPT_NOWFLDS** Don't restore any saved window field sets.
- OPT_NOUSET** Don't restore any saved user data sets.
- OPT_NEWNAME** Don't use the saved window's name.

name

(optional) A new name to be used for the restored window. **(a)**

Discussion

W_RESTORE creates a window from one that was previously compressed with W_SAVE.

The windowing system assumes that the window being restored was previously saved using W_SAVE, and that the information buffers returned by W_SAVE are passed to W_RESTORE in the exact order in which they were obtained.

On the initial call to `W_RESTORE`, *func_status* must be the value `CTL_BEGIN`. If *func_status* is returned with the value `CTL_DONE`, *buffer* contains all of the information required to restore the window, and the restore operation is complete. However, if *func_status* is returned with the value `CTL_MORE`, multiple calls to `W_SAVE` are required to save the window, and you must repeatedly call `W_RESTORE` to get the additional buffer information until `CTL_DONE` is returned. If *func_status* returns the value `-1`, an error occurred in the restore operation.

On the final call to `W_RESTORE` (`CTL_DONE` is returned), the options specified by *options* determine which components of the window are not to be restored. If the `OPT_NEWNAME` option is specified, *name* must be specified. If *name* is a null string, a unique name based on the returned window ID is created.

The “CTL_” and “OPT_” codes are defined in **wmisc.def**.

Here’s how you might code a window restore operation:

```

    initialize for window restore input
    set options to any restore options, or zero if none
    set name if OPT_NEWNAME is set in options

fnc_sts = CTL_BEGIN
while (fnc_sts)
    begin
        .
        .   input buffer data from prior W_SAVE
        .
        if (fnc_sts.lt.0)
            .
            .   generate error
            .
        xcall w_restore(fnc_sts, wndid, buffer, len, opts, wname)
    end
    .
    .   terminate window restore input
    .

```

W_SAVE – Save a window in compressed form

```
xcall W_SAVE (func_status, id, buffer, length[, options])
```

Arguments

func_status

Passed as the value CTL_BEGIN on the initial call to W_SAVE and is returned with the one of the following status values: (n)

- CTL_BEGIN (1)** Start the save operation.
- CTL_MORE (2)** Another call to W_SAVE is required.
- CTL_DONE (0)** Save operation is complete.

id

The ID of the window you want to save in compressed form. (n)

buffer

The buffer to use while performing the save function. The size of this buffer must be at least 400 characters. (a)

length

Returned with the number of bytes loaded into *buffer*. (n)

options

(optional) One or more of the following options, combined using .bor. or a vertical bar (|): (n)

- OPT_NODATA** Don't save window's text and attributes.
- OPT_NOTITLE** Don't save window's title.
- OPT_NOWFLDS** Don't save any window field sets.
- OPT_NOUSET** Don't save any user data sets.

Discussion

W_SAVE saves a window in compressed form.

On the initial call to W_SAVE, *func_status* must be the value CTL_BEGIN. If *func_status* is returned with the value CTL_DONE, *buffer* is large enough to hold the entire saved window, and the save operation is complete. However, if *func_status* is returned with the value CTL_MORE, the size of *buffer* is not large enough, and you must repeatedly call W_SAVE until CTL_DONE is returned. If *func_status* returns the value -1, an error occurred in the save operation.

On the initial call to W_SAVE, the options specified by *options* determine which components of the window are not to be saved.

The “CTL_” and “OPT_” codes are defined in **wmisc.def**.

Here’s how you might code a window save operation:

```
<initialize for window save output>

<set options to any save options, or zero if none>

fnc_sts = CTL_BEGIN
while (fnc_sts)
  begin
    xcall w_save(fnc_sts, wndid, buffer, len, opts)
    if (fnc_sts.lt.0)
      <generate error>
      <output buffer(1:len) data>
    end
  end

<terminate window save output>
```

W_UPDT – Update the physical screen

```
xcall W_UPDT
```

Discussion

W_UPDT is one of four ways to update the physical screen.

A window display changes only when the physical screen is updated to reflect changes in the logical screen. W_UPDT and the WD_UPDT function of W_DISP are the only way to explicitly update the physical screen. You can also implicitly update the screen with W_EXIT or with the window input functions WD_READS, WD_GETS, WD_ACCEPT, or WF_INPUT, which automatically update the screen before data is read in. (The screen is also implicitly updated by the WA_INSERT and WA_SCROLL functions when hardware scrolling is in effect.)

Because of this automatic updating, you won't need to use W_UPDT too frequently. You will need it whenever you want to display something when no terminal input is occurring. For example, you could use it to display a "Please wait..." message during a long process.

W_UPDT updates the entire screen when a WP_REDRAW function was processed earlier. If W_UPDT is not preceded by a WP_REDRAW function, only those positions that have been modified are updated.

Examples

In this simple call, we use a screen update to display a "Please wait" message.

```
wait,  
    xcall w_proc(WP_CREATE, window, choice, 1, 15,  
    &           WP_PLACE, window, 24, 32)  
    xcall w_brdr(window, WB_OFF)  
    xcall w_disp(window, WD_ATTR, ATTR_SET + ATTR_BLNK + ATTR_BOLD,  
    &           "Please wait")  
    xcall w_updt  
    return
```

Error Processing in the Windowing System

This section describes how to detect and prevent runtime errors in a program that uses the windowing routines.

Trapping errors

The windowing system does not trap runtime errors in the same way standard Synergy Language programs trap errors. Most window errors are considered fatal unless you precede the faulty function with the function `W_RETERR`. (Exceptions are errors generated by functions such as `WD_READS`, `WD_GETS`, and `WD_WAIT`, which do return normal, trappable Language errors.) If the function that causes an error is preceded by a `W_RETERR` function in a subroutine's argument list, your program continues to run. However, once you've trapped an error with `W_RETERR`, the rest of the functions in that subroutine are not executed.

`W_RETERR`, *errfield*

The `W_RETERR` function traps an error and returns error information in the **errfield** record, which has the following layout:

```
record errfield
    error_number      ,d2
    routine_name      ,a6
    arg_number        ,d2
    error_text         ,a50
```

You can use `W_RETERR` in any windowing subroutine that uses regular functions. For example, you can use `W_RETERR` to trap errors in the following subroutines:

```
xcall w_disp(id, W_RETERR, errfield, ...)
xcall w_area(id, W_RETERR, errfield, ...)
xcall w_proc(W_RETERR, errfield, ...)
```

To detect an error, you must test the contents of the **error_number** field. If the field contains a value of zero (in other words, the field tests “false”), no error has occurred. If the field contains any number other than zero, an error has occurred. The error number, routine name, argument number, and error text that are returned identify the error.

Examples

If you wanted to prevent a fatal error on a `WD_READS` function, use the following code, which shows the argument definitions necessary to store the returned information. We use the `W_RETErr` function to avoid the fatal window error #11, "Input field not completely visible." This error occurs when you attempt to obtain input from a portion of the window that is partially or completely occluded. To avoid this problem, simply move the window to the top of the pile with the `WP_MOVE` function.

```
.define WNDCHNL,      1
.define MAXWINS,     10
.include "WND:windows.def"
record wnd_ids
    wndw_1            ,i4
    wndw_3            ,i4
record input
    selection         ,a1
record errfield
    error_number      ,d2
    routine_name      ,a6
    arg_number        ,d2
    error_text        ,a50

proc
    open(WNDCHNL, o, "tt:")
    xcall w_init(0, WNDCHNL, MAXWINS)

;Create window1 (menu)

    xcall w_proc(WP_CREATE, wndw_1, "menu", 12, 30)
    xcall w_brdr(wndw_1, WB_ATTR, ATTR_SET + ATTR_RVRS,
&                WB_TITLE, "Record Maintenance",
&                WB_TPOS, WBT_TOP, WBT_CENTER)
    xcall w_disp(wndw_1, WD_POS, 2, 2, "1. Display",
&                WD_POS, 4, 2, "2. Enter",
&                WD_POS, 6, 2, "3. Update",
&                WD_POS, 8, 2, "4. Delete",
&                WD_POS, 10, 2, "Selection? ")
    xcall w_proc(WP_PLACE, wndw_1, 1, 1)

;Create window3 (error window)

    xcall w_proc(WP_CREATE, wndw_3, "error", 10, 35)
    xcall w_disp(wndw_3, WD_POS, 2, 2,
&                "Invalid selection", WD_POS, 3, 2,
&                "Please press the number", WD_POS, 4, 2,
&                "corresponding to your", WD_POS, 5, 2,
&                "menu selection.")
```

```
xcall w_proc(WP_PLACE, wndw_3, 5, 10)
xcall w_disp(wndw_1, W_RETERR, errfield, WD_READS, selection)
if (error_number .eq. 11)
  begin
    xcall w_proc(WP_REMOVE, wndw_1, WP_MOVE, wndw_1, 0, 0)
    xcall w_disp(wndw_1, WD_READS, selection)
  end
done,
  xcall w_exit
stop
end
```

Window errors

All window errors are first categorized as a Synergy Language “Window Manager error” (\$ERR_WNDERR).

If a window error occurs, you’ll first receive the “Window Manager error”. Then you’ll receive two lines of informational errors to give you more details. The informational messages tell you the routine name, the number of the argument that triggered the error, and the window error number. The next line gives you the window error message that describes the problem. Finally, a fourth line indicates the line number and the routine in which the error occurred.

See “[Window error messages](#)” in the “Error Messages” chapter of *Synergy Language Tools* for a list of window error numbers and their accompanying error text.

Converting Your Code to Use the Windowing API

This section describes the straight conversion method of converting your application to use the windowing subroutines. This method uses one window for all screen I/O, and it doesn't use any of the UI Toolkit features. The conversion only involves a few steps, but the improvement in speed can be significant.

The straight conversion strategy assumes the following about your application:

- ▶ It has localized input and output subroutines.
- ▶ It chains between main menus and has some kind of a start-up dispatcher routine. (For example, in MCBA-style systems, this subroutine is called TERID.)

If your application doesn't meet these criteria, your conversion could be more complex.

Converting your code

You need to add or change code in four places:

Start-up subroutine

1. Call `W_INIT` to initialize the windowing system and test whether the system has already been initialized.
2. Define a global data section with a variable for the window ID.
3. Create a borderless 24 x 80 window and place it on the screen at position (1,1).

Input subroutine

1. Include the global data section that has the global window ID.
2. Change your Synergy Language `READS` and `ACCEPT` statements to use `W_DISP` with the `WD_READS` and `WD_ACCEPT` functions.

Output subroutine

1. Include the global data section that has the global window ID.
2. Change your Synergy Language `WRITES` and `DISPLAY` statements to use `W_DISP`. Use the `WD_FIELD` function for decimal displays and the `WD_ANSI` function for ANSI escape sequences.

Message displays

Whenever you display a message on the screen that isn't followed by terminal input, use `WD_UPDT` or `W_UPDT` to update the screen.

Updating the screen

The screen is only updated implicitly by an input function or explicitly by calling `W_UPDT` or the `WD_UPDT` function of `W_DISP`.



Do *not* use `W_UPDT` or `W_DISP(WD_UPDT)` after each screen change: it slows down your screen I/O. The only time you need to use `W_UPDT` is when you change the screen and then don't do any screen input.

For example, if you display a "Working...Please Wait" message, call `W_UPDT` immediately after making the display (or use the `WD_UPDT` function in the `W_DISP` call) so you update the screen to show the message. If you don't explicitly update the screen in this instance, the user may never see the message!

Examples

Here's an example of the additions you can make to your start-up subroutine:

```
global data section gwindow
record
    g_wndid    ,i4
endglobal
.define TTCHN,1
record
    iflg      ,i4           ;Initialization test flag
    .
    .
    .
proc
    .
    .
    .
    xcall w_init(0, TTCHN, 4, 24, 80, iflg)
    if (.not.iflg)          ;Is it already set up?
        begin              ;No, create it w/o border, place it
            xcall w_proc(WP_CREATE, g_wndid, "main", 24, 80)
            xcall w_brdr(g_wndid, WB_OFF)
            xcall w_proc(WP_PLACE, g_wndid, 1, 1)
        end
    .
    .
    .
end
```

In the input and output routines calls, change the following:

From	To
READS	WD_READS
ACCEPT	WD_ACCEPT
alpha display	W_DISP
decimal display	WD_FIELD

For example:

From	To
<code>reads(TTCHN, avar)</code>	<code>xcall w_disp(g_wndid, WD_READS, avar)</code>
<code>accept(TTCHN, achr)</code>	<code>xcall w_disp(g_wndid, WD_ACCEPT, achr)</code>
<code>display(TTCHN, avar1, & avar2, dvar)</code>	<code>xcall w_disp(g_wndid, avar1, avar2, WD_FIELD, & dvar)</code>
<code>writes(TTCHN, avar)</code>	<code>xcall w_disp(g_wndid, WD_WRITES, avar)</code>

20

Synergy .NET Assembly API

Introduction to the Synergy .NET Assembly API 20-2

Discusses the purpose of the Synergy .NET assembly API, requirements and restrictions, callback routines, error handling, and debugging.

Error handling 20-3

Describes the DotNetException class.

Synergy .NET Assembly API Methods 20-9

This API includes the following methods:

DotNetAssembly – Load an assembly.....	20-10
DotNetObject – Create an instance of a .NET type.....	20-12
DotNetObject.Call – Invoke methods of an instance	20-13
DotNetObject.SetProperty – Set the value of a property on an instance	20-15
DotNetObject.GetProperty – Query the value of a property	20-17
DotNetObject.SetField – Set the value of a field on an instance.....	20-19
DotNetObject.GetField – Query the value of a field.....	20-21
DotNetObject property indexer set method.....	20-23
DotNetObject property indexer get method	20-24
DotNetDelegate – Create an instance of a delegate.....	20-25
DotNetObject.AddDelegateToEvent – Associate a delegate with an event.....	20-27
DotNetObject.RemoveDelegateFromEvent – Remove a delegate from an event.....	20-28

Introduction to the Synergy .NET Assembly API

The Synergy .NET assembly API enables you to interface to .NET from a non-.NET environment, so your existing code can take advantage of its features without migrating to .NET. Using the .NET assembly API, you can load a .NET assembly; instantiate types defined in that assembly; and communicate with the methods, properties, fields, and events of those objects. By adding .NET components to your application, you can extend its capabilities with a wide variety of interactive and multimedia effects.



We recommend that you use this API for reference only, and instead use the **gennet** utility to generate wrappers if you want to have natural Synergy syntax for interoperation with .NET. Using **gennet** will also enable your syntax to move almost seamlessly to Synergy .NET in the future. See “[The gennet Utility](#)” in the “General Utilities” chapter of *Synergy Language Tools* for more information.

Requirements

To use the .NET assembly API, the .NET Framework 2.0 Service Pack 1 must be installed on *each* computer that uses the API. (We recommend that you use .NET Framework 3.5 Service Pack 1, which is a cumulative update that includes 2.0.) This means that if the Synergy application that uses the .NET assembly API is installed in a shared configuration, the Framework must be installed on each client machine. Neither the Synergy/DE installation nor the Synergy/DE client installation check for or install the .NET Framework. You can obtain Framework updates through Windows Update. If you use Windows Server Update Services, you can force this update to be applied to all computers that need it.

The Synergy .NET assembly API has the following additional requirements and restrictions:

- ▶ The .NET Framework must match the address size of the runtime used. (In other words, the 32-bit .NET Framework must be used for the 32-bit runtime, and the 64-bit .NET Framework must be used for the 64-bit runtime.)
- ▶ If you use STOP chains with this API, .NET assemblies will not be unloaded, and if static members are set in those assembly instances, they will still be set if the assembly is accessed in the chained-to program. Chaining does not remove any threads for the loaded assembly. If a .NET method creates any threads, those threads are not shut down on a STOP chain; they are only shut down when the **dbx.exe** process exits.
- ▶ Any UI components that are automated with this interface should operate modally. The API does not expect an application to leave a UI component displayed and simultaneously enable access to Synergy/DE components.
- ▶ Synergy code that uses the API cannot be compiled on non-Windows systems.
- ▶ The API does not support passing the result of a property call (that is, a .NET property) into a method that accepts it as a by-reference parameter.

- ▶ Static methods and fields are not directly supported, except through the assembly object.
- ▶ If you deploy an application created with the .NET assembly API in a shared configuration, you will need to deal with the challenges of the .NET security implementation, which include intranet security zone settings and Code Access Security Policy (caspol). Note that these security issues apply any time you use a .NET component on a network share—not just .NET components used with Synergy/DE products. See Synergex KnowledgeBase article synkb100001853 for details. (This is not an issue in .NET Framework 3.5 Service Pack 1.)

Callback routines

A callback routine in .NET is called a delegate. A delegate is a Synergy Language routine that is called whenever a specific event is raised by the .NET Framework. A delegate must be created with the DotNetDelegate constructor and bound to the event using the AddDelegateToEvent method. You can write to arguments in callback routines if the delegate defines them as writable. The .NET parameters for delegates determine the type of argument passed to the Synergy method. (See [DotNetObject.AddDelegateToEvent on page 20-27](#) for syntax and more information on the conversion rules.)

You can add and remove event handlers for code generated by the **gennet** utility using the **ADDHANDLER** and **REMOVEHANDLER** statements, respectively. See [ADDHANDLER on page 4-13](#) and [REMOVEHANDLER on page 4-162](#) for more information.

Be aware of synchronicity with your callback routines. The .NET common language runtime (CLR) may fire many simultaneous events as the user moves the mouse, for instance. While your program is in a callback routine, it can be interrupted and the same callback called recursively. Ensure all callback routines are re-entrant, and don't use writable global data.

Error handling

The DotNetException class represents an exception raised by the .NET Framework. Any exception thrown by the .NET assembly code triggers a DotNetException exception, which references and wraps the .NET exception so you can catch it and then look at the native .NET exception properties using the .NET assembly API.

The DotNetException class consists of the following members:

Object member	Description
Message	The text of the Message property of the original .NET Framework exception.
NativeException	A public property of type DotNetObject. It contains a reference to the original exception raised by .NET Framework.

If an exception is thrown (or a runtime error is generated that is not caught by either a TRY-CATCH or ONERROR statement) while executing code in a Synergy delegate, the exception will unwind the stack up to the point where .NET code is executing. The interop layer throws a .NET exception that can be caught by the .NET assembly code, allowing it to handle the exception or rethrow it back to Synergy. When possible, the exception object will have accurate Synergy line information for the error location, as well as a stack trace.

When crossing the Synergy/.NET boundary, the interop layer attempts to convert to a native exception type (for example, System.Exception, which exists in both Synergy and .NET). When a conversion is not possible, .NET will throw a dynamically generated exception that inherits from System.Exception. The interop layer converts .NET exceptions to a Synergex.Synnet.DotNetException when they cannot be converted to a native Synergy runtime exception.

Using .NET assemblies referenced by the Synergy .NET assembly API

Whenever the version or strong name of an assembly (or a dependent assembly) changes, the Synergy wrappers must be regenerated and recompiled. We recommend that you don't use auto-incrementing version numbers on assemblies used with the .NET assembly API. The version should change whenever any of the method signatures or existing interfaces are changed. The version does not have to change if new methods or interfaces are added, but you must regenerate your wrappers to take advantage of these new entities. This is because the wrappers generated by **gennet** load assemblies either from a Synergy environment variable location or from the .NET display name, and specific versions of the assemblies and wrappers will need to change if these locations or versions change.

We also strongly recommend the following:

If you are...	We recommend that you...
Not going to place your assemblies in the same location as your .dbr files	Put these assemblies in the GAC so the generated wrappers will work on any system
Distributing software, rather than creating an in-house solution	Strong name your assemblies
Using shared installations for your .dbr files and assemblies	Use the environment variable form instead of putting them in the GAC, and add a local policy to allow them to load



You cannot use **dbr.exe.config** to set your .NET settings, because Microsoft does not support an **app.exe.config** file with hosted .NET applications.

If an assembly is loaded into **gennet** as a dependency and later at runtime has one of its types loaded before the original loading assembly, it will not be able to find the DLL outside of the GAC or currently running **dbt.exe** directory. If an assembly dependency fails to load from the GAC or the current **gennet.exe** directory, **gennet** will try to load from each of the directories from which it has successfully loaded other assemblies. This list of directories is processed in order of appearance. If an attempt is made to load a duplicate assembly (with the same display name but a different file path), the DLL will not be found, and a “Could not load assembly” error (\$ERR_NOLOAD) will be generated.

Workbench support

Professional Series Workbench facilitates the use of .NET assemblies in your applications as long as you use **gennet** (rather than the Synergy .NET API) to generate your wrappers. Including the files generated by **gennet** in a separate tag file allows the tag information for the .NET assemblies to be used across multiple projects. See “[Tagging imported classes](#)” in the “Developing Your Application in Workbench” chapter of *Getting Started with Synergy/DE* for the most efficient way to tag your classes in Workbench.

UI Toolkit support

To integrate .NET Windows Forms into UI Toolkit, you must use a special Toolkit container window. A Toolkit container window has the basic characteristics of a Toolkit window (character-based increments, placement order, maintained environment, and title manipulation, for example), yet it has the display and operational characteristics of a .NET container (load controls, get/set property values, call methods, bind callback routines, and so forth).

Debugging

The .NET assembly API includes a debugging system to enable you to analyze problems in your code. To enable debugging, set the SYNNET_DEBUG environment variable to 1 before the first assembly is loaded. (See [SYNNET_DEBUG](#) in the “Environment Variables” chapter of *Environment Variables and System Options* for more information.)

When debugging is enabled, debug messages are logged using the OutputDebugString Windows API function. You can view these messages using Visual Studio, other compatible C-level debuggers, or the **debugview** program from <http://www.sysinternals.com>.

Debug messages are logged when any of the following occur:

- ▶ An assembly is loaded, along with its parameters and returned object ID.
- ▶ An object or delegate is created, along with its parameters and returned ID.
- ▶ Any method or property is invoked, along with its parameters and return value.
- ▶ Any field is accessed, along with its value.

- ▶ Any delegate is associated with or removed from an event.
- ▶ Any error is encountered. The information from the exception is logged and includes the following:
 - ▶ The typename of the exception (fully qualified)
 - ▶ The Message property of the exception
 - ▶ The Source property of the exception
 - ▶ The StackTrace property of the exception

If the `InnerException` property is non-null, the same information is recursively output for the `InnerException`.

When the Synergy runtime shuts down, the debug output includes a count of objects deleted at shutdown. If this count is nonzero, objects were not released by the application.

Sample programs

Example 1

```
main
record
    asm          ,@DotNetAssembly
    obj          ,@DotNetObject
    obj2         ,@DotNetObject
    obj3         ,@DotNetObject
    itmp         ,i4
endrecord

proc
    open(1,o,'tt:')
    asm = new DotNetAssembly("mscorlib")

    ;Constructor with parameter
    ;DateTime(Int64) initializes a new instance of the DateTime structure to
    ; a specified number of ticks.
    obj = new DotNetObject(asm, "System.DateTime", 34567)

    ;Call example, capitalization is important in object names
    writes(1, (string)obj.Call("ToString"))

    ;Call example with parameter, (void) is to prevent a level 4 compiler warning
    ;AddHours adds the specified number of hours to the value of this instance.
    (void)obj.Call("AddHours", 5.5)

    ;Static call example with parameter
    obj2 = (DotNetObject)((DotNetObject)asm).Call("System.DateTime.Parse",
    & "13 March 1986")
```

```

;Get property
writes(1, %string((int)obj.GetProperty("Hour")))
;Get static property
obj = (DotNetObject)((DotNetObject)asm).GetProperty("System.DateTime.Now")
;No boxing form
obj.GetProperty("Hour", itmp)
writes(1, %string(itmp))

obj3 = new DotNetObject(asm, "System.Collections.BitArray", (int)102293);
writes(1, %string((int)obj3[1]))
endmain

```

Example 2

```

main
record
    asm                ,@DotNetAssembly
    sys                ,@DotNetAssembly
    form               ,@DotNetObject
    button              ,@DotNetObject
    form_OnLoad         ,@DotNetDelegate
    button_OnClick      ,@DotNetDelegate
    callbackObject      ,@ns1.cls1
    controls            ,@DotNetObject
endrecord

proc
    open(1,o,'tt:')
    ;Display name-based loading
    asm = new DotNetAssembly("System.Windows.Forms, Version=2.0.0.0,
Culture=neutral,
& PublicKeyToken=b77a5c561934e089")
    sys = new DotNetAssembly("mscorlib")
    callbackObject = new ns1.cls1()
    form = new DotNetObject(asm, "System.Windows.Forms.Form")
    button = new DotNetObject(asm, "System.Windows.Forms.Button");
    button_OnClick = new DotNetDelegate(sys, "System.EventHandler", ^NULL,
& "ns1.cls1.static_meth1")
    form_OnLoad = new DotNetDelegate(sys, "System.EventHandler", callbackObject,
& "meth2")

    button.AddDelegateToEvent("Click", button_OnClick)
    form.AddDelegateToEvent("Load", form_OnLoad)

    button.SetProperty("Text", "button1")
    form.SetProperty("Text", "form1")
    controls =
(DotNetObject)((DotNetObject)form.GetProperty("Controls")).Call("Add",
& button)

; (void) suppresses the level 4 warning about not using the return value
(void)((DotNetObject)asm).Call("System.Windows.Forms.Application.Run", form)

```

```
;This call isn't strictly required, as the references are cleaned up
; automatically when the delegate goes out of scope.
button.RemoveDelegateFromEvent("Click", button_OnClick)
form.RemoveDelegateFromEvent("Load", form_OnLoad)
endmain

namespace ns1
  class cls1
    public static method static_meth1, void
      sender, @DotNetObject
      e, @DotNetObject
    endparams

    proc
      writes(1, "In meth1 callback")
      mreturn
    endmethod

    public method meth2, void
      sender, @DotNetObject
      e, @DotNetObject
    endparams

    proc
      writes(1, "In meth2 callback")
      mreturn
    endmethod
  endclass
endnamespace
```

Synergy .NET Assembly API Methods

The Synergy .NET assembly API methods belong to one of the following classes:

- ▶ `DotNetAssembly`, which represents a .NET assembly
- ▶ `DotNetObject`, which represents an instance of a .NET object
- ▶ `DotNetDelegate`, which represents a .NET delegate

DotNetAssembly – Load an assembly

WIN

```
assembly = new DotNetAssembly(name)
```

Arguments

assembly

The returned object (@DotNetAssembly) representing the assembly.

name

A required alphanumeric file path, filename, partial name, or display name for the assembly.

Discussion

The DotNetAssembly constructor loads a .NET assembly from a file path, filename, partial name, or display name. When this constructor is called, if the .NET CLR has not yet been loaded, it will be loaded using the following default settings:

- ▶ The latest installed version of the CLR
- ▶ The workstation build flavor (**wks**)
- ▶ Single-domain loader optimization

Unless the MAXMEM environment variable is set to 50,000,000 or higher, the constructor uses a special mode of operation that reduces memory at the expense of more garbage collection. (The default 32-bit CLR uses up to 64 MB of memory per process and the 64 bit CLR uses up to 128 MB of memory per process when this special mode is not being used.) If the CLR cannot be loaded for any reason, an \$ERR_NODOTNET will occur.

Name can include an environment variable using the standard Synergy syntax (for example, MYVAR:) unless the .NET display name form is used. If neither an environment variable nor the .NET display name is used, *name* is searched for in the following locations, in order:

- ▶ The .NET global assembly cache (GAC)
- ▶ The location of **dbf**
- ▶ The dbf\bin directory
- ▶ The windows\system32 directory



The .NET assembly API does not support search paths. If you specify an environment variable that is set to multiple directory paths, DotNetAssembly will only use the first one.



If you are loading from the GAC and didn't specify a display name, do not include the filename extension in *name*. For example, **system.windows.forms.dll** is not valid; use **system.windows.forms** instead.

Name cannot be an *xfServer* file designation (for example, "file@server").

If the assembly cannot be loaded, one of the following Synergy errors will be generated:

- ▶ "File not found" (\$ERR_FNF)
- ▶ "Protection violation" (\$ERR_PROTEC)
- ▶ "Could not load assembly" (\$ERR_NOLOAD)

Loading exceptions generated by the .NET Framework are logged to the debug viewer if it is present.

The assembly is loaded using only the security evidence provided by the loader. If a security exception is thrown (for example, if you try to load an "unsafe" assembly from a mapped drive without the local client machine policy being set to specifically allow such access in the intranet zone), an \$ERR_PROTEC error is generated.

Dependent assemblies are loaded automatically by the .NET assembly loader. If an assembly with the same name has already been loaded, the assembly that was loaded previously will be used. This may not be what you expect, and you should use the debug viewer to ensure that the DLLs load from the correct places.

If an attempt is made to load a duplicate assembly (with the same display name but a different file path), the DLL will not be found, and a "Could not load assembly" error (\$ERR_NOLOAD) will be generated.

If you delete an assembly, all of the objects created from that assembly are deleted as well.

Examples

The example below loads a new assembly called **mscorlib**. See ["Sample programs" on page 20-6](#) for an example in the context of a complete program.

```
asm = new DotNetAssembly("mscorlib")
```

DotNetObject – Create an instance of a .NET type

WIN

```
object = new DotNetObject(assembly, type[, arg ...])
```

Arguments

object

The returned object instance (@DotNetObject).

assembly

The assembly in which this component is found.

type

An alphanumeric fully qualified type name, which includes the type's namespace but not its assembly name.

arg

(optional) One or more constructor arguments.

Discussion

A constructor for the specified type will be invoked based upon the *arg* arguments passed. See [DotNetObject.Call on page 20-13](#) for a description of how arguments are converted to .NET types.

To directly load a System type (for example, System.String) using this API, the corresponding .NET assembly must be loaded in order to retrieve a DotNetAssembly object that corresponds to it, even though this system assembly was already loaded automatically when the CLR was loaded. The search for *type* is case-sensitive.

If the type cannot be found in the specified assembly or it is not public, an \$ERR_MISSFLD error will occur.

Examples

The example below initializes a new instance of System.Windows.Forms.Form. See [“Sample programs” on page 20-6](#) for an example in the context of a complete program.

```
form = new DotNetObject(asm, "System.Windows.Forms.Form")
```

DotNetObject.Call – Invoke methods of an instance

WIN

retval = *object*.Call(*methodname* [, *arg* ...])

Arguments

retval

The return value of the method (@System.Object).

object

The handle of an object returned by the DotNetObject constructor, the DotNetAssembly constructor, or some other method or property that returns an object.

methodname

The name of the method to call.

arg

One or more arguments for the method.

Discussion

Methodname must be a public or public static method defined for the specified object; otherwise, an \$ERR_MISSMETH error will occur.

If a *method* is defined more than once using different parameters, the default binder is used to attempt to locate the best fit for the parameters passed. The binder attempts an exact match, a conversion match honoring integer variable sizes matching the same or larger types, and finally, a conversion match from any int to any int. If no match is found, an \$ERR_MISSMETH error will occur.

The type and number of arguments in *arg* must exactly match one of the argument signatures for the method; optional arguments and variable-length argument lists are not supported.

The system attempts to convert parameters from Synergy/DE types to .NET types via an intermediate type according to the rules below:

- ▶ Alpha is converted to System.String.
- ▶ Implied-decimal is converted to System.Decimal.
- ▶ If the target is float or double the value passed is converted to that type.
- ▶ Decimal is converted to Int64.
- ▶ If the target is Boolean, Byte, Int16, Int32, Int64, SByte, UInt16, UInt32, or UInt64, the Int64 value passed is converted to that type.
- ▶ Integer arguments rely on the types of the arguments expected.

- ▶ An enumeration is treated like an **i4**, and an **i4** can be passed to a .NET enumeration.
- ▶ If there is no exact match, integer arguments are converted to Boolean, Byte, Int16, Int32, Int64, SByte, UInt16, UInt32, or UInt64.

The intermediate type is then converted by the standard .NET conversion functions as permitted by the binder. It is important to follow the .NET rules for conversions, or the .NET Framework will generate an exception. A .NET exception is also caused by the following circumstances:

- ▶ A negative value is passed to an unsigned type.
- ▶ A value in a larger type is passed to a smaller type (for example, Int32 containing 100000 is passed to Int16).

To pass a null object as a parameter, use ^NULL.

Objects are returned as follows, depending on the .NET type:

.NET type returned	Returned object
System.String	Synergy System.String
System.Decimal	Boxed Synergy implied-decimal value
One of the integer types described above	Boxed Synergy integer value of appropriate size
Any other returned .NET type	Type DotNetObject, wrapping the returned .NET object

If *object* is an object returned from the DotNetAssembly constructor, only static methods can be accessed using the *namespace.class.method* form of *methodname*.

Examples

The example below invokes DateTime.Parse. See [“Sample programs” on page 20-6](#) for an example in the context of a complete program.

```
obj2 = (DotNetObject)((DotNetObject)asm).Call("System.DateTime.Parse",  
        & "13 March 1986")
```

DotNetObject.SetProperty – Set the value of a property on an instance

WIN

object.SetProperty(propertyname, value)

Arguments

object

An object returned by the DotNetObject constructor, the DotNetAssembly constructor, or some other property or method that returns an object.

propertyname

The name of the property to set.

value

The new value for the property.

Discussion

Propertyname must be a public or public static property defined for the specified object, and a set method must exist for it so that it is not read-only. Otherwise, an \$ERR_MISSMETH error will occur.

If a property is defined more than once using different value types, the default binder is used to attempt to locate the best fit for the parameter passed. If no match is found, an \$ERR_MISSMETH error will occur

The *value* parameter will be converted from a Synergy/DE type to a .NET type using the same rules as for [DotNetObject.Call](#) on page 20-13.

If a structfield member is to be set, *object* will be the value of the structfield returned from DotNetObject.GetProperty or DotNetObject.GetField. Structs are boxed objects (copies), and once the members have been updated with DotNetObject.SetField, the struct must be copied back to the original field or property location with DotNetObject.SetField or DotNetObject.SetProperty using the boxed struct returned from the original DotNetObject.GetField or DotNetObject.GetProperty.

If *object* is an object returned by the DotNetAssembly constructor, only static properties can be accessed using the *namespace.class.property* form of *propertyname*.

Examples

The example below sets the Text property to “button1”. See [“Sample programs” on page 20-6](#) for an example in the context of a complete program.

```
button.SetProperty( "Text", "button1" )
```

DotNetObject.GetProperty – Query the value of a property

WIN

object.GetProperty(*propertyname*, *value*)

or

value = *object*.GetProperty(*propertyname*)

Arguments

object

An object returned by the DotNetObject constructor, the DotNetAssembly constructor, or some other property or method that returns an object.

propertyname

The name of the property to query.

value

Returned with the coerced value for the property.

Discussion

If *object* is an object returned from the DotNetAssembly constructor, only static properties can be accessed using the *namespace.class.property* form of *propertyname*. *Propertyname* must be a public or public static property defined for the specified object, and a get method must exist for it so that it is not write-only. Otherwise, an \$ERR_MISSMETH error occurs.

If a property is defined more than once using different value types, the default binder will be used to attempt to locate the best fit for the parameter passed. If no match is found, an \$ERR_MISSMETH error occurs.

If the first GetProperty syntax above is used (that is, the value is returned in a writable parameter), the value of the property will be coerced to the type of the passed argument as follows:

- ▶ If the argument is of type string, the property value is converted to a Synergy string, using Object.ToString if it is not a .NET string.
- ▶ If the argument is of type integer, the property value is converted to integer if possible. If not, an \$ERR_INVCAST error occurs.
- ▶ If the argument is of type decimal, the property value is converted to a string and passed to the Synergy store subsystem for conversion to decimal. If a conversion error occurs, an \$ERR_DIGIT error occurs.

- ▶ If the argument is of type DotNetObject or System.Object, the value of the property is wrapped in a DotNetObject and returned. If the property value is null, ^NULL is returned instead.
- ▶ If the argument is of type decimal or integer, truncation may occur if the argument is not large enough (for example, if a UInt32 property is returned into an **i4** variable).

If the second GetProperty syntax is used, the value is coerced following the rules in the table on [page 20-14](#), and then the coerced type is boxed as a System.Object if it is not already an object type.

Struct members can only be accessed after returning an instance of the struct using DotNetObject.GetProperty or DotNetObject.GetField, and then using DotNetObject.GetField on that returned instance to access its members.

Examples

The example below retrieves the static property System.DateTime.Now by casting **asm** to a DotNetObject, as well as casting the result of GetProperty to a DotNetObject from an **@*** type.

```
obj = (DotNetObject) ((DotNetObject) asm).GetProperty("System.DateTime.Now")
```

The example below gets the value of the Hour property and returns it in **itmp**.

```
obj.GetProperty("Hour", itmp)
```

The following example does the same thing as the previous example, except that it unboxes **int** on its way into **itmp**.

```
itmp = (int)GetProperty("Hour")
```

See “[Sample programs](#)” on [page 20-6](#) for examples in the context of a complete program.

DotNetObject.SetField – Set the value of a field on an instance

WIN

object.SetField(*fieldname*, *value*)

Arguments

object

An object returned by the DotNetAssembly constructor, the DotNetObject constructor, or some other property or method that returns an object.

fieldname

The name of the field to set.

value

The new value for the field.

Discussion

Fieldname must be a public or public static field defined for the specified object; otherwise, an \$ERR_MISSFLD error occurs. If a field is defined more than once using different value types, the default binder will be used to attempt to locate the best fit for the parameter passed. If no match is found, an \$ERR_MISSFLD error occurs.

If a value type is passed for the *value* argument, the system attempts to convert *value* from a Synergy/DE type to a .NET type using the same rules as for [DotNetObject.Call on page 20-13](#).

If an object is passed for the *value* argument, *value* must be either ^NULL (to indicate a null object) or an object returned by the DotNetObject constructor, DotNetObject.Call, DotNetObject.GetField, or DotNetObject.Get. Otherwise, an \$ERR_INVCAST error occurs.

If a structfield member is to be set, the object will be a structfield returned from DotNetObject.GetProperty or DotNetObject.GetField. Structs are boxed objects (copies), and once the members have been updated with DotNetObject.SetField, the struct must be copied back to the original field or property location with DotNetObject.SetProperty or DotNetObject.SetField using the boxed struct returned from the original DotNetObject.GetProperty or DotNetObject.GetField.

If *object* is an object returned from the DotNetAssembly constructor, only static fields can be accessed using the *namespace.class.field* form of *fieldname*.

Examples

The example below sets the value of an instance field.

```
obj.SetField("instanceField", "fieldValue")
```

The example below sets the value of a static field.

```
asmObj.SetField("namespace.classname.StaticField", "fieldValue")
```

DotNetObject.GetField – Query the value of a field

WIN

object.GetField(*fieldname*, *value*)

or

value = *object*.GetField(*fieldname*)

Arguments

object

An object returned by the DotNetAssembly constructor, the DotNetObject constructor, or some other property or method that returns an object

fieldname

The name of the field to query.

value

Returned with the coerced value for the field.

Discussion

If *object* is an object returned from the DotNetAssembly constructor, only static fields can be accessed using the *namespace.class.field* form of *fieldname*. *Fieldname* must be a public or public static field defined for the specified object; otherwise, an \$ERR_MISSFLD error occurs.

If a structfield member is to be accessed, *object* will be the object of the structfield returned by DotNetObject.GetField.

If a field is defined more than once using different value types, the default binder is used to attempt to locate the best fit for the parameter passed. If no match is found, an \$ERR_MISSFLD error will occur.

If the first GetField syntax above is used (returning the value in a writable argument), the system attempts to coerce parameters from .NET types using the same rules as [DotNetObject.GetProperty on page 20-17](#).

If the second GetField syntax is used, the value is coerced following the rules in the table on [page 20-14](#).

Examples

The example below queries the value of an instance field.

```
obj = obj.GetField("instanceField")
```

The example below queries the value of an instance field with no boxing.

```
obj.GetField("instanceField", obj)
```

The following example queries the value of a static field.

```
obj = asmObj.GetField("namespace.classname.StaticField")
```

DotNetObject property indexer set method

WIN

object[*index*] = *value*

Arguments

object

An object returned by the DotNetObject constructor (if the indexer is used on the class) or some other indexed field, property or method return type (for using the indexer on).

value

The value for the array member, which must be an object.

index

The value of the index to use, starting at 0.

Discussion

Calling the indexer set method is similar to setting a member of a single-stride array based on its index.

If a value type is to be set, you must box it first. The system then attempts to convert the passed type to the destination type for the object's indexer using the same rules as for [DotNetObject.Call](#) on [page 20-13](#). If the conversion fails, an \$ERR_INVCAST occurs.

If the indexer type is an object, *value* must be either ^NULL (to indicate a null object) or an object returned by the DotNetObject constructor, DotNetObject.Call, DotNetObject.GetField, DotNetObject.GetProperty, or the indexer get method. Otherwise, an \$ERR_INVCAST error will occur.

Index must be greater than or equal to 1 and less than or equal to the number of elements; otherwise, an \$ERR_SUBSCR error will occur.

See [“Properties and indexers” on page 8-8](#) for more information about indexers.

Examples

The example below is similar to setting the string in position 5 of an array to “hello”.

```
dnObject[5] = "hello"
```

DotNetObject property indexer get method

WIN

value = *object*[*index*]

Arguments

object

An object returned by the DotNetObject constructor (if the indexer is used on the class) or some other indexed field, property, or method return type on which to use the indexer.

value

Returned with the value of the field as System.Object.

index

The value of the index to use, starting at 0.

Discussion

Calling the indexer get method is similar to getting the value of a member of a single-stride array based on its index.

If the member contains a value instead of an object, the value will be returned following the rules in the table on [page 20-14](#).

Index must be greater than or equal to 1 or less than or equal to the number of elements; otherwise, an \$ERR_SUBSCR error will occur.

See “[Properties and indexers](#)” on [page 8-8](#) for more information about indexers.

Examples

The example below sets **str** to equal the string located at position 5 in dnObject (similar to an array).

```
str = dnObject[5]
```

DotNetDelegate – Create an instance of a delegate

WIN

```
delegate = new DotNetDelegate(assembly, type, object, method)
```

Arguments

delegate

The returned delegate instance (@DotNetDelegate).

assembly

The assembly that contains this delegate type.

type

The fully-qualified type name of the delegate, which includes the type's namespace but not its assembly name.

object

A reference to an object that owns the specified method. (*Object* is ^NULL if you don't want an instance method.)

method

The name of an instance method in *object*'s class.

Discussion

The DotNetDelegate constructor creates an instance of a delegate, which binds a Synergy Language instance or static method to a delegate defined in the .NET assembly.

If the .NET Framework throws an exception, a corresponding DotNetException will be thrown.

The search for *type* is case-sensitive. If the type cannot be found in the specified assembly or it is not public, a "Field/Type/Property/Event not found" error (\$ERR_MISSFLD) occurs. If *method* is not the name of an instance method within *object*'s class, an \$ERR_MISSMETH occurs.

When the *delegate* function is invoked, the arguments passed to it will correspond to the type definition for the delegate. The following conversions will occur:

- ▶ The .NET System.String and Char are converted to a Synergy string argument.
- ▶ System.Decimal, float, and double are converted to an implied-decimal argument.
- ▶ Boolean, Byte, Int16, Int32, Int64, SByte, UInt16, UInt32, or UInt64 are converted to an integer argument.
- ▶ An enumeration is treated like an **i4**, and an **i4** can be passed to a .NET enumeration.

- ▶ All other types are passed as a DotNetObject argument.
- ▶ If the parameter is defined as OUT for the delegate, the value of the argument is converted conversely upon return.

Examples

The example below creates a System.EventHandler delegate and stores it in an instance called button_OnClick. See [“Sample programs” on page 20-6](#) for an example in the context of a complete program.

```
button_OnClick = new DotNetDelegate(sys, "System.EventHandler", ^NULL,  
    & "ns1.cls1.static_meth1")
```

DotNetObject.AddDelegateToEvent – Associate a delegate with an event

WIN

object.AddDelegateToEvent (*eventname*, *delegate*)

Arguments

object

One of the following:

- ▶ An object returned by the DotNetObject constructor
- ▶ A method, property, or field that returns an object
- ▶ For a static event, an assembly

eventname

The name of the event to add the delegate to.

delegate

A delegate returned from the DotNetDelegate constructor.

Discussion

The order in which delegates are called by the .NET Framework is determined by the rules of the .NET Framework, which is usually first in, first out. A delegate should not contain any user interface components and should be implemented with the same limitations as *xfServerPlus* calls.

Examples

The second line of the example below associates an instance of the System.EventHandler delegate named `button_OnClick` with the Click event. See [“Sample programs” on page 20-6](#) for an example in the context of a complete program.

```
button_OnClick = new DotNetDelegate(sys, "System.EventHandler", ^NULL,  
    & "ns1.cls1.static_meth1")  
button.AddDelegateToEvent("Click", button_OnClick)
```

DotNetObject.RemoveDelegateFromEvent – Remove a delegate from an event

WIN

object.RemoveDelegateFromEvent (*eventname*, *delegate*)

Arguments

object

One of the following:

- ▶ An object returned by the DotNetObject constructor
- ▶ A method, property, or field that returns an object
- ▶ For a static event, an assembly, returned by the DotNetAssembly constructor

eventname

The name of the event from which the delegate is removed.

delegate

A delegate returned from the DotNetDelegate constructor.

Discussion

Delegates are automatically removed from all events when they are released if the reference count of the delegate goes to zero.

Examples

The last line of the example below removes an instance of the System.EventHandler delegate named button_OnClick from the Click event. See [“Sample programs” on page 20-6](#) for an example in the context of a complete program.

```
button_OnClick = new DotNetDelegate(sys, "System.EventHandler", ^NULL,  
    & "ns1.cls1.static_meth1")  
button.AddDelegateToEvent("Click", button_OnClick)  
.  
.  
.  
button.RemoveDelegateFromEvent("Click", button_OnClick)
```

21

Synergy Language C Interface

This chapter assumes you are familiar with both Synergy Language and C.

Introduction to the C Interface 21-3

Briefly describes the C interface and tells you what you need to include in your C routines.

Passing Arguments to C Routines from Synergy Language 21-5

Describes the data reference operations that you can use to pass arguments to non-Synergy Language routines.

C Routine Processing 21-6

Explains how C routines are referenced and processed.

Runtime Interface Functions 21-8

Contains specifications for the following runtime functions:

dbl_exec – Execute a routine within a Synergy Language program	21-11
dblerror – Issue a trappable Synergy Language error.....	21-12
dblfatal – Issue a nontrappable Synergy Language error	21-13
get_chn_sts – Get the status of a channel	21-14
get_errnum – Convert an error name to a Synergy Language error number.....	21-15
get_gbl_ptr – Get the address of a named global data section	21-16
get_seg_ptr – Get the address of a routine’s control segment	21-17
get_xarg – Get a subroutine argument’s descriptor.....	21-18
get_xarg_cstr – Get a subroutine argument as a null-terminated C string	21-19
get_xarg_fval – Get a subroutine argument as a floating-point value.....	21-20
get_xarg_val – Get a subroutine argument as an integer value	21-21
put_descr – Perform a Synergy Language store operation.....	21-22
put_xarg – Load data into a subroutine argument	21-23
put_xarg_fval – Load a floating-point value into an argument	21-24

put_xarg_str – Load a string into a subroutine argument 21-25

put_xarg_val – Load a value into an argument..... 21-26

Building a Dynamic Library with C Routines 21-27

Describes how to build a DLL or shared object that contains your C routines.

Introduction to the C Interface

WIN, UNIX

Synergy Language's C interface enables you to add user-designed C and Assembly routines to your Synergy application by building a Windows DLL or UNIX shared object that contains your C routines.

If you don't use C or Assembly subroutines in your Synergy Language applications, you don't need to read these instructions; just use the runtime provided with your Synergy Language distribution.



All data and routine access is accomplished with the routines described in this chapter. Any other routine or data access is not supported.

The runtime object code is distributed and can be linked at your site. Parts of the runtime are distributed as object modules; other parts are distributed as object libraries.

Synergy Language distributions have a default runtime that contains all Synergy Language system-supplied external subroutines. Any C routines that you include in your DLL or shared object can also access interface functions in the runtime. See [“Runtime Interface Functions” on page 21-8](#) for a list of the runtime interface functions you can access.

You must include the file **xcall.h** at the beginning of your C routine. For example:

```
#include "xcall.h"
```

The **xcall.h** file defines the following typedefs that are specified in the syntax in this chapter:

DESCRIP	Synergy Language data descriptor
SEENTRY	Synergy Language routine access structure
MEMENTRY	C routine access structure
GBLENTY	Global data access structure
DBL_CSTAT	Synergy Language channel status structure

Because **xcall.h** may change between releases, we recommend that you recompile and relink your C routines.

VMS

There is no Synergy Language C interface on OpenVMS. On OpenVMS, you can simply write a routine in C, compile it, link it into your program, and call it directly from your Synergy code without doing anything special.

The following routines can be called from a C language routine. They are described further in the **xcallv.h** file that came with your Synergy Language distribution:

dblerror	Issue a Synergy Language error.
dscr_to_cstr	Load a string into a field.
str_to_dscr	Get a subroutine argument as a null-terminated string.
dscr_to_int	Get a subroutine argument as a value.
dscr_to_dscr	Do a Synergy Language store operation.
int_to_dscr	Load an integer value into a field.

Passing Arguments to C Routines from Synergy Language

Synergy Language provides the following data reference operations for use when specifying a subroutine argument. ^DESCR is provided for consistency only, because passing a Synergy Language argument by descriptor is the default. The ^REF, ^VAL, and ^XTRNL operations are primarily used to pass arguments to non-Synergy Language subroutines, like C routines.

<code>^DESCR(variable)</code>	Passes an argument by descriptor. Synergy Language descriptors are memory blocks that contain information about the associated variable or literal.
<code>^REF(variable)</code>	Passes an argument by reference. ^REF passes a “pointer,” or virtual address, to <i>variable</i> ’s data as the argument.
<code>^VAL(expression)</code>	Passes an argument by value. ^VAL evaluates the specified expression, converts it to a binary representation, and then passes that value as an argument.

VMS

<code>^XTRNL(symbol)</code>	Passes the value associated with the specified symbol, which is resolved when you link the program.

When a non-Synergy Language routine passes an unsigned integer descriptor to a Synergy Language routine, the integer is treated as signed during the operation.

Examples

```
xcall sub(^REF(buffer))  
xcall sub(^VAL(d + c))
```

Since ^DESCR is the default calling mechanism, the following call:

```
xcall sub(^DESCR(variable))
```

is the same as

```
xcall sub(variable)
```

C Routine Processing

WIN, UNIX

A C routine is referenced as if it were declared as follows:

```
INT_REG c_routine(DESCRIP *argblk[])
```

where *argblk* is the controlling argument block. The return value depends on whether or not *c_routine* was declared as a ^VAL function in the calling routine:

- ▶ If *c_routine* was not declared as a ^VAL function in the calling routine, the return value is one if the routine completed successfully, or some other system-specific value if it did not complete successfully. (This value is the same as that obtained by the calling routine when the DBL\$XSTAT subroutine is used on OpenVMS.)
- ▶ If *c_routine* was declared as a ^VAL function in the calling routine, the return value is the actual result of the routine's processing.

INT_REG is defined in **machine.h** as the largest value on the machine that can hold a pointer.

Data is maintained within the Synergy Language environment by descriptor and is passed to and from routines using descriptor pointers within the argument block. If the C routine is called with a subroutine, or with a function reference that was declared as a ^VAL function in the calling routine, ((INT_REG)*argblk*[0]) is the actual number of arguments passed, *argblk*[1] points to the descriptor of the first actual argument, *argblk*[2] points to the descriptor of the second actual argument, and so forth. (For more information, see [^VAL on page 9-376](#).)

If the C routine is being called with a function reference that was not declared as a ^VAL function in the calling routine, ((INT_REG)*argblk*[0]) is one greater than the actual number of arguments passed, *argblk*[1] points to the descriptor of an automatically created return value field, *argblk*[2] points to the first actual argument's descriptor, *argblk*[3] points to the second actual argument's descriptor, and so on.

Note that the called routine is responsible for loading the return value field. If an *argblk* entry is null, the associated actual argument was not passed. If entry *n* within *argblk* was generated with a ^REF(*variable*), ((void *)*argblk*[*n*]) is a pointer to *variable*'s data. If entry *n* in *argblk* was created using a ^VAL(*expression*) function, ((INT_REG)*argblk*[*n*]) is the evaluated value of *expression*.

Many of the data types and storage mechanisms are incompatible between Synergy Language and C. In Synergy Language, for example, alpha data is not null-terminated, and space for an alpha function's return value is not allocated until data is stored.



For these reasons, among others, you should use the routines in the "Runtime Interface Functions" section exclusively whenever you're accessing Synergy Language data. Your data or memory could become corrupted if you don't.

The above warning doesn't apply to arguments that were generated using either a ^VAL or ^REF function. An *argblk* entry is a native CPU (void *) pointer for ^REF and a native CPU (long) integer for ^VAL. In such cases, it is your responsibility to access the argument block and any related data in a manner that is compatible with Synergy Language.

Be careful when a ^REF to an integer is passed. Unless the data has been aligned using the .ALIGN compiler directive, the referenced integer probably doesn't lie on a natural int boundary. This situation can cause access violations in some environments and considerably slower access in others.

For Synergy Language access of memory allocated in a C subroutine, either as variables or malloc, we recommend using ^M memory handles.



On UNIX, do not use `signal()` in your C code; use the newer `sigaction` routine. Use of `signal` will cause random and catastrophic errors at runtime.

Runtime Interface Functions

Your foreign subroutines can access the C string and data conversion functions, which are part of the Synergy runtime, listed in this section. You can only use these functions when you build a DLL or shared object with C routines (see [“Building a Dynamic Library with C Routines”](#) on page 21-27).

You must pass the argument block pointer to runtime interface functions. This argument block pointer gets passed to your C routine. More than one argument block pointer can be active. For example, the following call is valid (from Synergy Language):

```
xcall csub(%cfunc1, %cfunc2(%cfunc3))
```

Example using the Synergy Language C interface

The example below illustrates the following:

- ▶ A user-supplied C subroutine called from a Synergy Language program
- ▶ A Synergy Language subroutine called from a user-supplied C subroutine

```
;
; Calculate x to the yth power through the use of the system-supplied
; subroutine pow() in the C routine pwrkti.c.
;
```

```
.define TTCHN          ,1
record
    input               ,a30
    xval                ,d10
    yval                ,d10
    result              ,d10
    rlen                ,d2

proc
    open(TTCHN, o, "tt:")
    repeat
        begin
            display(TTCHN, "x: ")
            reads(TTCHN, input) [eof=done]
            xval = input
            display(TTCHN, "y: ")
            reads(TTCHN, input) [eof=done]
            yval = input
            xcall power(result, xval, yval)
            input = result [left:rlen]
            writes(TTCHN, "(DBL) result: " + input(1:rlen))
        end
```

```

done,
end

subroutine subl
    a_result      ,n

proc
    writes(TTCHN, "(C)    result: " + %string(a_result))
    return
endsubroutine

/* -----
*
* Routine:      XCALL POWER(RESULT, IVAL, POWR)
*
* Description:  Calculate X to the nth power, display the result
*               with a Synergy Language program, and return the
*               result to the calling routine.
*
* Arguments:    RESULT, D; the result with size defined in the
*               Synergy Language program limited to D10
*
*               IVAL,    D; Decimal value X
*               POWR,    D; Power value n
*
* Requirements: UNIX    - link with -lm (math library)
*
* Notes:
*
*   To identify this routine, like .IDENT, declare a variable
*   in the form:
*
*       static char *var = "@(#)ident";
*
*   The runtime executable will contain the ident string and can
*   be seen with the "what" command (e.g., what $DBLDIR/bin/dbr).
*
* ----- */

#include <math.h>
#include "xcall.h"

#define WROARG 6
static char *what = "@(#)POWER V2.0 (C-interface)";
extern double pow();

```

```
void
power(argblk)
DESCRIP **argblk;
{
    double num1, num2, result;
    SEENTRY *segp;                /* SEENTRY & DESC are
    DESCRIP *exec_argblk[2];      * defined in xcall.h */
    DESCRIP r_descr;

    if ((long)argblk[0] != 3)      /* Expecting 3 args */
        dblerror(WROARG);

    num1 = get_xarg_fval(argblk, 2); /* Get the x and n */
    num2 = get_xarg_fval(argblk, 3); /* arguments */
    result = pow(num1, num2);        /* Call pow function */
    put_xarg_fval(argblk, 1, result); /* Load return val */

    /* Set up a pointer to our return
    * argument pairs to Synergy sub1*/

    exec_argblk[0] = (DESCRIP *)1; /* Define one argument*/
    exec_argblk[1] = get_xarg(argblk, 1, &r_descr);
    dbl_exec(get_seg_ptr("SUB1"), exec_argblk);
    /* Call Synergy subroutine */

    return;
}
```

dbl_exec – Execute a routine within a Synergy Language program

```
INT_REG dbl_exec (SEGENTRY *segment, DESCRIP **argblock)
```

Arguments

segment

A pointer to a segment entry.

argblock

The argument block pointer paired by the Synergy Language subroutine to the C function.

Discussion

The `dbl_exec` function executes a Synergy Language routine.

This function provides a bidirectional calling mechanism between C and Synergy Language routines. In particular, it enables a C routine to call any routine within a Synergy Language program (regardless of whether the routine is written in C or Synergy Language).

The return value is the same as the value that would be returned by the `XSTAT` subroutine.

It is the calling routine's responsibility to ensure that the argument block is valid.

dblerror – Issue a trappable Synergy Language error

```
void dblerror(int error)  
void dblerror2(int error, char *string)  
void dblerror3(int error, char *string1, char *string2)
```

Arguments

error

The number of the Synergy Language error to generate.

string

string1

string2

An optional support string for the error.

Discussion

The `dblerror` function issues a trappable Synergy Language error to the Synergy Language subroutine that was most recently active.

The support string is only accessed if the error message associated with *error* requires it. No return is made from this routine. An error is generated in the most recently active routine.

Errors are defined by mnemonic, not number. Use the `get_errnum` function to get the error number to be passed. The “[Error Messages](#)” chapter of *Synergy Language Tools* contains a complete list of Synergy runtime errors.

VMS

The syntax for `dblerror` is

```
void dblerror(int, char*, char*)
```

Examples

The following example generates an `$ERR_ARGSIZ` error.

```
dblerror(get_errnum("ARGSIZ", 6))
```

dblfatal – Issue a nontrappable Synergy Language error

The dblfatal function is no longer available. Use dblerror. (See [dblerror](#) on page 21-12.)

get_chn_sts – Get the status of a channel

```
int get_chn_sts(int channel, DBL_CSTAT *structure, char *filename)
```

Arguments

channel

The Synergy Language channel to reference.

structure

A pointer to a DBL_CSTAT structure to load.

filename

A pointer to a buffer in which to load the channel's associated filename as a null-terminated string.

Discussion

The `get_chn_sts` function gets information about a Synergy Language channel.

The return value is a copy of the `cs_flags` entry, which defines the state of the channel. See the **xcall.h** file for more information about `cs_flags`.

If the specified channel is invalid or if it is not currently open, the returned flag is zero. If *structure* is null, the return value is still valid, but no structure is loaded.

If *filename* is null, no filename is copied. If *filename* is not null, the routine assumes that the buffer is large enough to contain the filename and should be up to 255 bytes in length.



The information returned by this routine is provided as a convenience to you. It is only a copy of a subset of the data maintained by the Synergy runtime for the referenced channel. The runtime assumes that it is the only manipulator of Synergy Language channels, and any I/O to or from a file that is “unknown” to the runtime could cause data corruption.

get_errnum – Convert an error name to a Synergy Language error number

```
int get_errnum(char *error, int length)
```

Arguments

error

A pointer to a string identifying the error mnemonic.

length

The length of the string.

Discussion

The `get_errnum` function gets the Synergy Language error number for an error mnemonic.

The return value is the error number associated with the mnemonic.

Because the error number can change from environment to environment, we strongly recommend that you always reference errors by their mnemonics (the text that follows the “\$ERR_” in a runtime error literal).

get_gbl_ptr – Get the address of a named global data section

```
GBENTRY *get_gbl_ptr(char *global)
```

Arguments

global

A pointer to a null-terminated string that defines the named global data to find.

Discussion

The `get_gbl_ptr` function gets the address of a named global data section.

The return value is a pointer to the controlling `GBENTRY` structure for the data. If `get_gbl_ptr` can't find the named data, the return value is null.

A `GBENTRY` structure exists for each global data section name and for each global common variable that is declared in the executing program.

get_seg_ptr – Get the address of a routine’s control segment

```
SEENTRY *get_seg_ptr(char *routine)
```

Arguments

routine

A pointer to a null-terminated string that defines the name of the routine.

Discussion

The `get_seg_ptr` function gets the address of a routine’s control segment.

The return value points to the control segment for the routine. If `get_seg_ptr` can’t find the named routine the return value is null.

Each routine that can be accessed from Synergy Language with a subroutine or function reference has an associated entry within the segment tables. If the routine is written in Synergy Language, the `SE_DBL` flag within the flags field is set and the entry is a `SEENTRY` structure. If the `SE_DBL` flag is not set, the routine is assumed to be written in C and the entry is laid out according to a `MENTRY` structure. (See the `xcall.h` file for more details.)

get_xarg – Get a subroutine argument’s descriptor

```
DESCRIP *get_xarg(DESCRIP **argblock, int arg, DESCRIP *descriptor)
```

Arguments

argblock

The argument block pointer passed by the Synergy Language subroutine to the C function.

arg

The number of the argument to reference.

descriptor

A pointer to a descriptor to load.

Discussion

The `get_xarg` function makes a copy of a subroutine argument’s descriptor.

If the argument was passed, the return value is a pointer to the *descriptor*. If the argument was not passed, the returned pointer is null.

get_xarg_cstr – Get a subroutine argument as a null-terminated C string

```
int get_xarg_cstr(DESCRIP **argblock, int arg, char *buffer,
                 int length, int strip_flag)
```

Arguments

argblock

The argument block pointer passed by the Synergy Language subroutine to the C function.

arg

The number of the argument to reference.

buffer

The pointer to the buffer in which to load the string.

length

The length of the buffer in which to load the string.

strip_flag

A flag that indicates if trailing blanks are to be stripped from the string.

Discussion

The `get_xarg_cstr` function gets a subroutine argument as a null-terminated C string.

Strip_flag is true if trailing blanks are to be stripped. The return value is the length of the resulting string. The length can be zero.

If the argument wasn't passed, the returned length is zero.

If the size of the resulting string is greater than or equal to *length*, only *length* characters are transferred and the string isn't null-terminated.

If *buffer* is null, no data is transferred, but the string's size is returned.

We've provided this routine for those situations in which a Synergy Language alpha field must be accessed as a C, null-terminated string. In general, we do *not* recommend that you manipulate Synergy Language fields as null-terminated strings, which requires the caller to load and transform a copy of a Synergy Language field into a local buffer, manipulate the resulting null-terminated string in a way that is not consistent with Synergy Language data, and then reload the string back into the Synergy Language field. It is much more efficient to manipulate the data directly in its native Synergy Language form.

VMS

This function's functionality is available as `dscr_to_cstr`. See the `xcallv.h` file for details.

get_xarg_fval – Get a subroutine argument as a floating-point value

```
double get_xarg_fval (DESCRIP **argblock, int arg)
```

Arguments

argblock

The argument block pointer passed by the Synergy Language subroutine to the C function.

arg

The number of the argument to reference.

Discussion

The `get_xarg_fval` function gets a subroutine argument and converts it into a floating-point value of the native C “double” type.

The return value is the referenced argument evaluated as a double. If the argument wasn’t passed, the returned value is zero. If the argument is alpha, it is treated as if it were decimal.

get_xarg_val – Get a subroutine argument as an integer value

```
INT_REG get_xarg_val (DESCRIP **argblock, int arg)
```

Arguments

argblock

The argument block pointer passed by the Synergy Language subroutine to the C function.

arg

The number of the argument to reference.

Discussion

The `get_xarg_val` function gets a subroutine argument as an integer value.

The return value is the referenced argument evaluated as an integer. If the argument wasn't passed, the returned value is zero. If the argument is alpha, it is treated as if it were decimal. If the argument is implied-decimal, it is rounded or truncated according to the Synergy Language subroutine that was most recently called.

You must use the `get_xarg_val` function whenever the value of a Synergy Language numeric field is required, even when the field is an integer. (Remember, integer data is stored in Synergy Language fields in a variable-length, unaligned form, and you may not be able to access it directly.)

VMS

The functionality of this function is available as `dscr_to_int`. See the **xcallv.h** file for details. The prototype is

```
INT_REG dscr_to_int (DESCRIP*)
```

put_descr – Perform a Synergy Language store operation

```
void put_descr (DESCRIP *destination,  DESCRIP *source)
```

Arguments

destination

A pointer to a Synergy Language destination field descriptor.

source

A pointer to a Synergy Language source field descriptor.

Discussion

The put_descr function does a Synergy Language store operation.

It performs a store operation exactly as if the following Synergy Language statement:

```
DSTFLD = SRCFLD
```

were present in a Synergy Language program, with *source* pointing to the descriptor for SRCFLD and *destination* pointing to the descriptor for DSTFLD. All of the required conversions, truncations, alignments, and so forth occur automatically, according to the data types of the two fields.

VMS

The functionality of this function is available as dscr_to_dscr on. See the **xcallv.h** file for details.

put_xarg – Load data into a subroutine argument

```
void put_xarg(DESCRIP **argblock, int arg, DESCRIP *source)
```

Arguments

argblock

The argument block pointer passed by the Synergy Language subroutine to the C function.

arg

The number of the argument into which to store data.

source

Points to a descriptor for data to be transferred.

Discussion

The put_xarg function loads data into a subroutine argument.

If the argument wasn't passed, or if it's a literal, this routine generates an error.

The put_xarg operation performs a store operation exactly as if the following Synergy Language statement:

```
DSTARG = SRCFLD
```

were present in a Synergy Language program, with *source* pointing to the descriptor for SRCFLD and *argblock[argument]* pointing to the descriptor for DSTARG. All of the required conversions, truncations, alignments, and so forth occur automatically according to the data types of the two fields.

put_xarg_fval – Load a floating-point value into an argument

```
void put_xarg_fval(DESCRIP **argblock, int arg, double value)
```

Arguments

argblock

The argument block pointer passed by the Synergy Language subroutine to the C function.

arg

The number of the argument to reference.

value

The value to be loaded.

Discussion

The `put_xarg_fval` function converts and stores a floating-point value into an argument.

If the argument wasn't passed, or if it's a literal, the routine generates a trappable Synergy Language error.

put_xarg_str – Load a string into a subroutine argument

```
void put_xarg_str(DESCRIP **argblock, int arg, char *string, int length)
```

Arguments

argblock

The argument block pointer passed by the Synergy Language subroutine to the C function.

arg

The number of the argument to reference.

string

A pointer to the string to transfer.

length

The length of the string to transfer.

Discussion

The put_xarg_str function loads a string into a subroutine argument.

If the argument wasn't passed, or if it's a literal, the routine generates a trappable Synergy Language error.

The argument's field is loaded left-justified over blanks.

VMS

The functionality of this function is available as str_to_dscr. See the **xcallv.h** file for details. Its prototype is

```
void str_to_dscr(DESCRIP*, char *string, int length)
```

put_xarg_val – Load a value into an argument

```
void put_xarg_val(DESCRIP **argblock, int arg, INT_REG value)
```

Arguments

argblock

The argument block pointer passed by the Synergy Language subroutine to the C function.

arg

The number of the argument to reference.

value

The value to be loaded.

Discussion

The `put_xarg_val` function loads a value into an argument.

If the argument wasn't passed, or if it's a literal, the routine generates a trappable Synergy Language error.

The value is stored according to the rules for moving integer data to the data type of the destination field. (See [“Assignment Statements” on page 3-26](#) for details about these rules.)

You must use the `put_xarg_val` routine whenever a Synergy Language numeric field is to be loaded with a value, even when the field is an integer. (Remember, integer data is stored in Synergy Language fields in an unaligned form, and you may not be able to access it directly.)

VMS

The functionality of this function is available as `int_to_dscr`. See the `xcallv.h` file for details. Its prototype is

```
void int_to_dscr(DESCRP*, INT_REG value)
```

Building a Dynamic Library with C Routines

Before you build a DLL or shared object that contains your C routines, you must first initialize the runtime C interface using the optional second argument to the `%DLL_OPEN` routine. Follow the instructions below:

1. (Windows only) Add the following export command to the EXPORTS section of the `.def` file that you use to build your DLL:

```
xclapi_setup_ptr
```

2. Link the file `xclapi.obj` (or `xclapi.o` on UNIX) into your shared object. If you are using the floating point routines (`get_xarg_fval` and `put_xarg_fval`), you must also link in `xclapif.obj` (or `xclapif.o` on UNIX).

3. Add a new DLL handle to your common global area. For example:

```
dllhand, D_ADDR
```

4. Modify your programs to open your DLL by calling `%DLL_OPEN`, making sure to specify the optional C interface flag as the second argument. (See [%DLL_OPEN on page 11-10](#).) For example:

```
dllhand = %DLL_OPEN(mydll, 1)
```

5. Either modify your programs to use `%DLL_CALL` instead of `XCALL`, or use a `.DEFINE` macro. (See [%DLL_CALL on page 11-4](#) for details about `%DLL_CALL` and [“Parameterized macros” on page 5-8](#) for more information about `.DEFINE` macros.)

For example, if your code contained

```
xcall ttqis(row, col, "text")
```

and `ttqis` was a routine you linked into the Synergy runtime, you could add the following to your standard include files:

```
.define ttqis(a,b,c) dll_call(dllhand, DLL_TYPE_DBLCALL, "ttqis", a,b,c)
```

If you have a variable number of arguments, always specify the highest number of arguments in the `.DEFINE` statement. The maximum number of arguments is 20.

Appendices

Appendix A: Maximums

Provides a list of maximum sizes and maximum and minimum values for Synergy Language.

Appendix B: ASCII Character Set

Provides a table of the ASCII characters and their corresponding decimal codes and relations.

Appendix C: Terminal Keys with Special Functions

Specifies the control key combinations that Synergy Language supports on each operating system.

A

Maximums

Item	Usage	Maximum size	OpenVMS (if different)	Minimum value	Maximum value
Filename	OPEN, ISINFO	a255 (512 on Windows)	a255		
ISAM file options	ISINFO FOPT	a64			
ISAM key type	ISINFO KTYPE	a1			
ISAM null value type	ISINFO NULL	a11			
ISAM null value	ISINFO NULLVALUE	a1 or a1024	a1		
ISAM key name	ISINFO, ISAMC KEYNAME	a15	a32		
ISAM key specification	ISINFO STRING	a1500			
ISAM key length	ISINFO KEYLENGTH	d3 or i2		1	255
ISAM key size	READ	a251 or a254	a255		
ISAM record length	READ	a65534	a32234		
Number of ISAM keys	ISINFO, ISAMC NUMKEYS			1	255
RFA	READ	a6		a6	a6
SEND/RECV message	SEND/RECV	4096 on UNIX (configurable through synd)	a16383		
SEND/RECV ID	SEND/RECV	a6	a39		
Variable name size		30	31		
Common variable name		30	31		
Label		30	31		
Channel numbers		d4 or i2		1	1024
Alpha variable		a65535			

Maximums

Item	Usage	Maximum size	OpenVMS (if different)	Minimum value	Maximum value
Integer variable		i1 or i2 or i4 or i8			
Decimal variable		d28			
Implied-decimal variable		d28.28			
Packed variable		p18			
Implied-packed variable		p28.10			
i1 (signed)		d3		-128	127
Unsigned i1	%UNSIGNED	d3		0	255
i2 (signed)		d5		-32768	32767
Unsigned i2	%UNSIGNED	d5		0	65535
i4 (signed)		d10		-2147483648	2147483647
Unsigned i4	%UNSIGNED	d10		0	4294967295
i8 (signed)		d19 (returned as i8)		-9223372036 854775808	9223372036 854775807
Unsigned i8	%UNSIGNED BIGNUM on negative i8	d19 (returned as i8)		0	9223372036 854775807
Program source physical line length		255			
Program source logical line length		8192			
Modules in an ELB		4096			
Open ELBs		256			
ELB specification length		32			

B

ASCII Character Set

In alpha relational comparisons and ISAM key comparisons, character strings are compared from left to right based on the ASCII collating sequence. The ASCII characters are arranged so that the sequence proceeds from the lowest decimal code to the highest. Therefore, an “A” is considered less than a “J” but greater than a “3.” Similarly, the string “abcde” is less than “abcje” but greater than “abcTe.” In alpha comparisons of different sized operands, the comparison is made for the length of the shorter operand. Thus, “abcXYZ” is equal to “abc” because the first three letters of “abcXYZ” match “abc.”

When decimal operands are processed, only the low-order 4 bits of each digit are used to determine the effective value of the operand. Synergy Language assumes that valid ASCII digits (0 through 9) are present. Negative decimal values are stored the same way as positive values, except that bit 6 (7 being the high-order bit in each 8-bit character) is turned on to indicate a negative value. Thus, if a decimal field is examined as an alpha value, negative values show up with the low-order digit position consisting of a character from “p” through “y.” The following table indicates the negative value to which each of these characters corresponds.

The following table presents the ASCII characters and their corresponding decimal codes.

ASCII Character Set

Decimal value	Character	Decimal value	Character	Decimal value	Character	Decimal value	Character
000	NUL (CTRL+@)	032	Space	064	@	096	`
001	SOH (CTRL+A)	033	!	065	A	097	a
002	STX (CTRL+B)	034	"	066	B	098	b
003	ETX (CTRL+C)	035	#	067	C	099	c
004	EOT (CTRL+D)	036	\$	068	D	100	d
005	ENQ (CTRL+E)	037	%	069	E	101	e
006	ACK (CTRL+F)	038	&	070	F	102	f
007	BEL (CTRL+G)	039	'	071	G	103	g
008	BS (CTRL+H)	040	(	072	H	104	h
009	HT Horiz Tab	041	)	073	I	105	i
010	LF Line Feed	042	*	074	J	106	j
011	VT Vert Tab	043	+	075	K	107	k
012	FF Form Feed	044	,	076	L	108	l
013	CR (RETURN)	045	-	077	M	109	m
014	SO (CTRL+N)	046	.	078	N	110	n
015	SI (CTRL+O)	047	/	079	O	111	o
016	DLE (CTRL+P)	048	0	080	P	112	p -0
017	DC1 (CTRL+Q)	049	1	081	Q	113	q -1
018	DC2 (CTRL+R)	050	2	082	R	114	r -2
019	DC3 (CTRL+S)	051	3	083	S	115	s -3
020	DC4 (CTRL+T)	052	4	084	T	116	t -4
021	NAK (CTRL+U)	053	5	085	U	117	u -5
022	SYN (CTRL+V)	054	6	086	V	118	v -6
023	ETB (CTRL+W)	055	7	087	W	119	w -7
024	CAN (CTRL+X)	056	8	088	X	120	x -8
025	EM (CTRL+Y)	057	9	089	Y	121	y -9
026	SUB (CTRL+Z)	058	:	090	Z	122	z
027	ESC Altmode	059	;	091	[	123	{
028	FS (CTRL+ \)	060	<	092	\	124	
029	GS (CTRL+])	061	=	093	]	125	}
030	RS (CTRL+ ^)	062	>	094	^	126	~
031	US (CTRL+ _)	063	?	095	_	127	DEL Rubout

C

Terminal Keys with Special Functions

Certain characters and control key combinations have special functions that Synergy Language supports as input. The following table lists and describes the control key combinations that Synergy Language supports on the various operating systems.

The UNIX control key combinations listed in the table are the most common default settings, although you can change any one of them using the `stty` command. (For example, most Synergy Language programs use `CTRL+C` as interrupt and `DELETE/RUBOUT` as `BACKSPACE`.) Refer to [“Terminal Settings Used by Synergy Language”](#) in the “Terminal Characteristics on UNIX” chapter of your *Professional Series Portability Guide* or your UNIX system commands manual for more information.

Function	Windows	UNIX	OpenVMS
Resume terminal output at the point that <code>CTRL+S</code> suspended it. Does not echo to the terminal.	N/A	<code>CTRL+Q</code>	<code>CTRL+Q</code>
Suspend terminal output until a subsequent <code>CTRL+Q</code> is used to resume it. <code>CTRL+S</code> enables you to temporarily stop information from scrolling off the top of the CRT screen. Does not echo to the terminal.	N/A	<code>CTRL+S</code>	<code>CTRL+S</code>
Interrupt program execution, unless the program uses the <code>FLAGS</code> subroutine to ignore this signal. If it's not ignored, a single key terminates a Synergy Language program that is not running in debug mode. If the program is running in debug mode, this key returns control to the debugger. This key typed as input to the debugger terminates the program. Echoes to the terminal as <code>CTRL+C</code> on OpenVMS. Does not echo on UNIX.	<code>CTRL+C</code>	<code>DELETE</code> or <code>RUBOUT</code>	<code>CTRL+C</code>
Quit program execution. System option #16 maps this key to the interrupt key.	N/A	<code>CTRL+\</code>	<code>CTRL+Y</code>
Toggle suspension of terminal output.	N/A	N/A	<code>CTRL+O</code>

Terminal Keys with Special Functions

Function	Windows	UNIX	OpenVMS
Discard the current line of input. You can use the FLAGS subroutine flag 4 to specify how kill-character echoing is to be handled. Echoes to the terminal as CTRL+U or CTRL+X.	CTRL+U or CTRL+X	CTRL+U or CTRL+X	CTRL+U or CTRL+X also flushes type ahead
Discard the last character typed on the current, unfinished line of input. You can use the FLAGS subroutine flag 4 to specify how character deletions will be indicated to the user.	DELETE or RUBOUT	CTRL+H or BACKSPACE	DELETE, RUBOUT, or BACKSPACE
Logically signal an “End of file” condition for terminal input. Echoes to the terminal as the EOF character on UNIX and as CTRL+Z on OpenVMS.	CTRL+Z	CTRL+D	CTRL+Z
Display process status.	N/A	N/A	CTRL+T

Glossary

A

absolute ranging	A form of ranging that indicates the starting and ending positions of a character-sequence range using two arithmetic expressions separated by a comma.
absolute value	The value of a number without reference to the positive or negative sign.
abstract class	A class that describes a contract of basic functionality that must be inherited and implemented to be used. An abstract class cannot be instantiated.
abstract method	A method that has a signature but does not have an implementation. An abstract method is, by default, a virtual method. Abstract methods can only be defined in an abstract class.
accelerator key	An accelerator key is one that becomes mapped to an application-specific command message when it is pressed. In UI Toolkit terms, an accelerator key is a key that is mapped to a Toolkit menu entry or an equivalent signal. Third-party ActiveX controls can also map keys as accelerators, which means that the keys trigger special events when pressed.
access	To locate and obtain data.
access method	A method of retrieving pieces of data (records) from a larger set of information (file).
access mode	The method (input, output, or update) by which data from a specified file or device is accessed.
accessibility	The degree to which a class or its members can be accessed. Accessibility is primarily controlled by the PUBLIC, PROTECTED, and PRIVATE access modifiers.
accessor	The get and set methods of a property.

activation	A specific instance of a routine.
activation character	A nonprinting character that affects READS input from a channel opened in terminal mode. The READS operation ends when either a standard record terminator (LF, VT, FF, ESC) or an activation character is encountered. Additional characters are defined by the ACCHR subroutine.
activation parameter	A parameter that is presented internally to a routine.
active routine	A routine that has been entered but not exited.
ActiveX	A set of technologies developed by Microsoft that provides the ability to develop active, executable objects for Windows applications and web sites.
ActiveX automation	A way of integrating development tools and applications by enabling an application to make its functionality accessible to other applications or to control the exposed functionality of other applications.
ActiveX container	See container .
ActiveX control	An active, executable object that implements the ActiveX technologies and that enables you to provide interactive, multimedia effects in your application.
actual argument	An argument that is listed in a subroutine or function call.
alpha	All ASCII characters except the nonprintable characters that control I/O devices (such as carriage return, line feed, and form feed).
alpha data	Any consecutive sequence of ASCII characters that can be treated as a unit of information.
alpha expression	An expression whose result is an alpha value.
alphabetic	The uppercase characters “A” through “Z” and the lowercase characters “a” through “z.”
AlphaDesc	An alpha descriptor (for example, a20). AlphaDesc extends VariantDesc by adding alpha-specific functionality.
alphanumeric	Both numeric and alphabetic characters.
argument	A variable or routine parameter passed to another routine via a subroutine, function, or method.

arithmetic operator	An operator that performs addition, subtraction, multiplication, or division.
array	A group of fields, with each field the same defined size and data type.
assembly	A collection of all the files required to make a .NET application run. An assembly is what results when you compile your Synergy .NET source files, and it can have the extension .exe or .dll .
assignment operator	The equal sign, because it assigns one operand to another.
attribute	A way of providing descriptive or behavioral information about your code. In Synergy .NET, attributes precede the item they are decorating and are enclosed in curly brackets ({}). All attributes are classes that inherit from System.Attribute.
B	
base class	See parent class .
binary operator	An operator that acts on two operands. See operator .
binding	A method of grouping multiple main routines into one executable program.
bit	The smallest unit of data that can be accessed.
bitwise operator	An operator that performs Boolean operations on the bits within integer operands.
blank	A space.
block	A 512-byte unit of information that can be read or written one or more units at a time.
blocking factor	The number of blocks to be assigned to a file I/O buffer.
Boolean operator	A logical operator (OR, exclusive OR, AND, and NOT).
border	The outline of a window's display area.
boxing	The encapsulation of a descriptor type or a value type into an object type so it can be used like an object.
bracketed array	See real array .

breakpoint	A debugger feature that interrupts program execution in debug mode and returns to the debugger prompt.
buffer	A storage area that temporarily holds data.
byte	Eight bits.
C	
C Interface	A component of Synergy Language that enables Synergy Language programs to access C language routines and vice versa.
cache	Preprocessed data of limited lifetime to provide quicker access.
cascaded IF statement	A series of IF-THEN-ELSE statements that have another IF-THEN-ELSE statement following the ELSE keyword.
casting function	A function that enables data to be accessed as any valid data type, regardless of the actual data type with which it was declared.
certificate	An electronic ID card that establishes credentials or confirms identity when doing business or other transactions on the Web. You must have a digital certificate from a certificate authority to send encrypted messages.
certificate authority (CA)	A company that digitally creates and issues certificates. Examples are VeriSign and Thawte.
chain	To pass control to another main routine. Chaining is performed by the STOP statement.
channel	A conduit for data transmission between a file or device and your program.
child class	See derived class .
child process	A subprocess that is created and executed by the parent (current) process. Once the child process is completed, it returns control to the parent process.
cipher	Any method of encrypting text.
class	A named set of related data and functionality that works on that data.
class member	A general term for any item that can be declared within a class declaration, such as a method, a field, a property, another nested class, and so forth.

class structure	A structure that is defined inside a class.
closed routine	A self-contained routine that cannot be run independently from a main calling routine.
CLR	Common language runtime, the runtime environment for .NET Framework.
collection	An ordered group of objects, which are referred to as the collection's <i>elements</i> .
comment	A sequence of characters in your code that clarifies what your program is doing and is ignored by the compiler. A Synergy Language comment begins with a semicolon (;).
common data	Data that is shared between routines but that doesn't have to be passed as an argument.
compilable text	Any characters in your code that the compiler doesn't ignore.
compilation control directive	A type of compiler directive that instructs the compiler. It can precede or follow any logical line, and it applies to the remainder of the source file. See also structure control directive .
compile-time expression	An expression that can be completely evaluated at compile time. Dimensions, indexes, and range specifications are not allowed in compile-time expressions.
compiler	The component of Synergy Language that translates your source files containing Synergy Language statements into object files containing system-level information.
compiler directive	A statement that instructs the Synergy compiler and is evaluated only when a program is compiled. Compiler directives are not included in the resulting executable program.
compiler listing	A display of your program that is generated when you compile. The compiler listing may or may not include one or more compiler listing tables, depending on which compiler options you set.
compiler listing table	A table that is generated in the compiler listing at the end of each routine if you've set the appropriate compiler options.
compiler option	A switch on the compile command that tells the compiler how to behave or interpret data.
complex array	See real array .

compound statement	A single logical unit that consists of a BEGIN statement, an END statement, and all the Synergy Language statements that come between them.
condition	A set of circumstances that is either true or false.
conditional	An expression whose value is interpreted as a true or false value.
conditional block	A group of statements that begin with the .IFDEF or .IFNDEF directive, end with the .ENDC directive, and may contain the .ELSE directive. A conditional block specifies that compilation depends on either the declaration or nondeclaration of a variable or replacement identifier.
constant	A value that doesn't change while the program is being executed, such as a literal, a literal field, or a .DEFINE identifier. Also called a literal.
constructor	A class member function that is called when an instance of the class is created.
container	A window that can contain one or more controls or subcontainers. An ActiveX control container can contain one or more ActiveX controls. A Toolkit ActiveX container window is a Synergy window that can contain one or more ActiveX control containers. "Contain" means the containee is visually placed completely within the container, and the destruction of the container automatically causes the destruction of the containee.
continuation line	The second and all subsequent physical lines for a single logical line. A continuation line begins with an ampersand (&).
control statement	A Synergy Language statement that changes the order of execution by transferring program control to another statement, transferring control to another program or subroutine, ending processing of the current program, or returning control to a referencing routine.
CRC	Cyclic redundancy check, or an integer derived from a record in order to detect changes. Any change to the data generates a different CRC.
current record	The last record that was read by READ or READS or positioned to by FIND on an ISAM channel. There is no current record after a DELETE or WRITE.

D

daemon	A program running on an operating system that is not invoked explicitly but that sits dormant, waiting for a specific condition to occur.
data division	The part of a Synergy Language program in which you define the data structures that will be used in the program.
data manipulation statement	A Synergy Language statement that directly changes your data.
data reference operation	Either a casting function or a mechanism that obtains various data characteristics.
data type	Data representation such as alpha, decimal, implied-decimal, integer, String, <i>class</i> , <i>structure</i> , etc.
debugger	A productivity tool that enables programs to run in a special debugging mode so you can test, detect, locate, and remove any mistakes.
decimal data	A signed whole number that consists of any of the characters 0 through 9.
decimal expression	An expression whose result is a decimal value.
DecimalDesc	A decimal descriptor (for example, d10). DecimalDesc extends VariantDesc by adding decimal-specific functionality.
declarative statement	A statement that appears in the routine's data division and defines a variable that the routine can access.
default value	The value of a field before any procedure division statements are executed, where the field does not have an initial value.
delegate	A type-safe function pointer used by the .NET Framework. It specifies a method to call and optionally an object to call the method on.
delimiter	A character that separates adjacent language elements in a statement.
derived class	A class that extends another class, inheriting all of its members. A derived class is a child of its immediate parent.
descriptor	A memory block that contains information about the associated variable or literal (such as the size, type, and position of the data) and the default way of passing arguments to external subroutines.

destructive receive	When the RECV statement receives and deletes a message from the message manager.
destructor	A class member function that is called before an instance of the class is destroyed.
device	A piece of equipment within or attached to a computer.
dimension	The defined size of an array. For example, the dimension of an array with 2 rows and 3 columns is (2,3), or 2 by 3.
display area	A subset of the total area of a window that is physically displayed on the screen.
display attribute	A characteristic of a window or an associated element that affects its appearance, such as color, boldface, blinking, reverse video, and underline.
dynamic array	An array whose upper bounds are determined at runtime. You can specify multiple levels in a dynamic array by declaring a bracket pair for each level.
E	
ELB	An abbreviation for executable library. See executable library .
encapsulate	To contain and control the access to functionality or data within a class.
end-of-file character	A control character that marks the physical end of file.
enumeration	An enumeration is a data type that consists of a set of named values. You must list (enumerate) each of the values when you create the enumeration. For example, you could create an enumeration called color that consists of the values red , orange , yellow , green , blue , indigo , and violet .
environment variable	An abbreviation defined at the operating system level for a device, directory, or path name. Also called a logical assignment.
error literal	A constant that represents a runtime error. Error literals have the format \$ERR_ <i>string</i> , where <i>string</i> is an error identification mnemonic.

error trapping	The process of “catching” problems so that you can transfer control to another part of the program that will handle the error instead of letting your program die. Errors that aren’t trapped are considered fatal.
event	An action that occurs as a result of another action (for example, a mouse click).
event handler	A method that processes an event.
executable file	A file that can be executed or run. An executable file is normally composed of one or more object files linked together.
executable library	A group of executable subroutines and functions in a single file created by the Synergy linker.
expression	A combination of data elements (operands) and operators that produce a value. An expression must have one or more operands and zero or more operators.
extend	To inherit nonstatic methods, fields, and properties and nested classes from the parent class, as well as any members that the parent inherits from its parent, in recursive fashion.
external function	A user-defined or system-supplied nonintrinsic function that was defined with the FUNCTION statement. External functions must be declared with the EXTERNAL FUNCTION statement before they can be used.
external subroutine	A user-defined or system-supplied routine that begins with the SUBROUTINE statement and is executed from another routine with the XCALL statement.
external subroutine library	A collection of external subroutines supplied by Synergy Language as a standard feature of the language.
F	
fatal error	A problem that results in immediate, unconditional termination of your program, unless trapped by the FATAL subroutine.
field	A record component that contains an individual data element. It stores a specific item of data, such as a customer number, first and last name, telephone number, and so forth.
field position indicator	An argument that designates a field should begin at a specific point, or offset, in the record.

field set	A set of row, column, and length “fields” within a Synergy window.
file manipulation statement	A statement that accesses records within a file or group of files.
file type	Whether a file is ISAM, relative, sequential, or stream.
fill characters	A pattern of characters to be loaded into a variable.
filler	An unnamed field that merely occupies record space.
fixed-length record	A record that always has the same number of characters.
flag	An indicator of some special condition or that tells a later part of the program of a condition that occurred earlier.
formal parameter	A parameter that is declared in a subroutine or function.
fractional precision	The number of decimal places in an implied-decimal value.
function	A process that returns a value and can be called explicitly or referenced implicitly.

G

GAC	The .NET global assembly cache.
global data	Data that is stored in a named data area rather than the local data area and that can be accessed by any routine in a program.
global data section	A uniquely named data area that contains shared records.
global structure	A structure that is defined outside a routine, class, or namespace declaration.
GRFA	An abbreviation for Global Record File Address (or global RFA), which is stored as a 10-byte variable that includes a 4-byte record CRC and a 6-byte RFA. See also RFA and CRC .
group	A structure within a structure, as defined by the GROUP statement. Fields or other group declarations can be members of a group.

H

hardware scrolling	A method of moving data on a terminal screen that uses the capabilities of the terminal to do the scrolling instead of repainting all scrolled characters.
high-order bit	The bit that has the highest, unsigned binary significance.
HTTP	Hypertext Transfer Protocol. A set of rules used for network communication in which information is sent via sockets.
HTTPS	An extension to HTTP that enables you to securely transmit sensitive data over the Web.

I

I/O	An abbreviation for input/output.
identifier	A sequence of alphanumeric characters, underscores (_), and dollar signs (\$) that signifies a statement, text string, variable, or routine to the compiler.
implied-decimal data	A signed number with a whole number part and decimal places.
ImpliedDecimalDesc	An implied-decimal descriptor (for example, d10.1). ImpliedDecimalDesc extends VariantDesc by adding implied-decimal-specific functionality.
implied key of reference	A field in your data record that matches the key defined by the READ or FIND statement. The implied key of reference is used if you don't explicitly specify a key of reference with the KEYNUM qualifier.
include file	A file containing data definitions that is included by a routine using the .INCLUDE compiler directive. Also called a definition file.
index	An ordered set of key values associated with records in an ISAM file. An index can be in ascending or descending order and can allow duplicate keys.
indexed access	The method of retrieving a specific record within an index file that eliminates having to search the file sequentially. Also called keyed access.
Indexed Sequential Access Method	See ISAM .

indexer	A special property that allows data in a class to be accessed using array syntax.
inherit	To derive functionality and data from another class.
initial value	The value for a field before any procedure division statements are executed. You can set an initial value in your field definition in the data division.
inner class	See nested class .
in-place merge	Combining two or more ordered files without creating a new output file; the files are merged back into the first input file.
input	The data that a computer receives.
input/output statement	A statement that controls the exchange of data between the executing program and the input/output devices (such as printers, terminals, disks, and so forth) attached to the computer system.
instantiate	To create an object of a specific class.
integer data	A byte-oriented, binary representation of a signed whole number.
IntegerDesc	An integer descriptor (for example, i1). IntegerDesc extends VariantDesc by adding integer-specific functionality.
interface	A contract that a class agrees to abide by. An interface defines the properties, methods, and events that a class must use without providing any code itself. A class that implements an interface agrees to implement every aspect of that interface exactly as it is defined.
internal subroutine	A routine that begins with a label and whose code is part of the calling routine.
interprogram communication statement	A statement that passes and obtains data from interactive processes.
interrupt character	A keyboard character sequence that interrupts the normal flow of program execution.
intrinsic function	A Synergex built-in function that generates a value.
ISAM	An abbreviation for Indexed Sequential Access Method, which is a way of organizing data in which an index is created to show where data records are stored.

ISAM file	A data file in b-tree format that is organized to support both indexed access for optimal, keyed record access and efficient sequential record processing in any key order. See ISAM .
iteration	A single repetition of a loop.
K	
key	A field, fields, or parts of a field used to locate records in a file and define a sequential order in which to process that file.
key of reference	A key that defines which ISAM file index is used in sequential operations. It is established by the READ or FIND statement, or by the POSRFA subroutine. Records accessed by READS are retrieved in the order of the key of reference.
key value	A value that points to a specific data record as defined by the key. The index file contains these key values in sequential order as defined by the key.
keyword	A word that has predefined meaning in the Synergy Language and that identifies a statement, compiler directive, or language feature.
keyed access	The method of retrieving a specific record within an indexed file that eliminates having to search the file sequentially. Also called indexed access.
L	
leaf block	A block in the final level of an ISAM file's index that is read last on any keyed access. A leaf block consists of sorted key values with a one-to-one correspondence between a leaf entry and a data record.
label	A symbolic name for a statement in the procedure division. A label enables you to transfer program control to the referenced statement.
librarian	The component of Synergy Language that creates and maintains object libraries.
linker	The component of Synergy Language that combines the object files created by the compiler to produce executable code files.
linking	The process that combines one or more routines into an executable program or library.

listing counter	A tally of the number of times you use .LIST (the counter is incremented) and .NOLIST (the counter is decremented). Listing is enabled while the counter is greater than or equal to zero.
literal	A value that doesn't change while the program is being executed, such as literal, a literal field, or a .DEFINE identifier. Also called a constant.
local data	Data that can only be accessed by the defining routine, such as global common data and subroutine arguments.
local routine structure	A structure that is defined inside a local routine.
lock	To keep a file or record from being modified by another channel or program.
logical assignment	An abbreviation defined at the operating system level for a device, directory, or path name. Also called an environment variable.
logical line	One or more physical lines of code that function as a single Synergy Language instruction. The second and all subsequent physical lines must begin with a continuation character (&).
logical screen	The characters and attributes associated with a window that are stored in memory.
loop	A sequence of instructions that is executed repeatedly until a specified condition is met.
M	
mailbox	A logical device in which a character stream can be written to and read from in first-in-first-out order.
main routine	The entry point into a program.
member	A general term for any item that can be declared within another declaration, such as a namespace or class.
memory handle	A unique reference for protected Synergy Language mappable memory.
memory usage summary table	An optional compiler listing that lists the amount of memory required by each program component.

message manager	One of two facilities provided by Synergy Language to process messages sent between programs.
message pool	A memory area used by the Synergy message manager that contains all Synergy Language messages.
method	A routine within a class declaration that performs an operation that also has access to all fields of that class and any inherited classes.
method signature	The fully qualified name of a method which includes the namespace name(s), class name(s), method name, argument types, and return type in the form <i>namespace.class.method(type[, type, ...])rettype</i> .
modular programming	A method of programming that divides a large program into smaller routines.
module descriptor block	A block that contains descriptive information about a routine, such as the routine name, the program section offsets, and the program section lengths.
module identifier string	A string defined by the .IDENT compiler directive that is to be included in a module's object records. Refer to your OpenVMS manual for more information about "module identifier" strings.
"module version" string	A string defined by the .IDENT compiler directive.
N	
<i>n</i>-way merge	Combining two or more ordered files into a separate output file.
namespace	A unique name that represents a grouping of classes, structures, subroutines, and functions (and interfaces and delegates in .NET) that is used to prevent naming collisions between them. A default namespace is the namespace to which subroutines and functions that have no declared namespace are added.
native integer	The storage format for integer data on a given CPU.
nested class	A class declared entirely within another class declaration.
null	An empty value or lack of information. (Note that a zero and a blank are not null, because they convey information.)
numeric	The digits 0 through 9.

numeric data type	A term that encompasses the data types decimal, implied-decimal, integer, packed, and implied-packed.
numeric expression	An expression whose result is a numeric value.
O	
object	An instance of a class.
object code	The system-specific information that the compiler generates.
object file	A file that is created when a source file is compiled. An object file contains one or more object modules.
object handle	A variable that is used to operate on a class or its members.
object library	A group of subroutine object modules in a single file created with the Synergy librarian. Object libraries are linked into an executable file with the Synergy linker.
object module	A compiled routine.
occlusion	An overlapping that occurs when one window is placed over all or part of another window.
offset	A specific point in a record where a field begins.
OLB	An abbreviation for object library. See object library .
OpenSSL	A third-party library that provides SSL support to the Synergy HTTP document transport API.
operand	The data element in an expression.
operator	The part of an expression that performs the action on the operands. Synergy Language has the following operators: unary plus and minus, arithmetic, assignment, rounding, relational, string relational, Boolean, bitwise, and concatenation and reduction.
outer class	The class in which a nested class is nested.
overlay	A technique that allows two or more fields to occupy the same data location.
overlay indicator	The symbol “,X”, which indicates that you are defining an alternate layout for a record.

overlay window	A window that occupies the same area as another window. The data, window size, and data attributes of the window are the same as those of the overlaid window.
overload	To give another method in the same class the same name but different parameters.
override	To provide a method declaration in an inherited class that supersedes the method declaration with the same signature in the parent class.
P	
paint character	The character that replaces a deleted character on the screen.
palette	A set of colors available to Synergy Language programs.
parameter	A variable to which a value may be assigned.
parent class	A class higher in the class inheritance hierarchy.
parent process	The process that has created a child process or subprocess. Control returns to the parent process once the child process is complete.
parse	To separate statements into syntactic units.
path	The file hierarchy through which control passes to locate a particular file, or a way to reference a field within a group. The latter is referred to as a “variable path” in this manual.
physical line	An actual line of code in your program that contains no line breaks or carriage returns.
physical screen	The CRT on which the visible portions of placed windows are displayed. Before you can see changes to the physical screen, you must update the screen.
PID	A unique project and programmer number combination assigned by the OpenVMS operating system.
placement order	The order in which windows are placed on the screen, with the last window placed being the highest in the order. Windows higher in the placement order occlude those that are lower in the placement order.
pointer	An address or other indication of a data location.
portable integer	An integer that is stored in a constant, system-independent format.

procedure division	The part of your routine that contains the executable statements and defines the processing algorithms.
processing area	A subset of the total window area that can be modified.
prefetch	To read ahead records or instructions that are expected to be processed into a higher-speed storage or memory buffer, thus speeding up program execution.
profiler	A performance analysis tool included with Synergy Language.
property	A group of specialized methods that are accessed as if they were a field.
prototype	A member signature without an implementation.
pseudo array	A single-dimension array of type a , d , d. , i1 , i2 , i4 , i8 , p , or p. (for example, 10a5). We do not recommend using pseudo arrays, because there is no runtime bounds checking even when compiling with -qstrict .

Q

quit character	A character that signals the computer to quit program execution.

R

random access	The process of getting data from or placing data into a data location in which access is independent of the order of storage.
ranged variable	A variable followed by a ranging expression, either in absolute or relative form.
ranging expression	An expression in parentheses after a variable reference that specifies starting and ending character positions relative to the beginning of the variable's data space.
rank	The number of possible dimensions in an array. For example, an array declared as arr, [5,3]a10 has rank 2, because there are two dimensions (5 and 3).
real array	A single or multi-dimensional array of type a , d , d. , i1 , i2 , i4 , i8 , p , or p. that is specified in square brackets immediately preceding the data type (for example, [10]a5).

receive option flag	An indicator set by the RCFLG subroutine that controls the way a RECV statement searches for local, group, and global message queues.
record	A data area containing one or more consecutive fields on a related subject, such as a customer record or a file layout.
record terminator	A character or character sequence that signals the end of a record.
recursion	The ability to XCALL a routine that is currently activated.
re-entrant	The property that enables a routine to be called recursively.
region	A block of code delineated by the .REGION-.ENDREGION compiler directives that Visual Studio in .NET or Workbench will collapse and mark with a small box containing a plus sign. The classes, fields, groups, etc., defined within that region are not visible until the plus sign is clicked to expand the region.
relational operator	An operator that compares two operands (equal to, not equal to, greater than, less than, greater than or equal to, or less than or equal to).
relative file	A file that consists of a series of fixed-length records referenced by their relative position within the file.
relative ranging	A form of a ranging expression that indicates the starting or ending character position and length of a character-sequence range using two arithmetic expressions separated by a colon.
replacement identifier	A symbol that defines a text string that is to be substituted for the symbol throughout the program. The symbol is created with the .DEFINE statement.
replacement text	The character sequence that is substituted for a .DEFINE identifier.
RFA	An abbreviation for Record File Address, which uniquely identifies a record within a file of any type of organization. An RFA is stored as an a6 variable. The RFA enables the program to bypass the index and reaccess the record directly.
RFA use counter	A byte that is incremented each time an RFA position is restored by a STORE or WRITE statement.
root block	The block of an ISAM file's index that is read first on any keyed access. The root block contains entries that point to separator blocks.

rounding operator	The operator that truncates the left operand by the number of digits specified by the right operand in an expression.
routine	A main routine, subroutine, function, or method.
runtime	The main component of Synergy Language that supports the actions requested by your Synergy Language programs during execution.
runtime option flag	An indicator set by the DFLAG or FLAGS subroutine that controls the way the runtime acts in response to the execution of Synergy Language statements.

S

scope	The smallest of one of the following groupings that has the identifier declared as one of its members, not including inherited members: namespace, class, structure, record, group, CATCH block, or local data block.
screen function command	A command that is used with the DISPLAY statement to control terminal-independent screen functions (such as cursor position).
screen update	The act of copying the changes stored in the logical screen to the physical screen.
search list logical	A logical assignment that specifies one or more directory paths to search when looking for a file.
search path	A file specification that includes a search list logical.
separator block	A block of an ISAM file's index that exists as a pathway to the leaf blocks. A separator block consists of entries that contain partial key values that point to other separator blocks or leaf blocks.
sequential access	The process of finding a record by searching a file serially.
sequential file	A file that contains records delimited by an operating-system-specific record terminator. A sequential file can only be accessed in record mode.
shared data	Data that can be accessed by more than one routine.
shared image	A library of executable routines to which you can link statically or dynamically at runtime. Shared images may contain data references, as well as procedure references.

shell	The UNIX operating system command processor.
sign extension	Extending an integer value by replicating the high-order bit throughout additional bytes.
simple array	See pseudo array .
simple variable	A variable that consists of the name of a data area including any path specifications. If the variable is defined as an array, it refers to the first element of the array.
source file	A text file that contains Synergy Language code that has not been compiled or linked.
spooling	The process by which input and output devices appear to be working simultaneously, when in fact the system is inputting or outputting data using buffers, or temporarily storing data on disk or tape files until another part of the system is ready to print or otherwise process it.
SSL	Secure Socket Layer. The most common security protocol on the Internet.
stack	A temporary storage area for processing data and instructions.
statement	A single, logical line of instruction in Synergy Language that declares a data element or directs the procedural execution of a program.
static data	Data accessible to all activations of a routine and guaranteed to keep its current state across activations.
stream file	A file containing data that can be accessed with or without the knowledge of the record structure.
stride	A dimension in an array.
string	The result of an alpha expression that is defined by both its length and content.
string relational operator	An operator (equal to, not equal to, greater than, less than, greater than or equal to, less than or equal to) that compares two alpha operands for the length of both.
structfield	A named reference to a structure and an allocation of memory onto which the structure is mapped.

structure	A layout definition that doesn't allocate any memory but that can be mapped to memory during program execution or used to define a structfield.
structure control directive	A type of compiler directive that classifies and delimits program structure. It is position-dependent within the program source code. See also compilation control directive .
subclass	See derived class .
subfield	A part of a field that is ranged.
submode	The definition of a file's record format.
subroutine	A module of code that is only executed when it is called by another routine. A subroutine does not constitute a complete program. It can be either internal or external.
subscripted variable	A variable that refers to an individual element within a list of like elements. The variable name is followed by the subscript value.
superclass	See parent class .
symbol table	An optional cross-reference table in a compiler listing that lists all data and constant definitions used in a routine.
symbol table offsets	An optional list in a compiler listing that includes all symbols referenced in the preceding routine with their offsets into the symbol table.
symbolic access table	A table created by the compiler and linker when you use the debug options. It contains controls that enable the debugger to perform source-level displays and look up variables by name.
symbolic string relational operator	The symbol representation of a string relational operator (==, !=, >, <, >=, and <=).
Synergy Control Panel	A utility that enables you to translate or otherwise modify error text for errors in the Synergy compiler, linker, librarian, and runtime.
Synergy DBMS	Synergy Language's proprietary I/O subsystem, which is a collection of Synergy database utilities and other components distributed with Professional Series. Synergy DBMS includes all four file types used by Synergy: ISAM, relative, sequential, and stream.
Synergy Language daemon	A daemon that processes License Manager and Synergy message manager requests.

system option	A parameter set by the DBLOPT environment variable.
system-supplied function	An intrinsic or external function that is supplied with Synergy Language.
system-supplied subroutine	An external subroutine that is included in the Synergy runtime.

T

termcap file	The terminal data base file on UNIX. It contains a list of the terminal capability codes and their escape sequences.
terminal buffer	An internal storage area that temporarily holds data from the terminal.
terminating character	A character that ends input.
token	A symbol that represents a name or other entity.
traceback	A record of how your program reached a line of code that encountered a nontrappable error (or a trappable error for which no error trapping was established).
trailing blank	A space at the end of an alpha data area.
trappable error	A warning error from which your program can recover unless you fail to trap it.
tree lock	A short-term lock applied on the index of an ISAM file during operations that access the index. This lock guarantees exclusive index access for the period required by the I/O statement.
truth value	Whether an expression is true or false.
type parameter	A data type placeholder for which an actual type will be substituted when a class member is used.

U

UIC	A unique project and programmer number combination assigned by the UNIX operating system.
unary operator	An operator that acts on a single operand. See operator .
unboxing	The automatic extraction of the type from its related object.

UNIX “what” string A string defined by the .IDENT compiler directive. “What” strings are available on UNIX systems that have Source Code Control System (SCCS). Refer to your UNIX manual for more information about “what” strings.

URI Uniform Resource Indicator. The address technology from which URLs are created.

user data set A one-dimensional alpha array that is associated with a window and maintained within the window data area by the windowing system. You can completely control its parameters and contents.

user-defined function A function that is not supplied by Synergy Language and that begins with the FUNCTION statement.

user-defined subroutine A subroutine that is not supplied by Synergy Language and that begins with the SUBROUTINE statement.

V

value type A structure in the .NET Framework System namespace. A value type accesses data directly.

variable An identifier that refers to a field or record.

variable-length record A record in a file in which records are not equivalent in length.

variable path A way to reference a field or variable within a group.

VariantDesc The base type in the Synergy .NET runtime for traditional Synergy data types. A variable defined as VariantDesc can be set to any descriptor type: IntegerDesc, DecimalDesc, AlphaDesc, etc.

virtual A method qualifier that allows the method to be overridden. Abstract and override methods imply virtual.

W

watchpoint A setting in the debugger. When you set a watchpoint for a variable, the debugger breaks on the line that changes the variable.

wildcard A character that enables an operating system or program to perform functions on multiple files with related names, without your having to specify each file by its full name. Wildcard characters are “?” (or “%” on OpenVMS) and “*”.

window	A portion of the screen committed to a specific purpose, such as a menu, a bordered display, or a pop-up help message. Windows can help you enhance your user interface.
window area	The size of the entire window.
windowing system	A group of external subroutines (referred to as the Synergy windowing API) that control the creation, display, and maintenance of windows and their attributes.
working storage	A section of memory allocated to hold immediate results. Examples of working variables held in working storage are loop-control variables, internal processing flags, and variables to hold transitional results.
wrap	To automatically move a word to the beginning of the next line if it doesn't fit on the current line.
X	
XCALL	An external subroutine or the Synergy Language statement that calls an external subroutine.
Z	
zoned numeric	A traditional DIBOL data storage mechanism.

Index

Symbols

- != operator 3-16
- # character 2-25
- # operator 3-9
- ## operator 3-10
- / operator 3-8
- // operator 3-8
- < operator 3-17
- <= operator 3-17
- = operator 3-4, 3-5
- == operator 3-16
- > operator 3-16
- >= operator 3-17
- @* class 9-406
 - creating a dynamic array 9-407
- @* class data type 2-11
 - assigning 2-16
- @ sign, in file specification 1-43
- @class data type 2-12, 2-16
- @delegate data type 2-6, 2-16
- @interface data type 2-6, 2-16
- @type data type 2-12
- [#,#]* data type 2-14
- [#,#]*class data type 2-14
- [#,#]*type data type 2-14
- [#,#]type data type 2-13
- \$ character 3-30
- % character 7-4
- * character
 - data format 3-30
 - operator 3-8
- + operator
 - addition 3-8
 - arithmetic 3-8
 - string concatenation 3-5
 - unary plus 3-7

- , character
 - currency 9-206
 - data format 3-30
 - decimal point 9-128
- . character
 - currency 9-206
 - data format 3-30
 - separator 9-128
- character
 - arithmetic operator 3-8
 - data format 3-30
 - string concatenation operator 3-5
 - subtraction operator 3-8
 - unary minus operator 3-7

A

- ^A data reference operation 9-10
- %ABS routine 9-11
- absolute ranging 1-32
- ABSTRACT modifier 4-22, 4-112 to 4-114, 4-136 to 4-137
- ACCEPT statement 4-11 to 4-12
- access
 - defining for file 4-231, 4-239
 - list, symbol table 14-2
 - mode, defining on OPEN 4-231
 - modifier
 - class 4-22, 4-48
 - group 4-86, 4-87
 - method 4-112
 - property 4-136, 4-137
 - record 4-156, 4-157
 - structure 4-179
 - type 4-55
- accessibility 8-16
- ACCHR routine 9-12 to 9-15
- ACESC routine 9-16
- activation argument. *See* argument

- activation character
 - carriage return 9-12
 - disabling 9-67
 - resetting 4-225 to 4-226, 9-13
- ActiveX API 10-2 to 10-11
 - binding routines 10-2
 - callback routine 10-2, 10-24 to 10-25
 - compiler option 10-2
 - data type 10-6
 - debugging 10-8 to 10-9
 - example 10-9
 - macros and constants required by 10-2
 - properties 10-4 to 10-6
 - UI Toolkit support 10-3
 - Windows support 10-3
- ActiveX container
 - creating 10-16
 - deleting 10-18
 - hiding 10-21
 - property
 - getting value 10-19, 10-20
 - setting 10-26
 - showing 10-28
- ActiveX control
 - browsing 10-3
 - calling method 10-15
 - deleting 10-18
 - licensing considerations 10-7
 - loading into container 10-22 to 10-23
 - property
 - getting value 10-19, 10-20
 - setting 10-26
 - registering 10-2
 - Workbench and 10-3
 - See also UI Toolkit Reference Manual*
- ActiveX Diagnostic utility 10-7
 - See also Synergy Language Tools*
- activex.def file 10-2
- actual argument. *See* argument
- AddDelegateToEvent method 20-27
- ADDHANDLER statement 4-13
- addition 3-8
 - alpha expression 3-5
 - examples 3-24
 - floating-point numbers 12-5
- ^ADDR data reference operation 9-17
- address of data element 9-17
- .ALIGN compiler directive 2-24, 5-4
- /alloc option 4-232
- ALLOC qualifier 4-208
- allocation size, defining on OPEN 4-208, 4-232
- alpha
 - assignment
 - to alpha 3-26
 - to numeric 3-27
 - character 1-10
 - data type 2-7
 - accessing data as 9-10
 - accessing formal parameter as 9-19
 - accessing formal parameter as 9-20
 - passing 2-15
 - expression
 - evaluated as binary number 9-32
 - evaluated as octal number 9-224
 - ISAM key 9-174
 - literal 1-22
 - adding 3-5
 - splitting across lines 4-6
 - subtracting 3-5
 - operand, comparing 3-11
 - relational comparisons B-1
 - variable, European conventions 9-128
- alphabetic character 1-10
- AlphaEnumerator class 9-439
- alphanumeric character 1-10
- Amb_BackColor property 10-5
- Amb_ForeColor property 10-5
- Amb_LCID property 10-5
- Amb_UserMode property 10-5, 10-6
- ambient property 10-5, 10-26
- ampersand 4-6
- .AND. operator 3-12 to 3-13
- and operator
 - bitwise 3-13
 - Boolean 3-12 to 3-13
- ANS DIBOL standard 1-4
- API
 - ActiveX 10-2 to 10-11
 - DLL 11-3 to 11-11
 - floating point 12-2 to 12-3
 - HTTP document transport 13-2 to 13-13
 - .NET assembly 20-1 to 20-27
 - routine call block 15-2 to 15-4
 - socket 16-3 to 16-13

- symbol table 14-2 to 14-6
- Windows printing 17-4 to 17-14
- XML 18-4 to 18-111
- append mode 4-124
- application window
 - caption 19-33
 - opening 4-221
 - position 19-88
- arc cosine, floating-point number 12-6
- arc sine, floating-point number 12-7
- arc tangent, floating-point number 12-8
- ^ARG data reference operation 9-19
- ^ARGA data reference operation 9-20
- ^ARGDIM data reference operation 9-21
- ^ARGN data reference operation 9-23
- ^ARGNUM routine 9-24
- ^ARGTYPE routine 9-25
- argument
 - accessing as numeric 9-23
 - activation 7-14, 7-15 to 7-20
 - actual 7-14, 7-15 to 7-20
 - array, declaring 2-36
 - data types 2-18, 2-34, 7-12
 - determining if defined 9-94
 - determining if passed 9-242
 - dimension number 9-21
 - displaying at current position 19-46
 - formal 7-15 to 7-20
 - accessing as alpha 9-19, 9-20
 - accessing as numeric 9-23
 - number passed 9-223
 - group, declaring 2-36
 - loading
 - data into 21-23
 - value into 21-24, 21-26
 - mapping during routine calls 7-15 to 7-20
 - modifying
 - ActiveX callback routine 10-24 to 10-25
 - value 2-35
 - number passed 9-223
 - optional 7-13
 - passing 2-35, 7-14 to 7-20
 - by descriptor 2-35
 - by reference 2-35, 9-263, 21-5, 21-7
 - by value 2-35, 9-376, 21-5
 - descriptor 9-97, 21-5
 - external reference 9-400
 - to non-Synergy Language routine 21-5
 - returning data type for 9-25
 - returning position of 9-24
 - setting in routine call block 15-15 to 15-17, 15-19
 - typing 7-12
- arithmetic operator 3-8
- array
 - accessing as single element 1-29
 - declaring arguments in routine 2-36
 - defining fields as 2-24 to 2-26
 - dynamic 2-25, 9-407
 - pseudo 2-25, 2-34, 2-38
 - real 1-27, 2-25
 - size 9-310
- Array class 2-25, 9-410
- ArrayList class 2-25, 2-26, 8-20, 9-403
 - 0-based 9-414 to 9-416
 - 1-based 9-417 to 9-419
- ASCII character 1-10
 - converting
 - decimal value to 9-26
 - to decimal 9-92
 - processing 4-70
 - set B-1 to B-2
- ASCII routine 9-26 to 9-27
- ASCR5 routine 9-28
- assembly
 - duplicate 20-5, 20-11
 - See also* .NET assembly API
- Assembly routines, interfacing to 21-3
- assignment operator 3-22
 - alpha 3-4
 - effect on precedence 3-22
 - numeric 3-5
- assignment statement 3-26 to 3-40
 - alpha
 - to alpha 3-26
 - to numeric 3-27
 - decimal
 - to decimal 3-33
 - to implied-decimal 3-34
 - to integer 3-34
 - format of 3-26

- implied-decimal
 - to decimal 3-36
 - to implied-decimal 3-35
 - to integer 3-36
- integer
 - to decimal 3-37
 - to implied-decimal 3-38
 - to integer 3-38
- numeric to alpha 3-28
- AST. *See* asynchronous: system trap
- ASTRST routine 9-29
- ASTSAV routine 9-30
- asynchronous
 - port 9-161
 - system trap 9-30
- %ATRIM routine 9-31
- ATTR_ALL option 19-13
- ATTR_BLNK option 19-13, 19-14
- ATTR_BOLD option 19-13
- ATTR_CHNG function 19-14
- ATTR_CLR function 19-15
- ATTR_ITAL option 19-13, 19-14
- ATTR_LOAD function 19-15
- ATTR_RVRS option 19-13
- ATTR_SET function 19-14
- ATTR_UNDR option 19-13
- attribute
 - ActiveX object 10-4 to 10-6
 - border 19-27
 - changing 19-27
 - returning 19-61
 - character 19-49
 - copying between windows 19-76
 - display 19-5
 - changing 4-42, 19-14
 - clearing 19-15, 19-47
 - setting 19-14
 - specification 19-13
 - display area 19-37
 - processing area contents 19-17
 - returning information about 19-61
 - screen 4-42
 - title 19-27, 19-31
- XML
 - adding 18-75
 - adding list of attributes to list 18-31
 - adding to list 18-30
 - assigning handle as value 18-28
 - clearing list 18-33
 - copying 18-19, 18-34
 - counting 18-35
 - creating 18-20
 - creating list 18-36
 - finding 18-38, 18-41
 - getting name 18-22
 - getting value 18-23, 18-65
 - inserting in list 18-39
 - manipulating 18-11
 - manipulating list 18-11
 - releasing 18-21
 - releasing list 18-37
 - removing from attribute list 18-42
 - removing from element 18-71
 - returning 18-60
 - returning handle 18-24
 - setting name 18-26
 - setting value 18-27
- AX_BIND routine 10-2, 10-13 to 10-14
- %AX_CALL routine 10-15
- %AX_CREATE routine 10-16
- %AX_DELETE routine 10-18
- AX_GET routine 10-4, 10-19
- %AX_GETINT routine 10-4, 10-20
- AX_HIDE routine 10-5, 10-21
- AX_INPUT routine. *See UI Toolkit Reference Manual*
- %AX_LOAD routine 10-22 to 10-23
- %AX_NEWVAL routine 10-24 to 10-25
- AX_SET routine 10-4, 10-5, 10-26
- AX_SHOW routine 10-5, 10-28
- AX_TIMEOUT routine 10-29
- AX_TKCID routine. *See UI Toolkit Reference Manual*
- AX_TKWIN routine. *See UI Toolkit Reference Manual*
- AX_WANTSKEY routine. *See UI Toolkit Reference Manual*
- axutil.exe utility 10-7

B

- `^B` routine 9-32 to 9-33
- background color
 - ActiveX object 10-5
 - setting 19-5
- backup mode 9-329, 9-332
- `.BAND.` operator 3-13
- base `e`
 - logarithm 12-14, 12-15
 - returning 12-12
- baud rate, setting 9-160
- `BEGFL` routine 9-34
- `BEGIN-END` block 4-9, 4-15
 - scope of data in 2-43, 2-47, 7-4
 - terminating execution 4-51
- bell, issuing character 19-37
- Berkeley Sockets 16-3
- `%BIN` routine 9-35 to 9-36
- binary
 - alpha expression as 9-32
 - input 4-75
 - operator 3-4, 3-15
 - representation, bit value in 9-37, 9-39
- binding 1-6
 - callback routine to ActiveX event 10-2, 10-13 to 10-14
 - automatic 10-2, 10-16, 10-22
 - UI Toolkit support 10-3

See also Synergy Language Tools
- bit, low-order 9-320
- `%BIT_IS_CLR` routine 9-37 to 9-38
- `%BIT_IS_SET` routine 9-39 to 9-40
- bitmap, printing 17-39
- bitwise operator 3-13
- `%BKSTR` routine 9-41
- `BKTSIZ` qualifier 4-209
- blank
 - line 4-7
 - not padding data area with 4-229
 - space in logical line 4-6
 - stripping trailing 9-350, 9-352
- `bldism` utility. *See Synergy Language Tools*
- blinking attribute 19-13
- `BLKSIZ` qualifier 4-210
- block
 - number of 4-214, 4-232
 - pointer 21-8
 - size, setting on `OPEN` 4-210
 - submode 4-124, 4-128
- `BLUE_PART` macro 10-5
- `.BNAND.` operator 3-13
- `.BNOT.` operator 3-13
- boldface attribute 19-13
- Boolean
 - data type 2-7
 - passing 2-15
 - operator 3-12 to 3-13
- `.BOR.` operator 3-13
- border
 - attribute 19-27, 19-61
 - changing 19-27
 - color 19-30, 19-61
 - creating 19-27
 - line-drawing character 19-28, 19-61
 - partial 19-30
 - reverse video 19-14
 - turning off 19-30
 - turning on 19-30
- bound program 1-6
- boxing 8-19
- bracketed array. *See* real array
- break on the channel 9-42
- `BREAK` routine 9-42
- breakpoint, ignoring 5-36
- broadcast message, trapping 9-359
- brush, creating 17-30
- `BTOD` routine 9-43 to 9-44
- bucket size, setting on `OPEN` 4-209
- buffer
 - draining 4-176
 - flushing 4-61
 - number of local I/O 4-211, 4-232
 - size, setting on `OPEN` 4-212, 4-232
 - terminal, size of 9-341
 - writing out modified 9-103
- `/bufnum` option 4-232
- `BUFNUM` qualifier 4-211
- `/bufsiz` option 4-232
- `BUFSIZ` qualifier 4-212
- `.BXOR.` operator 3-13
- `BYREF` modifier 2-33, 2-35
- byte data type 2-7
- `BYVAL` modifier 2-33, 2-35

C

- C interface 1-9, 21-1 to 21-27
 - building DLL or shared object with 21-27
 - example 21-8
 - initializing for use with DLLs 11-10 to 11-11
 - passing arguments 21-5
 - runtime interface functions 21-8 to 21-26
- C routine
 - adding to runtime 21-3
 - building DLL or shared object that contains 21-27
 - calling external subroutine from 7-3
 - calling intrinsic function from 7-6, 21-3
 - converting null-terminated string to Synergy alpha variable 9-322
 - interface functions 21-8 to 21-26
 - processing 21-6 to 21-7
- CA. *See* certificate: authority
- /cache option 4-232
- caching ISAM files 4-232
- call block. *See* routine call block
- Call method 20-13
- CALL statement 4-16 to 4-17
- callback routine 10-2
 - binding to ActiveX event 10-2, 10-13 to 10-14
 - contents of 10-14
 - modifying parameter 10-24 to 10-25
 - passing arguments to 10-2
- calling
 - external function 4-204
 - external subroutine 4-204
 - internal subroutine 4-16 to 4-17
 - using routine call block 15-6 to 15-7
- caption, ActiveX container 10-4
- Caption property 10-4
- carriage return as activation character 9-12
- cascaded IF statement 4-91
- case sensitivity 9-128, 9-174
- CASE statement 4-18 to 4-21
- casting 2-16, 7-10, 8-17
 - converting vs. 2-16
- CATCH statement 4-186, 6-4
- catching errors 4-122
- CDATA 18-5
- certificate 13-2, 13-5
 - authority 13-2, 13-5, 13-6, 13-9
 - importing and exporting in IE 13-8 to 13-9
 - requesting 13-6
- chaining
 - overhead 1-6
 - windowing subroutines 19-74
- channel
 - closing
 - deleting file and 4-140
 - saving file and 4-28 to 4-29
 - determining if open 9-48
 - finding next available 9-327
 - flushing buffers 4-61
 - opening 4-124 to 4-134
 - status 9-48, 21-14
- %CHAR routine 9-26, 9-45
- character
 - attributes 19-49
 - deleting on VT-style terminals 9-129
 - displaying 4-40 to 4-43
 - pointer, converting to alpha 9-322
 - receiving from channel 4-11 to 4-12
 - special function C-1
 - submode 4-125
- character set
 - ASCII B-1 to B-2
 - Synergy Language 1-10
- child, relationship to parent window 19-76
- chklock utility 4-244
 - See also Synergy Language Tools*
- CHNOPN routine 9-46 to 9-47
- %CHOPEN routine 9-48
- cipher 13-10 to 13-11
- ciphertext data, decrypting 9-69
- class 8-1 to 8-21
 - collection 2-26
 - common and global in 4-23, 8-7
 - creating 8-4
 - data type 2-6
 - destroying instance 8-13
 - field, record, and group 8-7
 - generating for .NET 20-2, 20-4
 - inheriting functionality and data 8-15
 - instantiating and using 8-12
 - method 8-5
 - nested 4-24, 8-9
 - operator 3-14 to 3-20
 - binary 3-15
 - unary 3-14

- structure 2-44
 - subroutine and function 8-6
- CLASS statement 4-22 to 4-25
- CLEAR statement 4-26
- Clear to Send protocol 9-160
- clearing
 - line 19-38
 - processing area 19-22
 - screen 4-41
 - variable 4-26
 - window 19-37, 19-38
- client/server. *See* xfServer; xfServerPlus
- ClipChildren property 10-4
- CLOSE statement 4-28 to 4-29
- closing
 - channel
 - deleting file 4-140
 - saving file 4-28 to 4-29
 - symbol table 14-10
- CMDLN routine 9-49 to 9-52
- %CNV_IP routine 9-53
- %CNV_PI routine 9-54
- code
 - controlling listing of 5-35, 5-37, 5-41
 - converting to use windowing subroutines 19-101 to 19-103
- collapsible region 5-40
- collating printed copies 17-23
- collection class 2-26, 8-20
 - 0-based 9-414 to 9-416
 - 1-based 9-417 to 9-419
- collection, iterating through 4-68
- color 19-5 to 19-10
 - ActiveX object 10-5
 - background 17-25
 - border 19-30
 - current palette entries 19-62
 - enabling 19-5
 - field 19-50
 - modifying palette 19-87
 - printing 17-23
 - processing area 19-19
 - returning information about 19-61
 - setting current 19-38
 - text, printed 17-25
 - title 19-31
 - transparent 17-40
- comma
 - currency 9-206
 - data format 3-30
 - instead of decimal point 9-128
- command, executing on OPEN 4-131
- command line, returning 9-49
- comment 4-7, 18-50
- common
 - area, loading into variable 9-139
 - global vs. 2-42
 - record 2-39
 - defining 2-40
 - in class 4-23, 8-7
 - overlay 2-45
 - variable 1-24, 4-31
- COMMON statement 2-40, 4-30 to 4-32
- EXTERNAL 2-40, 4-30
- EXTERNAL COMMON 2-40
- GLOBAL 2-40, 4-30
- GLOBAL COMMON 2-40
- comparing character strings B-1
- compile-time
 - data reference operation 7-11
 - expression 3-41
 - optimization 3-42
- compiler 1-8
 - directive 1-18 to 1-20, 5-1 to 5-44
 - option. *See* Synergy Language Tools
 - variable, returning value of 9-378
- compiler listing
 - conditional blocks in 5-41
 - controlling
 - with .START directive 5-41
 - with .LIST directive 5-35
 - with .NOLIST directive 5-37
 - header
 - defining 5-43
 - routine name in 4-105
 - memory usage summary table 5-41
 - page break
 - with .PAGE directive 5-39
 - with .START directive 5-41
 - symbol table offsets 5-41
- See also* Synergy Language Tools

- compiling
 - date of 9-79
 - returning machine code 9-339
 - See also Synergy Language Tools*
 - complex array. *See* real array
 - compound statement 4-9, 4-15
 - COMPRESS routine 9-55
 - compressing
 - data string 9-55
 - ISAM key 9-175
 - record data 9-171
 - computer, identifying 9-105 to 9-106
 - concatenation 3-5
 - concurrent processing, starting 9-277 to 9-280
 - conditional
 - block 5-23
 - compiler listing 5-41
 - with .IF directive 5-16
 - with .IFDEF directive 5-18
 - with .IFNDEF directive 5-20
 - processing 5-23
 - Connectivity Series 1-3
 - CONST modifier 2-21, 4-156
 - constructor 8-10
 - adding to a class 8-4
 - declaring 4-115
 - inheritance 8-15
 - initializer 4-113, 8-10
 - modifiers 8-10
 - container
 - ambient properties 10-5
 - control properties 10-4
 - creating ActiveX 10-16
 - deleting ActiveX 10-18
 - getting property 10-19
 - hiding ActiveX 10-21
 - loading ActiveX control into 10-22
 - local properties 10-6
 - setting property in 10-26
 - showing ActiveX 10-28
 - UI Toolkit 10-3
 - CONTIG qualifier 4-213
 - contiguity, specifying for file 4-213
 - continuation line 4-6
 - control
 - ActiveX property 10-4
 - character
 - modifying operation of 9-301
 - special function C-1
 - returning
 - from external subroutine 4-205
 - from internal subroutine 4-164
 - statement 1-13
 - transferring 1-13
 - conversion operator 3-18
 - converting
 - ASCII to decimal 9-92
 - decimal to ASCII 9-26
 - lowercase to uppercase 4-191
 - uppercase to lowercase 4-101
 - COPY routine 9-57
 - cosine, floating-point number 12-10
 - CPU, returning time 9-59, 9-60
 - %CPUTIME routine 9-59
 - CPUTM routine 9-60
 - CRC 4-218, 4-237, 4-244
 - CREMBX routine 9-61 to 9-63
 - CTL_BEGIN status 19-93, 19-95
 - CTL_DONE status 19-93, 19-95
 - CTL_MORE status 19-93, 19-95
 - currency
 - changing symbol 9-206 to 9-207, 9-220
 - localizing 9-206 to 9-207
 - cursor
 - appearance of 19-79
 - moving 4-42
 - resetting position of 4-41
 - windowing system 19-78
 - CVTIIV routine 9-64
 - CVTIZV routine 9-65
- D**
- ^D data reference operation 9-66
 - D_ABOVE_NORMAL_PRIORITY_CLASS 9-299
 - D_ADDR 9-391
 - data type 2-7
 - D_BACKGROUND_BEGIN 9-300
 - D_BACKGROUND_END 9-300
 - D_BELOW_NORMAL_PRIORITY_CLASS 9-299
 - D_COMP 9-55
 - D_DETACHED 9-314

- D_ERR_SUCCESS 10-31
- D_GAINFOCUS 9-306, 9-314
- D_GRFA_TYPE 4-218, 4-237
- D_HIGH_PRIORITY_CLASS 9-299
- D_IDLE_PRIORITY_CLASS 9-299
- D_INHERIT 9-306, 9-314
- D_MINIMIZED 9-306, 9-314
- D_NOACTIVATE 9-306, 9-308, 9-314
- D_NOCONSOLE 9-306, 9-308, 9-314
- D_NORMAL_PRIORITY_CLASS 9-300
- D_NOWAIT 9-306, 9-314
- D_NOWINDOW 9-306, 9-314
- D_NSPC_CASE flag 14-13, 14-15
- D_NSPC_NONE flag 14-15
- D_NSPC_SPACE flag 14-15
- D_RFA_TYPE 4-218, 4-237
- D_TYPE_A 9-75, 15-12, 15-15
- D_TYPE_D 9-75, 15-12, 15-15
- D_TYPE_HANDLE 15-12, 15-15
- D_TYPE_HVA 15-15
- D_TYPE_I 9-75, 15-12, 15-15
- D_TYPE_ID 9-75, 15-12, 15-15
- D_TYPE_IP 9-75, 15-12, 15-15
- D_TYPE_MEMARG 15-12
- D_TYPE_OBJ 15-12, 15-15
- D_TYPE_P 9-75, 15-12, 15-15
- D_TYPE_REF 15-12, 15-15
- D_TYPE_VAL 15-12 to 15-13, 15-15 to 15-17
- D_UNCOMP 9-55
- DACHR routine 9-67
- DAESC routine 9-68
- data
 - accessing
 - as alpha 9-10
 - as decimal or implied-decimal 9-66
 - from C routine 21-6
 - compressing 9-55, 9-171
 - formatting rules 3-28 to 3-32
 - hierarchy of 3-7
 - inheriting 8-15
 - instance 8-7
 - lifetime of 8-13
 - manipulation statement 1-12
 - returning characteristics of 7-11
 - static 8-7
 - storage 2-39
 - common 2-41
 - global 2-42
 - working 2-40
 - structure 4-8
 - defining in an .INCLUDE directive 5-25
 - mapping 9-208
- data characteristic operation 7-11
- data division 1-5, 4-8
- data reference operation 7-10
 - name 1-22
 - referenced as intrinsic function 7-10
- DATA statement 4-33
- Data Terminal Ready signal 9-161
- data type 2-2 to 2-14
 - ActiveX 10-6
 - argument 2-34
 - casting 2-16
 - class 2-6
 - classification in Synergy .NET 2-6
 - descriptor 2-5
 - field 2-22
 - parameter 2-34
 - return value 2-38
 - returning 9-75
 - returning numeric value for 9-25
 - rules for passing 2-15 to 2-17
 - supported 1-10
 - Synergy .NET 2-5
 - usage 2-18
 - value 2-5
- DATA_DECRYPT routine 1-36, 9-69
- DATA_ENCRYPT routine 1-36, 9-71
- DATA_SALTIV routine 1-36, 9-73
- ^DATATYPE data reference operation 9-75
- date
 - compilation 9-79
 - current 9-77, 9-221
 - Julian 9-198, 9-221
 - numeric 9-221
- %DATE routine 9-77
- %DATECOMPILED routine 9-79
- %DATETIME routine 9-80 to 9-81
- day of week, returning 9-387
- dbl command. *See Synergy Language Tools*
- DBL\$ subroutines 9-82 to 9-85
- dbl_exec runtime interface function 21-11

- dbl.def file 4-207
- DBL\$DEVCLT routine 9-86 to 9-87
- %DBL\$EXECUTE_IMAGE_ROUTINE routine 9-88 to 9-89
- dblproto. *See* Prototype utility
- DBL\$SETKRF routine 9-90
- DBL\$SNDOPR routine 9-91
- DBLSORT utility, interface to 9-312
- dbr command. *See* Synergy Language Tools
- dbrpriv runtime 9-300
- dbssvc service runtime 9-346
 - See also* Synergy Language Tools
- dbspriv service runtime 9-300
 - See also* Synergy Language Tools
- dbssvc service runtime 9-321, 9-346
 - See also* Synergy Language Tools
- debugger. *See* Synergy Language Tools
- debugging 1-9
 - ActiveX API 10-8 to 10-9
 - ignoring breakpoint 5-36
 - .NET assembly API 20-5
 - See also* Synergy Language Tools
- decimal
 - alpha assignment 3-28
 - assignment
 - to decimal 3-33
 - to implied-decimal 3-34
 - to integer 3-34
 - converting
 - from ASCII character 9-92
 - to ASCII character 9-26
 - to binary 9-101 to 9-102
 - data type 2-8
 - accessing data as 9-66
 - passing 2-15
 - ISAM key 9-174
 - literal 1-23
 - negative, storing 2-8, B-1
- decimal point
 - comma as 9-128
 - currency 9-206
 - storing 2-8
- declaration, XML 18-47
- declarative statement 1-11
- %DECML routine 9-92 to 9-93
- decompressing data 9-374
- DECR statement 4-35 to 4-36
- decrementing
 - listing counter 5-37
 - variable 4-35 to 4-36
- decryption 1-35 to 1-37, 9-69, 9-73
- /DECSCOPE compiler option 9-310
- .DEFINE compiler directive 5-7 to 5-12
- ^DEFINED data reference operation 9-94
- defining a variable 1-24
- definition
 - file, declaring data in 5-31
 - window 19-11
- delegate 8-9
 - associating with event 20-27
 - creating 20-25
 - removing from event 20-28
- DELET routine 9-95
- DELETE statement 4-37 to 4-38
 - restriction 4-125
- deleting
 - ActiveX object 10-18
 - argument from routine call block 15-10
 - character 9-129, C-1
 - file 9-95, 9-247
 - mailbox 9-96
 - pen 17-31
 - record 4-37 to 4-38
 - routine call block 15-11
 - symbol table 14-10
 - symbol table entry 14-11
 - window 19-81
- delimiter 1-10
- DELMBX routine 9-96
- \$DELPRC system service 9-203
- density
 - file 9-171
 - ISAM key 9-175
- /deq option 4-232
- DEQ qualifier 4-214
- DER 13-17, 13-21, 13-25
- ^DESCR data reference operation 2-35, 9-97, 21-5
- descriptor
 - data type 2-5
 - getting 21-18
 - passing argument by 9-97, 21-5
- design mode 10-5, 10-6
- destroying class instance 8-13

- destructor 8-11
 - adding to a class 8-4
 - declaring 4-115
 - inheritance 8-15
- DETACH statement 4-39
- detached process 4-39, 9-277
- device
 - getting class and type 9-87
 - setting and retrieving extended capabilities 17-27 to 17-35
- DFLAG routine 9-98 to 9-100
 - flag 1 (global messages first) 1-41, 9-99
 - flag 2 (wait for message) 1-41, 9-99
 - flag 3 (suppress execution until record locked) 9-100
 - flag 8 (suppress pagination) 9-100
 - flag 9 (suppress error message) 9-100
 - flag 10 (suppress traceback) 9-100
 - WAIT qualifier and 4-242
- diagnostics, ActiveX controls 10-7
- dimension
 - field 2-22, 2-24
 - multiple 1-27
 - parameter 2-34
 - returning for an argument 9-21
- DIRECTION qualifier 4-215
- directory
 - returning current path 9-328
 - setting current path 9-331
- disconnecting program from terminal 4-39
- display area 19-3
 - attributes 4-42, 19-37
 - border 19-4
 - copying parameters 19-20
 - placing 19-87
- DISPLAY statement 4-40 to 4-43
- division 3-8
 - floating-point numbers 12-11
 - implied-decimal or implied-packed operands 3-8
- DLL
 - building with C routines 21-27
 - calling a subroutine or function 11-4
 - closing 11-9
 - example 11-2
 - file extension 11-2
 - opening 11-10
 - variable declaration 11-5
 - written in Visual Basic 11-10
- %DLL_CALL routine 11-4 to 11-8
- %DLL_CLOSE routine 11-9
- %DLL_OPEN routine 11-10
- DM_ALLOC subfunction 9-212
- DM_FREE subfunction 9-213
- DM_GETSIZE subfunction 9-214
- DM_REG subfunction 9-215
- DM_RESIZ subfunction 9-216, 9-217
- DM_STATIC qualifier 15-8
- DO FOREVER statement 4-44
- document object model. *See* DOM tree
- document, XML 18-4
 - adding declaration to 18-43
 - creating 18-45
 - declaration 18-47
 - loading into memory 18-5
 - manipulating 18-9
 - releasing 18-46
 - removing declaration from 18-49
 - returning root element 18-48
 - setting root element 18-52
 - well-formed 18-5
 - writing contents
 - to file 18-53
 - to string 18-54
- DOM tree 18-6
 - example of building 18-16
 - manipulating 18-9
- DotNetAssembly class 20-9
- DotNetAssembly constructor 20-10
- DotNetDelegate class 20-9
- DotNetDelegate constructor 20-25
- DotNetException class 20-3
- DotNetObject class 20-9
 - property indexer get method 20-24
 - property indexer set method 20-23
- DotNetObject constructor 20-12
- DotNetObject.AddDelegateToEvent method 20-27
- DotNetObject.Call method 20-13
- DotNetObject.GetField method 20-21
- DotNetObject.GetProperty method 20-17
- DotNetObject.RemoveDelegateFromEvent 20-28
- DotNetObject.SetField method 20-19
- DotNetObject.SetProperty method 20-15
- double data type 2-8
- DO-UNTIL statement 4-45
- DR_COM 9-55

DR_COMP 9-55
 DR_UNCOMP 9-55
 drag bar 19-30
 drawing lines 17-30
 horizontal 19-40
 vertical 19-45
 DRCB_ARRAY 15-13, 15-16
 DRCB_READONLY 15-13, 15-16
 DRCB_REPLICATE qualifier 15-8
 DTOB routine 9-101 to 9-102
 duplex printing 17-23
 duplicate
 filename, checking for 9-128
 key 9-175
 DWP_ADDFILE subfunction 17-37
 DWP_BACKCOLOR subfunction 17-25
 DWP_BEGINJOB subfunction 17-36
 DWP_BEGINPAGE subfunction 17-37
 DWP_BITMAP subfunction 17-39
 DWP_COLLATE subfunction 17-23
 DWP_COLOR subfunction 17-23
 DWP_COPIES subfunction 17-22
 DWP_DELPEN subfunction 17-31
 DWP_DELPRINTER subfunction 17-30
 DWP_DUPLEX subfunction 17-23
 DWP_ENDJOB subfunction 17-37
 DWP_ENDPAGE subfunction 17-37
 DWP_FILL subfunction 17-38
 DWP_FONT subfunction 17-24
 DWP_FORMNAME subfunction 17-23
 DWP_GETFILENAME subfunction 17-30
 DWP_GETPEN subfunction 17-30
 DWP_GETPRINTER subfunction 17-30
 DWP_GETSTATUS subfunction 17-33
 DWP_LARGESTFONT subfunction 17-32
 DWP_LINE subfunction 17-38
 DWP_ORIENTATION subfunction 17-20
 DWP_PAPERLENGTH subfunction 17-22
 DWP_PAPERSIZE subfunction 17-21
 DWP_PAPERSOURCE subfunction 17-25
 DWP_PAPERWIDTH subfunction 17-22
 DWP_PHYSICALHEIGHT subfunction 17-23
 DWP_PHYSICALWIDTH subfunction 17-23
 DWP_PIXELHEIGHT subfunction 17-23
 DWP_PIXELWIDTH subfunction 17-23
 DWP_PREVIEW subfunction 17-16
 DWP_PREVIEW_CANCEL return value 17-16
 DWP_PREVIEW_PRINT return value 17-16
 DWP_PREVIEW_PRINTDLG flag 17-16
 DWP_PREVIEW_REGEN 17-18
 DWP_PREVIEW_REGEN return value 17-16
 DWP_PREVIEW_TOOLBAR flag 17-16
 DWP_PRINT subfunction 17-15
 DWP_PRINTDLG subfunction 17-27, 17-28
 DWP_PRINTDLG_ALLPAGES flag 17-28
 DWP_PRINTDLG_APPLY return value 17-28
 DWP_PRINTDLG_APPPAGE flag 17-28, 17-29
 DWP_PRINTDLG_CANCEL return value 17-28
 DWP_PRINTDLG_COLLATE flag 17-28
 DWP_PRINTDLG_CURRENT flag 17-28
 DWP_PRINTDLG_DISABLECURPAGE flag 17-28
 DWP_PRINTDLG_DISABLEPAGENUMS flag 17-28
 DWP_PRINTDLG_DISABLESELECT flag 17-28
 DWP_PRINTDLG_PAGES flag 17-28
 DWP_PRINTDLG_PRINT return value 17-28
 DWP_PRINTDLG_SELECTION flag 17-28
 DWP_PRINTDLG_SHOWTOFILE flag 17-28
 DWP_PRINTDLG_STYLEPRINT flag 17-28
 DWP_PRINTDLG_TOFILE flag 17-28
 DWP_QUALITY subfunction 17-23
 DWP_RESOLUTIONX subfunction 17-23
 DWP_RESOLUTIONY subfunction 17-23
 DWP_SCALE subfunction 17-22
 DWP_SETPAGES subfunction 17-40
 DWP_SPECS subfunction 17-25
 DWP_TEXTCOLOR subfunction 17-25
 DWP_TEXTHEIGHT subfunction 17-32
 DWP_TEXTMETRICS subfunction 17-32
 DWP_TEXTWIDTH subfunction 17-32
 DWP_TRANSPARENCY subfunction 17-40
 DWP_WRITEOUT subfunction 17-37
 DWP_X_FROM_COL subfunction 17-31
 DWP_XOFFSET subfunction 17-24
 DWP_Y_FROM_ROW subfunction 17-31
 DWP_YOFFSET subfunction 17-24
 DWPC_ values 17-23
 DWPD_ values 17-23
 DWPO_ values 17-20
 DWPP_ values 17-21 to 17-22
 DWPQ_ values 17-23
 DWPS_ values 17-25

- dynamic
 - call generation 15-2
 - library, building with C routines 21-27
 - memory 2-48
- dynamic array
 - data type 2-13 to 2-14
 - field 2-25
 - initial value 2-26
 - of classes 9-407
 - parameter 2-36

E

- echoing input, preventing 9-129

ELB

- global data section and 2-42
- maximum open 9-228
- xfServerPlus and 9-115
- See also Synergy Language Tools*

- element, XML 18-4

- adding
 - attribute 18-75
 - attributes in a list 18-55
 - child element 18-57, 18-58
 - list of elements to list 18-82
 - to list 18-81
- assigning handle 18-78
- clearing list 18-84
- copying 18-62
- counting 18-86
- creating 18-63
- finding 18-89, 18-90, 18-93
- getting child elements 18-61, 18-66
- list
 - adding to 18-81
 - clearing 18-84
 - copying 18-85
 - creating 18-87
 - inserting in 18-91
 - manipulating 18-12
 - releasing 18-88
- manipulating 18-10
- releasing 18-64
- removing
 - attributes from 18-72
 - child from 18-73, 18-74
 - from element list 18-94

- returning
 - attribute value 18-65
 - attributes 18-60
 - root 18-48
 - text handle 18-69
 - setting root 18-52
 - string representation 18-80
 - tag name 18-67, 18-76
 - text value 18-68, 18-77

- .ELSE compiler directive
 - with .IF directive 5-16
 - with .IFDEF directive 5-18
 - with .IFNDEF directive 5-20

- ELSE statement 4-19

- See also IF-THEN-ELSE statement*

- .emf extension 17-3

- EMPBUF routine 9-103

- encryption
 - data 1-35 to 1-37, 9-71, 9-73
 - HTTPS 13-2
 - legal restrictions 13-6
 - network 4-231, 9-171
- end of file, signaling C-2

- END statement 1-5
 - BEGIN block 4-15
 - procedure division 4-135
 - routine 4-46

- .ENDC compiler directive
 - with .IF directive 5-16
 - with .IFDEF directive 5-18
 - with .IFNDEF directive 5-20

- ENDCLASS statement 4-22 to 4-25

- ENDCOMMON statement 4-30

- ENDEXTERNAL statement 4-55

- ENDFL routine 9-104

- ENDFUNCTION statement 4-72

- ENDGLOBAL statement 4-80 to 4-83

- ENDGROUP statement 4-86 to 4-89

- ENDMAIN statement 4-105

- ENDMETHOD statement 4-112

- ENDNAMESPACE statement 4-117

- ENDPARAMS statement 4-47

- ENDPROPERTY statement 4-136

- ENDRECORD statement 4-156 to 4-158

- .ENDREGION compiler directive 5-40

- ENDSTRUCTURE statement 4-179 to 4-181

- ENDSUBROUTINE statement 4-182

- ENDTRY statement 4-186
- ENUM statement 4-48
- enumeration 8-9
 - declaring 4-48
 - including 5-25 to 5-34
- enumeration data type 2-12
 - assigning 2-16
- environment variable
 - changing program behavior based on 9-148
 - setting 9-304 to 9-305
 - translating 9-148 to 9-149
 - UI Toolkit-specific 9-304

See also Environment Variables and System Options
- ENVRN routine 9-105 to 9-106
- .EQ. operator 3-11
- .EQS. operator 3-6
- equal sign 3-4, 3-5, 3-22
- equal to operator 3-6, 3-11
- erasing
 - horizontal line 19-40
 - vertical line 19-44
- %ERLIN routine 9-107, 9-111
- %ERNUM routine 9-108, 9-111
- \$ERR_CATCH literal 4-122
- \$ERR_FILOPT error 9-42
- %ERR_TRACEBACK routine 4-122, 9-109
- ERRMOD routine 9-110
- Erno field 6-6
- error
 - catching 4-122
 - clarifying 9-124
 - displaying text of 9-111
 - handling 6-1 to 6-17
 - issuing in C interface 21-12, 21-13
 - lists 6-3
 - literal 1-24, 6-3
 - message
 - suppressing 9-100
 - system 9-334
 - XML 18-99

See also Synergy Language Tools
 - mnemonics 6-3, 21-15
 - number, determining 6-6
 - socket API 16-5 to 16-9
 - text on OpenVMS 9-382
 - traceback 9-100, 9-109
 - trapping 6-2
 - disabling 4-120
 - enabling 4-121 to 4-123
 - in windowing system 19-98
 - .NET assembly API 20-3
 - WD_ACCEPT and 19-35
 - WD_GETS and 19-39
 - WD_READS and 19-43
 - windowing system 19-98 to 19-100
 - XML parser 18-99
- %ERROR routine 9-111 to 9-112
- ERTXT routine 9-111, 9-113 to 9-114, 9-124
- escape
 - codes
 - HTTP 9-324 to 9-325
 - XML 9-324
 - key 9-12
 - sequence 9-14, 19-36
- escapement 17-24
- escaping characters in a string 9-324 to 9-326, 13-4
 - automatically 13-16, 13-21, 13-28
 - unescaping 9-336
- European convention for alpha variables 9-128
- event
 - ActiveX vs. Windows 10-14
 - binding callback routine to 10-13 to 10-14
 - handler, adding 4-13
 - handler, removing 4-162
- Event class 9-438
- event log 9-330
- /ex option 4-231
- exception
 - catching 4-186, 6-4 to 6-5, 6-6
 - class
 - error mapping 6-8 to 6-17
 - hierarchy 6-6 to 6-8
 - handling 6-4, 9-408
 - .NET 20-3
 - throwing 4-185, 6-4
- Exception class 9-408
- exclusive or operator 3-12
- EXEC routine 9-115
- executable library. *See* ELB
- executing
 - interrupting C-1
 - non-Synergy Language program 9-115
 - program 9-314 to 9-317

- statements, set of 4-18 to 4-21
- suspending 4-168, 9-385
- terminating
 - BEGIN-END block 4-51
 - loop 4-53
 - program 4-175, C-1
- until
 - condition is true 4-45, 4-90 to 4-91, 4-92 to 4-93
 - control is transferred 4-44, 4-163
 - value is reached 4-64, 4-66 to 4-67
 - values are evaluated 4-62
- while condition is true 4-196
- Exif, printing 17-39
- EXIT statement 4-51
- EXITE routine 9-116 to 9-117
- EXITERROR routine 9-118
- exiting windowing system 19-47
- EXITLOOP statement 4-53
- EXITTRY statement 4-54
- explicit formatting 3-29
- exponentiating a floating-point number 12-12
- expression
 - alpha 3-2
 - converting
 - to decimal as zoned numeric 9-402
 - to integer 9-169
 - to packed decimal 9-232
 - evaluating
 - as octal number 9-224
 - at compile time 3-41
 - examples 3-24
 - numeric 3-2
 - optimizing 3-42
 - size of 9-310
- Ext_Focus property 10-4, 10-5
- Ext_hWnd property 10-4
- Ext_Rect property 10-4, 10-5
- Ext_Visible property 10-4, 10-5
- extended property 10-4
- EXTENDS modifier 4-22, 8-15
- Extensible Markup Language 18-4
- extension, default filename 1-43
- EXTERNAL COMMON statement 2-40, 4-30
- EXTERNAL FUNCTION statement 4-55, 7-5
- EXTERNAL modifier 4-99
- external reference, passing argument by 9-400
- external subroutine. *See* subroutine, external

F

- ^F routine 9-119
- false, returning value of 9-120
- %FALSE routine 9-120
- FATAL routine 9-121 to 9-122
- fcompare utility. *See* Synergy Language Tools
- fconvert utility. *See* Synergy Language Tools
- FDL qualifier 4-216
- field
 - character attributes 19-49
 - characteristics 2-21
 - class 8-7
 - color 19-50
 - creating in window 19-50
 - data types 2-18, 2-22
 - defining 2-20
 - displaying in window 19-38
 - filler 2-20
 - filling 19-51
 - getting data from 19-51
 - initial value 2-23
 - initializing 4-98
 - input 19-52
 - integer value conversion 9-64
 - length
 - automatic sizing 2-24, 2-30
 - returning 19-55
 - setting 19-56
 - loading data into 19-55
 - manipulating in window 19-48
 - name 2-24
 - position 2-23, 2-27
 - returning 19-55
 - setting 19-56
 - position indicator 4-100
 - querying value 20-21
 - setting value on instance 20-19
 - transferring contents of 19-51, 19-55
- file
 - access, defining 4-231, 4-239
 - allocation size 4-208, 4-232
 - blocks, adding 4-214, 4-232
 - contiguity 4-213
 - copying to new file 9-57
 - deleting 9-95, 9-247
 - extension 1-43

- ISAM 1-42
 - creating 9-170 to 9-178
 - density 9-171
 - See also Synergy Language Tools*
 - manipulation statement 1-13
 - merging 4-107 to 4-111
 - name, checking for duplicate 9-128
 - open mode 4-127, 4-231
 - pointer, positioning
 - first record 9-34
 - last record 9-104
 - position 4-233
 - queueing for printing 4-102 to 4-104
 - relative 1-42
 - sequential 1-42
 - setting access type 9-296
 - size of records, defining on OPEN 4-231, 4-234
 - sorting 4-169 to 4-174
 - source, including 5-25 to 5-34
 - specification 1-43, 4-216
 - stream 1-42
 - temporary 4-231, 4-241
 - type 1-42, 4-232, 4-236
 - windowing definitions 19-11
- fill character 19-51
- fill region 17-38
- FILL routine 9-123
- filler field 2-20
- filling field with character 19-51
- FILNM routine 9-124
- FINALLY statement 4-186, 6-4
- FIND statement 4-57 to 4-60
 - inconsistencies after I/O error 6-2
 - RFA and KEYNUM qualifiers on OpenVMS 4-147
- finding
 - record 4-57 to 4-60
 - symbol table entry 14-12
 - window 19-81
- ^FIRST qualifier
 - FIND statement 4-57
 - READ statement 4-145
 - WRITE statement 4-198
- fixed-length
 - data, writing 4-141 to 4-142, 4-143
 - record, specifying on OPEN 4-231, 4-234
- fixed-point data 2-8
- /fl option 4-231
- flag
 - runtime option 9-98
 - terminal-oriented 9-364
- FLAGS routine 9-126 to 9-130
 - flag 1 (European formatting conventions) 9-128
 - flag 2 (case-sensitive terminal input) 9-128
 - flag 3 (duplicate filename check) 9-128, 9-264
 - flag 4 (replace character with paint character) 9-129, 9-233, 9-234
 - flag 5 (echo input off) 9-129
 - flag 6 (record attributes disabled) 9-129
 - flag 7 (suppress program termination message) 9-129
 - flag 8 (ignore interrupt characters) 9-130
 - flag 9 (terminal input for READS) 9-130
 - flag 10 (terminator character echo off) 9-130
 - WD_ACCEPT function 19-35
 - WD_GETS function 19-39
 - WD_READS function 19-43
 - xfServerPlus and 9-128
- flicker, reducing 10-4
- float data type 2-9
 - arrays and 2-14
- floating point API 12-2 to 12-3
- floating-point number
 - adding 12-5
 - comparing 12-9
 - converting
 - from implied-decimal 12-13
 - to implied-decimal 12-22
 - cosine of 12-6, 12-10
 - dividing 12-11
 - encoding, passing, retrieving values 9-134
 - exponentiating 12-12
 - introduction and instructions 12-2
 - logarithm of 12-14, 12-15
 - multiplying 12-16
 - passing by value 9-376
 - raising to floating-point power 12-17
 - retrieving results on Alpha 9-138
 - sine of 12-7, 12-18
 - square root of 12-19
 - subtracting 12-20
 - tangent of 12-8, 12-21
- FLUSH statement 4-61
- flushing buffers 4-61
- focus, setting 10-4, 10-5
- font, printing characteristics 17-24

- font_specs structure 17-24
 - FOR-DO statement 4-62 to 4-63
 - FOREACH statement 4-68
 - foreground color
 - ActiveX object 10-5
 - setting 19-5
 - FOR-FROM-THRU statement 4-64 to 4-65
 - FORK routine 9-131
 - form
 - feed 9-12
 - printing 17-23
 - formal argument. *See* argument
 - formatting data 3-28 to 3-32
 - FORMS statement 4-70
 - FOR-UNTIL-DO statement 4-66 to 4-67
 - FP_ routines 12-2 to 12-3, 12-5 to 12-22
 - FP_ADD routine 12-5
 - FP_ARCCOS routine 12-6
 - FP_ARCSIN routine 12-7
 - FP_ARCTAN routine 12-8
 - FP_CMP routine 12-9
 - FP_COS routine 12-10
 - FP_DIV routine 12-11
 - FP_EXPE routine 12-12
 - FP_FROM_NUM routine 12-13
 - FP_LOG10 routine 12-14
 - FP_LOGE routine 12-15
 - FP_MUL routine 12-16
 - FP_POW routine 12-17
 - FP_SIN routine 12-18
 - FP_SQRT routine 12-19
 - FP_SUB routine 12-20
 - FP_TAN routine 12-21
 - FP_TO_NUM routine 12-22
 - FREE routine 9-133
 - FRETURN statement 4-71
 - From class 9-425
 - %FSTAT routine 9-134 to 9-136
 - FSTAT routine 9-137
 - function 1-5 to 1-6
 - beginning 4-73
 - calling 4-204, 7-5
 - as subroutine 7-6
 - from C routine 7-6, 21-3
 - casting 7-10
 - declaring 4-55, 9-376
 - grouping arguments 4-88
 - in class 8-6
 - keeping in memory 4-72, 4-73, 4-182
 - name
 - from segment address 9-337
 - length 4-73
 - passing arguments to 2-35
 - recursive 4-72, 4-73
 - system-supplied 7-9
 - user-defined 4-71, 7-4 to 7-6
 - ^VAL. *See* ^VAL
 - windowing 19-68 to 19-72
 - See also* argument; subroutine, external
 - FUNCTION statement 4-72 to 4-74, 7-5, 7-7
 - FXSUBR routine 9-138
 - result of last floating-point routine 9-137
 - returning value of last floating-point call 9-134
- ## G
- .GE. operator 3-11
 - gennet utility 20-2, 20-4
 - .GES. operator 3-6
 - get method 4-137, 4-139
 - .NET assembly API 20-24
 - GET statement 4-75 to 4-76
 - get_chn_sts runtime interface function 21-14
 - get_errnum runtime interface function 21-15
 - get_gbl_ptr runtime interface function 21-16
 - get_xarg runtime interface function 21-18
 - get_xarg_val runtime interface function 21-21
 - GETCM routine 9-139
 - GETDFN routine 9-140
 - GETFA routine 9-141 to 9-146
 - alternative to 9-267
 - I/O errors and 6-2
 - GetField method 20-21
 - GETFNM routine 9-147
 - GETLOG routine 9-148 to 9-149
 - GetProperty method 20-17
 - GETRFA qualifier 4-218
 - I/O errors and 6-2
 - with manual file locking 4-244
 - GETRFA routine 9-150 to 9-152
 - alternative to 9-267
 - I/O errors and 6-2
 - GETS statement 4-77 to 4-79
 - GIF, printing 17-39
 - GLINE routine 9-153 to 9-154

- global
 - common vs. 2-42
 - data section 2-39, 2-41
 - defining 4-80
 - getting address 21-16
 - size of 2-42
 - literal, overlaying a record 2-45
 - message 1-40
 - record, defining 4-80
 - structure 2-44, 8-21
 - variable 1-24, 2-42
- GLOBAL COMMON statement 4-30
- global data section
 - in class 4-23, 8-7
- GLOBAL modifier 4-99
- global RFA 4-218, 4-237
- GLOBAL statement 2-41, 4-80 to 4-83
- GOTO statement 4-84 to 4-85
- graphic, printing 17-39
- greater than operator 3-6, 3-11
- greater than or equal to operator 3-6, 3-11
- GREEN_PART macro 10-5
- GRFA 4-218, 4-237, 4-244
- group 2-43
 - arguments, declaring 2-36
 - class 8-7
 - defining 2-43, 4-87
 - initializing 4-98
 - message 1-40 to 1-41
 - name 4-87
 - objects in 2-36
 - referencing 1-24, 2-44
 - subroutine and function arguments 4-88
 - usage 2-20
- GROUP modifier 5-26
- GROUP statement 4-86 to 4-89
- .GT. operator 3-11
- GTPPN routine 9-155 to 9-156
- .GTS. operator 3-6
- /guiwnd option 4-232
- GUIWND qualifier 4-221

H

- handle 2-49
 - ActiveX object 10-4
 - appending contents to XML string 18-105
 - See also* memory: handle
- hardware scrolling 19-11, 19-84
 - controlling 19-84
 - terminal required 19-12
 - updating the screen 19-3, 19-23
- header
 - compiler listing 4-105, 5-43
- HTTP
 - adding or modifying in response message from
 - server 13-31
 - receiving value 13-28
 - specifying on GET 13-15, 13-17
 - specifying on POST 13-20, 13-21
 - specifying on RECEIVE 13-27, 13-30
- height
 - display area 19-20
 - processing area 19-18, 19-20, 19-25
 - window 19-77
- %HEX routine 9-157
- hexadecimal, converting to 9-157
- hiding
 - ActiveX container 10-21
 - ActiveX object 10-5
- horizontal line 19-40
- host IP 13-3
- host name, returning 16-22
- HTTP
 - client
 - requesting to post document 13-19 to 13-22
 - requesting to retrieve document 13-15 to 13-18
 - escape codes 9-324 to 9-325
 - header
 - adding or modifying in response message from
 - server 13-31
 - receiving value 13-28
 - specifying on GET 13-15, 13-17
 - specifying on POST 13-20, 13-21
 - specifying on RECEIVE 13-27, 13-30
 - log file 13-17, 13-21, 13-29, 13-31
 - password in URI 13-3
 - proxy server and 13-4
 - sending documents via 13-2 to 13-13

- server
 - creating class object 13-24 to 13-25
 - freeing memory 13-26
 - getting document from 13-15 to 13-18
 - posting document to 13-19 to 13-22
 - proxy 13-4
 - receiving request from client 13-27 to 13-29
 - sending response to client 13-30 to 13-31
 - status codes 13-31
 - unescaping escaped characters 9-336
 - HTTP document transport API 13-2 to 13-13
 - error messages 13-11
 - XML and 18-12
 - %HTTP_CLIENT_GET routine 13-15 to 13-18, 13-7
 - %HTTP_CLIENT_POST routine 13-19 to 13-22, 13-7
 - %HTTP_METHOD routine 13-7, 13-23
 - %HTTP_SERVER_CREATE routine 13-24 to 13-25, 13-7
 - %HTTP_SERVER_DELETE routine 13-26, 13-8
 - %HTTP_SERVER_RECEIVE routine 13-27 to 13-29
 - %HTTP_SERVER_SEND routine 13-30 to 13-31, 13-8
 - HTTPS 13-2, 13-5 to 13-13
 - accepting and responding to a request 13-7
 - ciphers 13-10 to 13-11
 - getting a document 13-7
 - runtime support DLL 13-6
 - testing setup locally 13-9
 - See also* HTTP; HTTP document transport API
 - httpslib file 13-6
 - HTTPSLIB logical 13-5
- I**
- ^I data reference operation 9-158
 - i4 variable, efficiency of 2-24
 - icon, ActiveX container 10-4
 - Icon property 10-4
 - ID, window 19-2
 - .IDENT compiler directive 5-13 to 5-15
 - identifier 1-20
 - defining 4-105 to 4-106, 5-7 to 5-12
 - removing declaration 5-44
 - replacement 1-21
 - .IF compiler directive 5-16
 - IF statement 4-90 to 4-91
 - .IFDEF compiler directive 5-18
 - .IFF compiler directive 5-23 to 5-24
 - .IFNDEF compiler directive 5-20
 - .IFT compiler directive 5-23 to 5-24
 - .ITF compiler directive 5-23 to 5-24
 - IF-THEN-ELSE statement 4-90 to 4-91
 - alternate form 4-92 to 4-93
 - cascaded 4-91
 - image, shareable 9-88 to 9-89
 - implicit formatting 3-28
 - %IMPLIED routine 9-159
 - implied-decimal
 - accessing data as 9-66
 - assignment
 - to alpha 3-28
 - to decimal 3-36
 - to implied-decimal 3-35
 - to integer 3-36
 - boxing 8-19
 - converting
 - from expression 9-159
 - from floating-point 12-22
 - to floating-point 12-13
 - data type 2-8
 - accessing data as 9-66
 - passing 2-15
 - literal 1-23
 - negative, storing 2-8
 - implied-numeric data type 2-10, 2-15
 - implied-packed
 - boxing 8-19
 - data type 2-10
 - passing 2-15
 - IMPORT statement 4-94, 8-2
 - importing namespace 8-2
 - IN modifier 2-33, 2-35
 - .INCLUDE compiler directive 5-25 to 5-34
 - include file, declaring data in 5-31
 - INCR statement 4-96 to 4-97
 - incrementing
 - listing counter 5-35
 - variable 4-96 to 4-97
 - indenting lines of code 4-6
 - index block size 9-172
 - indexed properties 10-4
 - indexed sequential access method. *See* ISAM
 - indexed submode 4-124
 - indexer 8-8
 - indexer property 4-138
 - inheritance 8-15

INIT

- message ID 1-41
- option 2-30, 4-80, 4-81
- statement 4-98

initial value

- array 2-29
- dynamic array 2-26
- field 2-23
- GLOBAL statement 2-30, 4-80, 4-81
- specifying 2-29
- subroutine 2-29
- with field position indicator 2-30

initialization vector, generating 9-73

initializing

- data structure 4-98
- set of variables 4-167
- variable 4-26
- windowing system 19-47

INITPORT routine 9-160 to 9-164

INOUT modifier 2-33, 2-35

input

- case sensitive 9-128
- discarding current line C-1
- field 19-52
- mode 4-124
- no echo 9-129
- terminal 9-130
- time limit 19-45, 19-58
- window, obtaining 19-35, 19-39, 19-42

input/output statement. *See* I/O

inserting line in scrolling area 19-23

instance data 8-7

instantiating class 8-12

%INSTR routine 9-165 to 9-167

%INT routine 9-168

INT_REG 21-6

integer

- assignment
 - to alpha 3-28
 - to decimal 3-37
 - to implied-decimal 3-38
 - to integer 3-38
- byte order conversion 16-27
- converting
 - expression to 9-169
 - native to portable 9-53
 - portable to native 9-54

data type 2-9, 2-15

expression 3-7

ISAM key 9-174

literal 1-24, 4-100

long, byte order conversion 16-32

passing 9-64

portable 9-171

short, byte order conversion 16-28, 16-33

%INTEGER routine 9-169

interface

DBLSORT 9-312

to C 1-9

internal function, creating 1-13

internal subroutine. *See* subroutine, internal

Internet

address

- byte order 16-27, 16-28
- conversion 16-29, 16-30
- retrieving 16-21

port, byte order 16-27, 16-28

Internet Explorer, importing and exporting

certificates 13-8 to 13-9

interprogram communication statement 1-18

interrupt character 9-130, C-1

intrinsic function 7-9

See also function

I/O

device, exchanging data with program 1-16

error list 6-3

statement 1-16

- qualifiers 4-207
- waiting 4-242

window 19-3

/io option 4-231

ipar utility. *See* Synergy Language Tools

irecovr utility. *See* Synergy Language Tools

.IS. operator 3-14, 9-25, 9-75

ISAM

file 1-42

caching 4-232

creating 9-170 to 9-178

density 9-171

key information, returning 9-181 to 9-184, 9-185 to 9-188

status information 9-181 to 9-184, 9-185 to 9-188

See also Synergy Language Tools

- key
 - comparing B-1
 - density 9-175
 - null 9-176
 - order 9-174
 - retrieving information about 9-181 to 9-184, 9-185 to 9-188
 - type 9-174
- ISAMC routine 9-170 to 9-178
- ISCLR routine 9-179 to 9-180
- ISINFO routine 9-181 to 9-184
- ISINFOA routine 9-185 to 9-188
- ISKEY routine 9-189 to 9-191
- isload utility. *See Synergy Language Tools*
- ismvfy utility. *See Synergy Language Tools*
- ISSTS routine 9-192 to 9-194
- isutl utility. *See Synergy Language Tools*
- italic attribute 19-13

J

- JBNAM routine 9-195
- %JBNO routine 9-196 to 9-197
- job number, current process 9-196 to 9-197
- JPEG, printing 17-39
- %JPERIOD routine 9-198 to 9-200
- Julian date 9-198, 9-221
- jump scrolling 19-84
- justification, controlling 3-32

K

- key
 - comparing B-1
 - compressing 9-175
 - contents of 9-201
 - density 9-175
 - duplicates 9-175
 - modifiable 9-175
 - null 9-176
 - of reference
 - defining 4-222
 - resetting 9-34
 - order 9-174
 - retrieving information about 9-181 to 9-184, 9-185 to 9-188
 - specifying 9-170, 9-173
 - type 9-174

- KEYNUM qualifier 4-147, 4-207, 4-222
- %KEYVAL routine 4-59, 4-147, 9-201 to 9-202
- keyword 1-21
- KILL routine 9-203
- killing current process 9-203
- KRF qualifier 4-207

L

- label 1-21
- Language. *See Synergy Language*
- ^LAST qualifier 4-198
- .LE. operator 3-11
- %LEN routine 9-204
- .LES. operator 3-6
- less than operator 3-6, 3-11
- less than or equal to operator 3-6, 3-11
- librarian 1-8
- library
 - external subroutine 1-9
 - including 5-25 to 5-34
- license, ActiveX controls 10-7
- line
 - blank 4-7
 - clearing 19-38
 - drawing character 19-28
 - feed 9-12
 - horizontal 19-40
 - inserting in scrolling area 19-23
 - numbering 9-111
 - vertical 19-44, 19-45
- %LINE routine 9-205
- linker 1-8
 - See also Synergy Language Tools*
- .LIST compiler directive 5-35
- listing counter
 - decrementing 5-37
 - incrementing 5-35
- literal
 - alpha 1-22, 4-6
 - data space 2-40
 - decimal 1-23
 - defining 4-99 to 4-100
 - error 1-24
 - field position indicator 4-100
 - global, overlaying a record 2-45
 - implied-decimal 1-23
 - integer 1-24, 4-100

- name 4-99
 - overlying 2-45, 4-100
 - textual 1-22
 - LITERAL statement 4-99 to 4-100
 - local
 - ActiveX property 10-6
 - data area 2-40, 4-156
 - host name, returning 16-22
 - ID, ActiveX object 10-5
 - message 1-40
 - message facility 1-38 to 1-41
 - record, default 4-182
 - routine structure 2-44
 - variable 1-24, 2-40
 - LOCAL modifier
 - in FUNCTION statement 4-72 to 4-73
 - in LITERAL statement 4-99
 - in RECORD statement 4-156
 - in SUBROUTINE statement 4-182 to 4-183
 - LOCALDATA modifier 4-72, 4-182
 - LOCALIZE routine 9-206 to 9-207
 - LOCASE statement 4-101
 - LOCK qualifier 4-223 to 4-224
 - locking
 - record 4-243 to 4-244
 - defining 4-223 to 4-224, 4-231
 - DETACH statement and 4-39
 - optimistic 4-218, 4-237, 4-244
 - releasing 4-189 to 4-190
 - suspending execution 9-100
 - log file
 - %HTTP_CLIENT_GET 13-17
 - %HTTP_CLIENT_POST 13-21
 - %HTTP_SERVER_RECEIVE 13-29
 - %HTTP_SERVER_SEND 13-31
 - logarithm, floating-point number 12-14, 12-15
 - logging events 9-330
 - logical
 - changing program behavior based on 9-148
 - in directory path 1-43
 - line 4-6
 - operator 3-12
 - screen 19-3
 - See also* environment variable
 - long data type 2-10
 - loop
 - creating 1-13
 - terminating 4-53, 4-118
 - lowercase, converting to 4-101
 - low-order bit, determining if on or off 9-320
 - LPQUE statement 4-102 to 4-104
 - replacing with Windows printing API calls 17-7
 - SJC\$_NOPAGINATE 9-100
 - .LT. operator 3-11
 - .LTS. operator 3-6
- ## M
- ^M data reference operation 9-208
 - machine, identifying 9-105 to 9-106, 9-339
 - macro, parameterized 5-8
 - mailbox
 - creating 9-62
 - deleting 9-96
 - main routine 1-5, 4-105 to 4-106
 - MAIN statement 4-105 to 4-106
 - manual conventions xxix
 - mapping
 - search index to symbol table entry 14-19
 - structure to data area 9-208
 - symbol table entry to search index 14-9
 - margin, screen 19-82
 - MASK qualifier 4-225 to 4-226
 - MATCH qualifier 4-227
 - matching record on OPEN 4-227
 - MATH\$ routine 9-138
 - %MEM_PROC routine 9-210 to 9-217
 - memory
 - estimating for window processing 19-73
 - freeing HTTP server 13-26
 - handle
 - allocating 9-212
 - assigning new 9-214, 9-216, 9-217
 - changing size of 9-216, 9-217
 - releasing 9-213
 - returning size of 9-214
 - XML string 18-110
 - managing 2-48, 9-210
 - registering external segment 9-214, 9-216, 9-217
 - usage summary table 5-41
 - MERGE statement 4-107 to 4-111

- message 1-39
 - disabling searching 1-41
 - displaying 19-101
 - encrypted 13-2
 - facility
 - changing searching algorithm 9-256
 - local 1-38 to 1-41
 - Synergy message manager 1-39 to 1-41
 - global 1-40, 9-99, 9-256
 - group 1-40 to 1-41, 9-256
 - identifying 1-39
 - IDs 1-41
 - INIT ID 1-41
 - local 1-40, 9-256
 - queue 1-39, 1-41, 9-256
 - receiving 1-41, 4-160 to 4-161
 - SELF ID 1-41, 9-258
 - sending 1-38 to 1-41, 4-165 to 4-166
 - waiting for 9-99
 - Windows 10-30, 10-31
 - message manager 1-39 to 1-41
 - metafile
 - playing to printer 17-15
 - retrieving name 17-30
 - saving after printing 17-3, 17-15
 - method 8-5
 - ActiveX 10-15
 - defining 4-112 to 4-115
 - overloading 8-6
 - overriding 4-112 to 4-113, 8-6
 - signature 8-5
 - virtual 4-113
 - METHOD statement 4-112 to 4-115
 - GET 4-137
 - SET 4-137
 - Microsoft Windows. *See* Windows
 - minus sign
 - arithmetic operator 3-8
 - data format 3-30
 - unary operator 3-7
 - MISMATCH modifier 2-33
 - mode 4-127
 - append 4-124
 - defining 4-124
 - input 4-124
 - output 4-124
 - overriding 4-231
 - update 4-124
 - MODNAME routine 9-218 to 9-219
 - modular programming 1-5
 - module 5-13
 - MONEY routine 9-220
 - Monitor utility. *See* Synergy Language Tools
 - monochrome printing 17-23
 - moving
 - cursor 4-42
 - symbol table entry 14-14
 - window 19-82
 - MRETURN statement 4-116
 - multidimensional array. *See* real array
 - multiplication 3-8
 - examples 3-24
 - floating-point numbers 12-16, 12-17
 - implied-decimal or implied-packed operands 3-8
- ## N
- name
 - function 4-73
 - routine 4-105 to 4-106, 4-182 to 4-184
 - namespace 8-2
 - creating 8-2
 - default 8-2
 - defining 4-117
 - importing 8-2
 - See also Environment Variables and System Options*
 - NAMESPACE statement 4-117
 - namspc.def file 14-2
 - %NDATE routine 9-221
 - .NE. operator 3-11
 - negative sign 2-8, 3-7
 - .NES. operator 3-6
 - nested class 8-9
 - .NET
 - interfacing to 20-1 to 20-27
 - type, creating 20-12
 - wrapping classes 20-2, 20-4
 - .NET assembly API 20-1 to 20-27
 - debugging 20-5
 - deleting assembly 20-11
 - deploying in shared configuration 20-3
 - error trapping 20-3
 - loading assembly 20-10

- restrictions 20-2
- sample programs 20-6
- UI Toolkit support 20-5
- Workbench and 20-5
- network
 - adapter 9-196
 - encryption 4-231, 9-171
 - security 13-2, 13-5 to 13-11
- NEW keyword 2-22, 2-26, 8-13
- NEW modifier 4-112 to 4-114, 4-136 to 4-137
- NEXTLOOP statement 4-118
- /nl option 4-231
- nocase ISAM key 9-174
- .NODEBUG compiler directive 5-36
- /nodelay option 4-232
- NOFILL qualifier 4-229
- .NOLIST compiler directive 5-37
- nonkey integer data. *See* portable integer
- NOP statement 4-119
- .NOPROTO compiler directive 5-38
- NOREPLICATE null type 9-176
- not and operator 3-13
- not equal to operator 3-6, 3-11
- .NOT. operator 3-12
- not operator
 - bitwise 3-13
 - Boolean 3-12
- %NSPC_ADD routine 14-8
- %NSPC_ATOS routine 14-9
- NSPC_CLOSE routine 14-10
- NSPC_DELETE routine 14-11
- %NSPC_FIND routine 14-12
- NSPC_GETDATA routine 14-13
- NSPC_MOVE routine 14-14
- %NSPC_OPEN routine 14-15 to 14-16
- NSPC_PUTDATA routine 14-17
- NSPC_RESET routine 14-18
- %NSPC_STOA routine 14-19
- ^NULL 2-5, 2-26
- null key, defining 9-176
- %NUMARGS routine 9-223
- number, terminal 9-346
- numeric
 - assignment to alpha 3-28
 - character 1-10
 - converting to packed 9-232

- data type 2-10
 - accessing formal argument as 9-23
 - passing 2-15
 - expression, returning whole number 9-168
- NUMREC qualifier 4-230

O

- ^O routine 9-224
- O_NDELAY option 4-232
- object 8-1 to 8-21
 - restrictions for defining 2-36
- Object class 4-23, 9-406
 - creating a dynamic array 9-407
 - data type 2-11
 - See also* @* class
- occluding windows 19-4
- %OCT routine 9-226 to 9-227
- octal, alpha expression as 9-224
- OFFERROR statement 4-120
- offset, specifying 2-27
- OLB. *See* Synergy Language Tools
- one, returning value of 9-354
- ONERROR statement 4-121 to 4-123
- op_Addition operator 3-15
- op_BitwiseAnd operator 3-15
- op_BitwiseOr operator 3-15
- op_Decrement operator 3-14
- op_Division operator 3-15
- op_Equality operator 3-16
- op_ExclusiveOr operator 3-15
- op_Explicit operator 3-18
- op_GreaterThan operator 3-16
- op_GreaterThanOrEqual operator 3-16
- op_Increment operator 3-14
- op_Inequality operator 3-16
- op_LessThan operator 3-16
- op_LessThanOrEqual operator 3-16
- op_LogicalAnd operator 3-15
- op_LogicalNot operator 3-15
- op_LogicalOr operator 3-16
- op_Multiply operator 3-15
- op_OnesComplement operator 3-16
- op_Subtraction operator 3-15
- op_True operator 3-15
- op_UnaryNegation operator 3-14
- op_UnaryPlus operator 3-15

- OPEN statement 4-124 to 4-134
 - defining
 - access 4-231
 - buffer size 4-232
 - cached access 4-232
 - file access 4-231
 - file size 4-231
 - fixed-length record format 4-231
 - mode 4-127, 4-231
 - new application window 4-232
 - number of blocks to add 4-232
 - number of local I/O buffers 4-232
 - O_NDELAY option 4-232
 - record locking 4-231
 - record type 4-232
 - sequential record access 4-231
 - stream file 4-231
 - temporary filename 4-231
 - examples 4-134
 - executing UNIX command 4-131
 - setting runtime options 4-231 to 4-232
- OPENELB routine 9-228 to 9-229
- opening
 - channel 4-124 to 4-134
 - symbol table 14-15 to 14-16
 - UNIX pipe 4-131
- OpenSSL 1-35, 13-5 to 13-13
- openssl utility 13-5, 13-6, 13-8
- OPENSSLIB environment variable 1-35
- operand 3-3
- operating system, identifying current 9-105 to 9-106
- operator 3-4
 - arithmetic 3-8
 - assignment 3-4, 3-5, 3-22
 - binary 3-4
 - bitwise 3-13
 - Boolean 3-12 to 3-13
 - class 3-14 to 3-20
 - binary 3-15
 - unary 3-14
 - conversion 3-18
 - precedence level 3-4, 3-20
 - relational 3-11
 - rounding 3-9
 - string relational 3-6
 - unary 3-4, 3-7
- OPT modifier 2-33, 2-35
- OPT_NEWNAME option 19-93
- OPT_NODATA option 19-93, 19-95
- OPT_NOTITLE option 19-93, 19-95
- OPT_NOUSET option 19-93, 19-95
- OPT_NOWFLDS option 19-93, 19-95
- optimistic locking 4-218, 4-237, 4-244
- optimizing at compile time 3-41
- %OPTION routine 9-230 to 9-231
- OPTIONAL modifier 2-33, 2-35
- OPTIONS qualifier 4-231 to 4-232
- .OR. operator 3-12 to 3-13
- or operator
 - bitwise 3-13
 - Boolean 3-12 to 3-13
- OrderBy class 9-437
- orientation
 - character 17-24
 - printed document 17-20
- OUT modifier 2-33, 2-35
- output
 - mode 4-124
 - sending to device for display 4-40 to 4-43
 - terminal, suspending and resuming C-1
- overlapping windows 19-4
- overlay 2-44, 4-156
 - common record 2-45, 4-30
 - external literal 2-45
 - global literal 2-45
 - group 4-87
 - indicator 2-30, 2-44, 4-30
 - initial value 2-45
 - literal 2-45, 4-100
 - local record 2-45
 - object variable 2-45
 - rules 2-45, 4-158
 - window 19-4, 19-86
- overloading method 8-6
- OVERRIDE modifier 4-112 to 4-114, 4-136 to 4-137
- overriding method 8-6

P

- packed
 - converting numeric expression to 9-232
 - data type 2-10
 - passing 2-15
- %PACKED routine 9-232
- padding data 1-36, 9-69, 9-71
- page
 - break 5-39, 5-41
 - ISAM 9-172
- .PAGE compiler directive 5-39
- paint character 9-129
- %PAINT routine 9-129, 9-233
- PAINT routine 9-129, 9-234 to 9-235
- PAK routine 9-236 to 9-238
- palette, color 19-5
 - modifying 19-87
 - returning current codes 19-62
- paper
 - size 17-21, 17-22
 - source 17-25
- parameter
 - accessing as alpha 9-20
 - accessing as its calling type 9-243
 - ActiveX 10-24 to 10-25
 - data types 2-18, 2-34
 - declaring in routine 2-33 to 2-38
 - ending list 4-47
 - formal 7-14
 - modifiers 2-33, 2-35
- parameterized macro 5-8
- parent, ActiveX window 10-16
- parent initializer 4-113, 8-11
- parent-child relationship between windows 19-76
- parenthesis and order of evaluation 3-23
- parity, setting 9-160
- PARSE routine 9-239 to 9-241
- parser, XML 18-6, 18-7
 - creating 18-97
 - getting last error 18-99
 - releasing 18-98
- parsing XML 18-6, 18-13, 18-101
 - example 18-14
 - string 18-102
- PARTIAL modifier 4-22
- ^PASSED data reference operation 9-242
- passing arguments 9-242
- ^PASSTHRU routine 9-243
- password
 - preventing echo 9-129
 - prompting for 13-23
- path
 - characteristics 1-25
 - directory 1-43
 - object 1-27
 - returning current 9-328
 - setting current 9-331
 - specification 1-24
- PEM file 13-17, 13-21, 13-25
 - appending key file to 13-8
 - password 13-23
- pen
 - creating 17-30
 - deleting 17-31
- pen_specs structure 17-30
- period
 - in currency 9-206
 - in data format 3-30
 - separator character 9-128
- physical
 - line 4-6
 - screen 19-3
- pipe, executing command through 4-131
- PKCS padding 1-36, 9-69, 9-71
- placement order, windows 19-62, 19-64, 19-87
- plaintext data, encrypting 9-71
- plus sign 3-7, 3-8
- PNG, printing 17-39
- /po option 4-232
- port
 - initializing 9-160 to 9-164
 - opening 4-127, 4-131
 - returning number given service name 16-24
 - setting server 13-3
- portable integer 4-246, 9-171
- position
 - ActiveX container 10-4, 10-5, 10-16
 - ActiveX control 10-4, 10-5, 10-23
 - application window 19-88
 - cursor, resetting 4-41
 - defining 4-57 to 4-60
 - indicator, field 2-27
 - processing area 19-25

- substring 9-41, 9-282
- window
 - moving by length of field 19-35, 19-38
 - moving specified amount 19-41
 - moving to beginning of field 19-55
 - moving to corner 19-21
 - moving to specified position 19-42
- POSITION qualifier 4-233
- positive sign 3-7
- POSRTA routine 9-150, 9-244 to 9-246
- precedence level, operators 3-20
- prefetching records 4-154
- previewing before printing 17-16 to 17-18, 17-40
- Print dialog 17-12, 17-28
- print submode 4-125
- print_specs structure 17-25
- printing
 - allocating report handle 17-30
 - collating 17-23
 - color 17-23, 17-25
 - device capabilities 17-27 to 17-35
 - duplex 17-23
 - executing 17-15 to 17-19
 - existing metafile 17-15
 - font 17-24
 - form to use 17-23
 - getting or setting capabilities 17-25
 - graphic 17-39
 - number of copies 17-22
 - page range 17-40
 - paper
 - orientation 17-20
 - size 17-21, 17-22
 - source 17-25
 - previewing 17-16 to 17-18, 17-40
 - quality 17-23
 - queueing file 4-102 to 4-104
 - retrieving printer settings 17-20 to 17-26
 - saving metafile after 17-3, 17-15
 - scaling factor 17-22
 - screen C-1
 - settings 17-42 to 17-43
 - status 17-33
 - transparent color 17-40
 - Windows 17-4 to 17-14, 17-15 to 17-43
- priority, setting 9-299
- Privacy Enhanced Mail. *See* PEM file
- PRIVATE modifier 8-16
 - in CLASS statement 4-22
 - in ENUM statement 4-48
 - in field definition 2-21
 - in GROUP statement 4-86
 - in METHOD statement 4-112
 - in PROPERTY statement 4-136 to 4-137
 - in RECORD statement 4-156
 - in STRUCTURE statement 4-179
- privilege, checking current process 9-155
- PROC statement 1-5, 4-135
- procedure division 1-5, 4-135
- process
 - I/O buffers, setting on OPEN 4-211, 4-232
 - identification number 9-196
 - returning information about 9-155
 - setting priority class 9-299
 - status, displaying C-1
 - terminating 9-203
- processing
 - ASCII control codes 4-70
 - errors 6-1 to 6-17
- processing area 19-3
 - adding text at top 19-21
 - attributes 19-17
 - clearing 19-22
 - color 19-19
 - current 19-25
 - default 19-16
 - defining 19-25
 - filling with characters 19-22
 - parameter, copying 19-20
 - placing 19-25
 - saving attributes and data 19-66
 - scrolling contents 19-23
 - size 19-4, 19-18, 19-25
 - transferring current attributes and data 19-66
- product support xxx
- Professional Series Workbench. *See* Workbench
- profiling routines 1-38
- program
 - detached 4-39
 - executing 9-314 to 9-317
 - non-Synergy Language, executing 9-115
 - organization of 1-5
 - suppressing termination message 9-129
 - suspending execution 4-168, 9-385

programmer number, returning 9-155

programming, structured 1-5

project number, returning 9-155

property

ActiveX object 10-4 to 10-6

getting value 10-19

integer value 10-20

setting 10-26

adding to a class 8-8

ambient 10-5, 10-26

control 10-4

declaring 4-136 to 4-139

extended 10-4

indexer 4-138

local 10-6

querying value of 20-17

setting value on an instance 20-15

property indexer

get method 20-24

set method 20-23

PROPERTY statement 4-136 to 4-139

PROTECTED modifier 8-16

in CLASS statement 4-22

in ENUM statement 4-48

in field definition 2-21

in GROUP statement 4-86

in METHOD statement 4-112

in PROPERTY statement 4-136 to 4-137

in RECORD statement 4-156

in STRUCTURE statement 4-179

.PROTO compiler directive 5-38

Prototype utility 1-37

See also Synergy Language Tools

prototyping 1-37

importing prototypes 4-94

turning off and on 5-38

proxy server 13-4

PROXY_HOST environment variable 13-4

PROXY_SUBNET environment variable 13-4

pseudo array 2-25, 2-34, 2-38

PUBLIC modifier 8-16

in CLASS statement 4-22

in ENUM statement 4-48

in field definition 2-21

in GROUP statement 4-86

in METHOD statement 4-112

in PROPERTY statement 4-136 to 4-137

in RECORD statement 4-156

in STRUCTURE statement 4-179

PURGE routine 9-247

PURGE statement 4-140

PUT statement 4-141 to 4-142

put_descr runtime interface function 21-22

put_xarg runtime interface function 21-23

put_xarg_fval runtime interface function 21-24

put_xarg_val runtime interface function 21-26

PUTCM routine 9-248

PUTS statement 4-143 to 4-144

Q

Q_NOWAIT identifier 19-58

Q_WAIT identifier 19-58

QSORT routine 9-249

qualifier 4-207 to 4-242

ALLOC 4-208

BKTSIZ 4-209

BLKSIZ 4-210

BUFNUM 4-211

BUFSIZ 4-212

CONTIG 4-213

DEQ 4-214

DIRECTION 4-215

FDL 4-216

GETRFA 4-218

GUIWND 4-221

KEYNUM 4-207, 4-222

KRF 4-207

LOCK 4-223 to 4-224

MASK 4-225 to 4-226

MATCH 4-227

NUMREC 4-230

OPTIONS 4-231 to 4-232

POSITION 4-233

RECSIZ 4-234

RECTYPE 4-236

RFA 4-237, 9-244

SHARE 4-239

TEMPFILE 4-241

WAIT 4-242

quality of printing 17-23

queueing file for printing 4-102 to 4-104

quotation mark 1-22

R

R5ASC routine 9-251

RAD50

- converting ASCII to 9-28
- converting to ASCII 9-251

RANDM routine 9-253 to 9-254

ranged variable

- absolute 1-32
- reference 1-32
- relative 1-33

%RCB_CALL routine 15-2, 15-6 to 15-7

%RCB_CREATE routine 15-2, 15-8

RCB_DELARG routine 15-10

RCB_DELETE routine 15-3, 15-11

RCB_INSARG routine 15-3, 15-12 to 15-14

RCB_SETARG routine 15-3, 15-15 to 15-17

RCB_SETARGS routine 15-3, 15-19

RCB_SETFNC routine 15-2, 15-20

rcbapi.def file 15-2

RCFLG routine 1-41, 9-255 to 9-257

RCVID routine 9-258 to 9-259

%RDLEN routine 9-260

%RDTRM routine 9-261

READ statement 4-145 to 4-149, 6-2

reading

- direction 4-215
- record
 - next sequential 4-77 to 4-79, 4-150 to 4-155
 - random access 4-75
 - specified 4-145 to 4-149

read-only data space, allocating 4-99 to 4-100

READONLY modifier 2-21, 4-156

READS statement 4-150 to 4-155

- inconsistencies after I/O error 6-2
- input without terminator 9-130
- prefetching records with xfServer 4-154

real array 1-27, 2-25

receiving

- character from channel 4-11 to 4-12
- message 4-160 to 4-161

%RECNUM routine 9-262

record

- address 4-218
- class 8-7
- defining 2-39, 4-156 to 4-159, 4-236
- deleting 4-37 to 4-38
- finding position 4-57 to 4-60

first in file 9-34

fixed-length 4-231, 9-171

initializing 4-98

layout of data area 2-39

length 2-39

local 2-40

- defining 4-156
- function 4-72 to 4-73
- subroutine 4-182 to 4-183

lock 4-243 to 4-244

- defining 4-223 to 4-224, 4-231
- DETACH statement and 4-39
- optimistic 4-218, 4-237, 4-244
- releasing 4-189 to 4-190
- suspending execution 9-100

matching 4-227, 4-237

multiple fixed-length 9-171

naming 2-39

number of 4-230

overlying 2-44

reading 4-145 to 4-149

- direction of 4-215
- next sequential 4-150 to 4-155
- reverse 4-150

returning RFA 4-218

sequential access 4-231

size of

- defining 4-231, 4-234
- returning 9-270

stack 4-156 to 4-157

- function 4-72 to 4-73
- subroutine 4-182 to 4-183

static 4-156

- overlying 2-45

storing 4-177 to 4-178

terminator 9-12, 9-261

variable-length 9-171

writing 4-198 to 4-201, 4-202

RECORD statement 2-40, 4-80, 4-156 to 4-159

RECSIZ qualifier 4-234

/rectype option 4-232

RECTYPE qualifier 4-236

recursive

- function 4-72, 4-73
- subroutine 4-182, 4-183

- RECV statement 1-38, 1-41, 4-160 to 4-161
 - changing default name 9-258
 - no message pending label 9-99
 - searching for message 9-256
- RED_PART macro 10-5
- redrawing screen 19-3, 19-89
- reduction, string 3-5
- re-entering a routine 4-73, 4-183
- REENTRANT modifier
 - function 4-72, 4-73
 - restriction 4-73, 4-183
 - subroutine 4-182, 4-183
- ^REF 2-35, 9-263
 - C interface and 21-5, 21-7
 - contrasted with ^VAL 2-37
 - See also* argument
- reference
 - external, passing argument by 9-400
 - passing argument by 9-263, 21-5, 21-7
- .REGION compiler directive 5-40
- region, defining 5-40
- registering ActiveX control 10-2
- registration string, current runtime 9-298
- relational operator 3-11, 3-24
- relative
 - file 1-42
 - ranging 1-33
 - submode 4-124
- RemoveDelegateFromEvent 20-28
- REMOVEHANDLER statement 4-162
- removing window 19-89
- RENAM routine 9-128, 9-264 to 9-266
- repeat count 19-43
- REPEAT statement 4-163
- replacement identifier 1-21
 - clearing 5-8
 - defining 5-7
 - removing declaration 5-44
- REPLICATE null type 9-176
- report handle 17-4
 - allocating and returning 17-30
 - deleting 17-30
 - retrieving metafile name 17-30
- ReportWriter 1-3
- Repository 1-3, 1-5
 - declaring data in 5-31
 - .INCLUDEing from 5-25 to 5-34, 8-16
- REQ modifier 2-33
- Request to Send
 - disabling flow control 9-162
 - protocol 9-160
 - signal 9-161
- REQUIRED modifier 2-33, 2-35
- resetting cursor position 4-41
- RESIDENT modifier
 - function 4-72 to 4-73
 - subroutine 4-182 to 4-183
- restoring window 19-93
- RestrictedAlphaEnumerator class 9-439
- RETURN statement 4-164, 7-2
- return value
 - data types 2-18, 2-38
 - last function or routine 9-394 to 9-395
- returning
 - control to calling routine 4-205
 - from user-defined function 4-71
- reverse
 - reading next sequential record 4-150
 - video 19-13
- RFA 4-246
 - constant 9-172
 - global 4-218, 4-237
 - matching with record 4-237
 - returning 4-218
 - static 4-246
- RFA qualifier 4-147, 4-237, 9-244
- %RFA routine 9-267
- RGB_VALUE macro 10-5
- RMS
 - block size 4-210
 - bucket size 4-209
 - buffer
 - number of 4-211, 4-232
 - size 4-212, 4-232
- %RND routine 9-268, 9-269
- ROUND modifier
 - in FUNCTION statement 4-72 to 4-74
 - in MAIN statement 4-105
 - in METHOD statement 4-113
 - in SUBROUTINE statement 4-182 to 4-184
- %ROUND routine 9-268, 9-269
- rounding 3-9
 - examples 3-9, 3-10, 3-24
 - expression 9-268, 9-269

- operator 3-9
 - rules for 3-9, 3-10
 - truncating vs.
 - in assignment statements 3-26
 - in FUNCTION statement 4-72 to 4-74
 - in MAIN statement 4-105
 - in METHOD statement 4-113, 4-114
 - in SUBROUTINE statement 4-182 to 4-184
 - routine
 - data division 4-8
 - end of 1-5, 4-46
 - executing from C 21-11
 - main 1-5
 - name 1-22
 - parts of 1-5
 - routine call block
 - API 15-2 to 15-4
 - calling routine with 15-6 to 15-7
 - creating 15-2, 15-8
 - deleting 15-3, 15-11
 - deleting argument from 15-10
 - inserting arguments into 15-12 to 15-14
 - making calls using 15-2
 - reusing 15-8
 - setting arguments 15-3, 15-15 to 15-17, 15-19
 - specifying routine to call 15-2, 15-20
 - %RSIZE routine 9-270 to 9-271
 - RSTAT routine 9-272 to 9-273, 9-275
 - RSTATD routine 9-274
 - %RTERM routine 9-275 to 9-276
 - RUNJB routine 9-277 to 9-280
 - runtime 1-8
 - interface function 21-8 to 21-26
 - option flag, setting 9-98
 - option. *See Synergy Language Tools*
 - setting options 4-231 to 4-232
 - %RVSTR routine 9-281 to 9-282
 - RX_SETMTFNC routine 15-2
 - RX_SHUTDOWN_REMOTE routine 15-3
 - %RX_START_REMOTE routine 15-2
- S**
- S_BLD routine 9-283 to 9-285
 - S_DETAB routine 9-286 to 9-287
 - S_MATCH routine 9-288 to 9-289
 - S_PARSE routine 9-290 to 9-293
 - S_WRAP routine 9-294 to 9-295
 - salt, generating 9-73
 - saving
 - metafile after printing 17-3, 17-15
 - window in compressed form 19-95
 - sbyte data type 2-10
 - scaling factor, printing 17-22
 - scheduled task 9-346
 - scope 1-20, 2-47
 - DATA statement 2-43, 4-33
 - going out of 8-13
 - instance data 2-48
 - local data 2-47
 - TRY-CATCH block 4-186, 6-4
 - \$SCR_ATT function 4-42
 - \$SCR_CLR function 4-41
 - \$SCR_MOV function 4-42
 - \$SCR_POS function 4-41
 - screen
 - changing attributes 4-42
 - clearing 4-41, 19-47
 - function command 4-40, 4-41
 - logical 19-3
 - physical 19-3
 - redrawing 19-3, 19-89
 - resizing 19-90
 - updating 19-3, 19-44
 - warnings 19-102
 - when exiting 19-47
 - scrolling
 - area, inserting line at top or bottom 19-23
 - hardware 19-11
 - controlling 19-84
 - updating the screen 19-3, 19-23
 - jump 19-84
 - processing area 19-23
 - smooth 19-84
 - vertical 19-12, 19-84
 - SCSPREFETCH environment variable 4-154
 - SDMS_SEL routine 9-296
 - SEALED modifier 4-22, 4-112 to 4-114, 4-136 to 4-137
 - search index, symbol table 14-9, 14-19
 - search list
 - logical
 - directory path 1-43
 - renaming file 9-264
 - symbol table 14-2
 - search path 9-264

- Secure Socket Layer. *See* SSL
- segment address, function name from 9-337
- Select class 9-420
- SELF message ID 1-41, 9-258
- semicolon 4-7
- SEND statement 1-38, 4-165 to 4-166
- sending
 - message 1-38 to 1-41, 4-165 to 4-166
 - output to device for display 4-40 to 4-43
- separator character 9-128, 9-206
- sequential
 - file 1-42
 - fixed-length data, writing 4-143
 - record access, optimizing 4-231
 - submode 4-124
- /sequential option 4-231
- serial port, circumventing DOS handling of 9-161
- SERIAL routine 9-298
- server. *See* xfServer; xfServerPlus
- service name, returning 16-25
- service runtime 9-321, 9-346
 - See also Synergy Language Tools*
- servstat program. *See Synergy Language Tools*
- set method 4-137, 4-139
 - .NET assembly API 20-23
- SET statement 4-167
- %SET_PRIORITY_CLASS routine 9-299
- SETCTL routine 9-301 to 9-302
- SETDFN routine 9-303
- SetField method 20-19
- SETKRF routine 9-90
- SETLOG routine 9-304 to 9-305
- SetProperty method 20-15
- /share option 4-231
- SHARE qualifier 4-239
- shared
 - data area 2-41
 - defining 4-30
 - size 4-31
 - image
 - PURGE routine 9-247
 - RUNJB routine 9-279
 - SETLOG routine 9-304
 - SHELL routine 9-308, 9-380
 - SPAWN routine 9-316
 - See also Synergy Language Tools*
 - object 21-27
 - shell meta characters 9-315
 - SHELL routine 9-306 to 9-309
 - short data type 2-10
 - SHORT null type 9-176
 - showing ActiveX container 10-28
 - /si option 4-231
 - sigaction routine 21-7
 - sign
 - implied-decimal literal 1-23
 - numeric literal 1-22
 - storing 2-8
 - signal 21-7
 - simple array. *See* pseudo array
 - simple variable reference 1-29
 - sine, floating-point number 12-18
 - size
 - ActiveX container 10-4, 10-5, 10-16
 - ActiveX control 10-4, 10-5, 10-23
 - expression 9-310
 - ^SIZE data reference operation 9-204, 9-310 to 9-311
 - SIZE routine. *See* ^SIZE data reference operation
 - SJC\$_NOPAGINATE 9-100
 - SKIP_BLNK option 19-14
 - SKIP_BOLD option 19-14
 - SKIP_ITAL option 19-14
 - SKIP_RVRS option 19-14
 - SKIP_UNDR option 19-14
 - SLEEP statement 4-168
 - smooth scrolling 19-84
 - SNDOPR routine 9-91
 - socket
 - abnormal program termination 16-19
 - API 16-3 to 16-13
 - Berkley Sockets vs 16-3
 - programming example 16-10
 - routines 16-15 to 16-47
 - binding name to 16-16
 - connecting 16-18
 - datagram, reading data from 16-38
 - descriptor, deleting 16-17
 - function errors 16-5 to 16-9
 - listening for connections 16-31
 - reading incoming data 16-34, 16-36, 16-38
 - request queue 16-31
 - returning a name 16-26
 - sending information via 13-2

- stream
 - accepting a connection 16-15
 - connecting 16-18
 - reading data from 16-36
 - waiting for data 16-40
- SORT routine 9-312
- SORT statement 4-169 to 4-174
- sorting files 4-169 to 4-174
- source file, `.INCLUDEing` 5-25 to 5-34
- SPAWN routine 9-314 to 9-317, 9-360
- SQL Connection 1-3
- square root, floating-point number 12-19
- `%SS_ACCEPT` routine 16-15
- `%SS_BIND` routine 16-16
- `%SS_CLOSE` routine 16-17
- `%SS_CONNECT` routine 16-18
- `SS_EACCES` error code 16-16
- `SS_ECONNRESET` error code 16-18
- `SS_EDESTADDRREQ` error code 16-18, 16-31
- `SS_EINVAL` error code 16-16, 16-31, 16-35, 16-37
- `SS_ETIMEDOUT` error code 16-40
- `%SS_FATAL` routine 16-19
- `%SS_GETHOSTBYADDR` routine 16-20
- `%SS_GETHOSTBYNAME` routine 16-21
- `%SS_GETHOSTNAME` routine 16-22
- `%SS_GETPEERNAME` routine 16-23
- `%SS_GETSERVBYNAME` routine 16-24
- `%SS_GETSERVBYPORT` routine 16-25
- `%SS_GETSOCKNAME` routine 16-26
- `%SS_HTONL` routine 16-27
- `%SS_HTONS` routine 16-28
- `%SS_INET_ADDR` routine 16-29
- `%SS_INET_NTOA` routine 16-30
- `%SS_LISTEN` routine 16-31
- `%SS_NTOHL` routine 16-32
- `%SS_NTOHS` routine 16-33
- `%SS_RECV` routine 16-34 to 16-35
- `%SS_RECVBUF` routine 16-36 to 16-37
- `%SS_RECVFROM` routine 16-38 to 16-39
- `%SS_SELECT` routine 16-40
- `%SS_SEND` routine 16-41 to 16-42
- `%SS_SENDBUF` routine 16-43 to 16-44
- `%SS_SENDTO` routine 16-45 to 16-46
- `%SS_SOCKET` routine 16-47
- SSL 13-2, 13-5 to 13-13
- STACK modifier
 - in `FUNCTION` statement 4-72 to 4-73
 - in `RECORD` statement 4-156 to 4-157
 - in `SUBROUTINE` statement 4-182 to 4-183
- STACKDATA modifier 4-72, 4-182
- `.START` compiler directive 5-41 to 5-42
- state option
 - retrieving 9-329
 - setting 9-332
- statement
 - assignment 3-26 to 3-40
 - compound 4-9
 - executing
 - repeatedly 4-44
 - set 4-18 to 4-21
 - until condition is true 4-45, 4-90 to 4-91, 4-196
 - until control is transferred 4-44, 4-163
 - until statement is true 4-92 to 4-93
 - until value is reached 4-64, 4-66 to 4-67
 - until values are evaluated 4-62
 - format 4-6
 - I/O 1-16
 - label 1-21
 - qualifiers 4-207
 - Synergy Language 4-1 to 4-206
 - types of 1-10 to 1-18
- static
 - data 8-7
 - memory 2-48
 - record 2-45, 4-156
 - RFA 4-246
- STATIC modifier 2-21
 - in `METHOD` statement 4-112
 - in `PROPERTY` statement 4-136
 - in `RECORD` statement 4-156
- status information
 - ISAM file 9-181 to 9-184, 9-185 to 9-188
 - process, displaying C-2
 - windowing system 19-65
- status utility. *See Synergy Language Tools*
- stop bit, setting 9-160
- STOP statement 4-175 to 4-176
- STORE statement 4-177 to 4-178
- storing data 2-40, 21-22
- stream file 1-42, 4-124, 4-231
- /stream option 4-231

- string
 - concatenation 3-5
 - data type 2-11
 - FDL 4-216
 - module identifier 5-13
 - reduction 3-5
 - relational operator 3-6
 - representation of value 9-318
 - searching for value 9-165
 - XML
 - appending 18-103, 18-104, 18-105
 - clearing contents 18-106
 - copying 18-107
 - creating 18-108
 - element 18-80
 - getting memory handle 18-110
 - manipulating 18-12
 - parsing 18-102
 - releasing 18-109
 - size of 18-111
- String class 9-404
 - data type 2-11, 2-15
- %STRING routine 9-318
- structfield
 - data type 2-13, 2-16
 - initializing 4-98
 - strong prototyping and 2-37
 - using structure with 2-44
- structure 8-7
 - data type 2-13
 - declaring field as 2-13
 - defining 2-44, 4-179 to 4-181
 - mapping to data area 9-208
 - types of 2-44
 - using with a structfield 2-44
- STRUCTURE statement 2-44, 4-179 to 4-181
- structured programming 1-5
- STTY routine 9-319
- submode 4-127
 - block 4-124, 4-128
 - character 4-125
 - defining 4-124
 - indexed 4-124
 - overriding 4-231
 - print 4-125
 - relative 4-124
 - sequential 4-124
 - wildcard 4-125
- subnet mask 13-4
- subroutine, external 1-5, 7-2 to 7-3
 - argument
 - as integer value 21-21
 - loading data into 21-23
 - beginning 4-182 to 4-184
 - calling 4-204, 7-2 to 7-3
 - as function 7-2
 - grouping arguments 4-88
 - in class 8-6
 - keeping in memory 4-183
 - name length 4-183
 - passing arguments to 2-35
 - profiling 1-38
 - recursive 4-182, 4-183
 - returning control from 4-164
 - system-supplied 7-9, 9-9 to 9-402
 - windowing 19-13 to 19-103
 - writing 7-2 to 7-3
 - See also* argument; function
- subroutine, internal 1-5, 7-3
 - calling 4-16 to 4-17
 - in class 8-6
 - returning control from 4-164
- SUBROUTINE statement 4-182 to 4-184, 7-2
- subscripted variable reference 1-29
- substring, position of 9-41, 9-282
- subtraction 3-8
 - alpha expression 3-5
 - floating-point numbers 12-20
- %SUCCESS routine 9-320
- support, contacting xxx
- symbol table
 - adding entry to 14-8
 - API 14-2 to 14-6
 - example 14-3 to 14-6
 - requirements 14-2
 - routines 14-7 to 14-19
 - closing 14-10
 - deleting 14-10
 - deleting entry from 14-11
 - finding entry in 14-12
 - mapping
 - entry to search index 14-9
 - search index to entry 14-19

- modifying entry data 14-17
- moving entry in 14-14
- opening 14-15 to 14-16
- resetting 14-18
- resource allocation 14-2
- retrieving entry's data from 14-13
- symbol table offset 5-41
- symbolic relational operator 3-16
- %SYN_ATEXIT routine 9-321
- %SYN_CHARTOSTR routine 9-322
- %SYN_ESCAPE_HANDLE routine 9-324 to 9-326
- %SYN_FREECHN 9-327
- %SYN_GETDIR 9-328
- %SYN_GETSTATE routine 9-329
- SYN_REPORTEVENT routine 9-330
- %SYN_SETDIR 9-331
- %SYN_SETSTATE routine 9-332
- %SYN_SYSERRTXT 9-334
- SYN_TRANSPARENCY_THRESHOLD environment variable 17-41
- SYN_TRANSPARENT_COLOR environment variable 17-41
- %SYN_UNAME routine 9-335
- %SYN_UNESCAPE_HANDLE routine 9-336
- %SYN_XNAME 9-337
- synbackup utility. *See Synergy Language Tools*
- synckini utility. *See Synergy Language Tools*
- SYNDEFNS environment variable 4-95, 8-2
- Synergex.SynergyDE.Collections.ArrayList class 2-26, 9-417 to 9-419
- Synergex.SynergyDE.Select.AlphaEnumerator class 9-439
- Synergex.SynergyDE.Select.Event class 9-438
- Synergex.SynergyDE.Select.From class 9-425
- Synergex.SynergyDE.Select.OrderBy class 9-437
- Synergex.SynergyDE.Select.Select class 9-420
- Synergex.SynergyDE.Select.Where class 9-427
- Synergex.SynergyDE.SynException class or data type 9-412
- Synergy ActiveX API. *See ActiveX API*
- Synergy application window, opening 4-221
- Synergy compiler. *See compiler*
- Synergy Control Panel. *See Synergy Language Tools*
- Synergy DBMS. *See Synergy Language Tools*
- Synergy debugger. *See Synergy Language Tools*
- Synergy floating point API. *See floating point API*
- Synergy HTTP document transport API. *See HTTP document transport API*
- Synergy Language 1-3
 - components of 1-8 to 1-9
 - daemon 1-39
 - file types 1-42
 - philosophy of 1-3
- Synergy Language Profiler. *See Synergy Language Tools*
- Synergy librarian. *See librarian*
- Synergy linker. *See linker*
- Synergy message manager. *See message manager*
- Synergy namespace API. *See symbol table: API*
- Synergy .NET assembly API. *See .NET assembly API*
- Synergy Prototype utility. *See Prototype utility*
- Synergy routine call block API. *See routine call block: API*
- Synergy runtime. *See runtime*
- Synergy socket API. *See socket: API*
- Synergy symbol table API. *See symbol table: API*
- Synergy Windows printing API. *See Windows: printing API*
- Synergy XML API. *See XML: API*
- Synergy/DE 1-3
- Synergy/DE xfServer. *See xfServer*
- Synergy/DE xfServerPlus. *See xfServerPlus*
- synergy.ini file. *See Environment Variables and System Options*
- SynException class or data type 9-412
- synglobal namespace 4-95, 8-2
- synsock.def file 16-4
- synuser.ini file. *See Environment Variables and System Options*
- synxml.def file 13-3, 13-31, 18-7
- synxml.elb library 18-7
- %SYSERR routine 9-338
- %SYSID routine 9-339
- system operator, sending message to 9-91
- system option
 - #55 4-154
 - #1 1-40
 - #5 9-234
 - #7 1-39, 9-99
 - #11 3-26
 - #12 9-341
 - #13 1-40
 - determining if set 9-230

- setting at runtime 9-230
- See also Environment Variables and System Options*
- system service 9-91, 9-382
- System.Array class 9-410
- System.Collections.ArrayList class 2-26, 8-20, 9-414 to 9-416
- System.Exception class 9-408
- System.Object class. *See* Object class
- System.String class. *See* String class

T

- tab 4-6, 9-12
- tag name, XML 18-66, 18-67
 - finding element by 18-90
 - setting 18-76
- tangent, floating-point number 12-21
- TBYTE qualifier 9-172
- TEMPFILE qualifier 4-133, 4-241
- temporary file 4-133, 4-241
- terabyte file 9-172
- TERM environment variable, scrolling and 19-12
- termcap file
 - scrolling 19-12, 19-24
 - See also Professional Series Portability Guide*
- terminal
 - buffer size 9-341
 - input 9-130
 - I/O, windowing system 19-4
 - name
 - converting to number 9-357
 - returning 9-371
 - number 9-346
 - determining 9-347
 - from name 9-357
 - opening 4-130, 4-131
 - output, suspending and resuming C-1
 - setting flags 9-364
- terminating
 - BEGIN-END block 4-51
 - channel
 - deleting file and 4-140
 - saving file and 4-28 to 4-29
 - iteration 4-118
 - loop 4-53
 - program execution 4-175, C-1
 - routine 4-46
- termination message, suppressing 9-129

- terminator
 - input 9-12, 9-130
 - last record read 9-261, 9-275
 - no echo 9-130
 - record 9-12
- terminfo file
 - scrolling 19-12, 19-24
 - See also Professional Series Portability Guide*
- text
 - adding to top of processing area 19-21
 - copying between windows 19-76
 - printing to window 19-35
- text_specs structure 17-32
- textual literal 1-22
- /tf option 4-231
- TFLSH routine 9-341 to 9-342
- Thawte 13-6
- THEN statement. *See* IF-THEN-ELSE statement
- this keyword 1-27, 4-113, 8-11
- THROW statement 4-185, 6-4
- TIFF, printing 17-39
- time, compilation 9-79
- TIME routine 9-343
- time zone of current machine 9-344
- %TIMEZONE routine 9-344
- .TITLE compiler directive 5-43
- title, window
 - attributes 19-31, 19-61
 - changing 19-27
 - color 19-31, 19-61
 - creating 19-27
 - length of 19-65
 - placement 19-32
 - returning information about 19-61, 19-64
 - setting 19-32
 - turning off 19-32
 - turning on 19-32
- TNMBR environment variable 9-347
- %TNMBR routine 9-346 to 9-348
- Toolkit. *See* UI Toolkit
- tools.def file 9-120, 9-354
- traceback
 - generating 9-109
 - printing current 9-349
- TRACEBACK routine 9-349
- trailing blank, stripping 9-350, 9-352
- transferring program control 1-13, 4-116, 4-205

- transparent color 17-40
- trapping errors 6-2
 - disabling 4-120
 - enabling 4-121 to 4-123
 - WD_ACCEPT and 19-35
 - WD_GETS and 19-39
 - WD_READS and 19-43
 - windowing system 19-98
- %TRIM routine 9-350 to 9-351, 9-352
- %TRIMZ routine 9-352 to 9-353
- true, returning value of 9-354
- %TRUE routine 9-354
- %TRUNC routine 9-355, 9-356
- TRUNCATE modifier
 - in FUNCTION statement 4-72 to 4-74
 - in MAIN statement 4-105
 - in METHOD statement 4-113
 - in SUBROUTINE statement 4-182 to 4-184
- %TRUNCATE routine 9-355, 9-356
- truncating
 - in assignment statements 3-26
 - rounding vs.
 - in FUNCTION statement 4-72 to 4-74
 - in MAIN statement 4-105
 - in METHOD statement 4-113, 4-114
 - in SUBROUTINE statement 4-182 to 4-184
- trusted Certificate Authorities 13-8 to 13-9
- truth value 3-2, 3-12
- TRY statement 4-186, 6-4, 7-4
- TT_NAME_TO_NUMBER routine 9-357
- TT_NUMBER_TO_NAME routine 9-358
- TTBRDCST routine 9-359 to 9-360
- TTCHAR routine 9-361 to 9-363
- TTFGLS routine 9-364 to 9-369
- TTMBX routine 9-370
- %TTNAME routine 9-371
- %TTSTS routine 9-372 to 9-373
- tty, resetting 9-315

U

- UI Toolkit 1-3
 - ActiveX controls with 10-3
 - %FALSE and 9-120
 - .NET assembly with 20-5
 - %TRUE and 9-354
 - windowing subroutines and 19-10 to 19-11

- unary operator 3-4
 - class 3-14
 - plus and minus 3-7
- unboxing 8-19
- UNC path 9-331
- .UNDEFINE compiler directive 5-44
- underline 19-13
- unescaping escaped characters 9-336, 13-4
- Uniform Resource Indicator. *See* URI
- UNIQUE modifier 4-112
- UNIX
 - pipe, opening 4-131
 - “what” string 5-13
- UNLOCK statement 4-189 to 4-190
- UNPAK routine 9-374
- unsigned ISAM key 9-174
- %UNSIGNED routine 9-375
- UPCASE statement 4-191
- update mode 4-124
- updating the screen 19-3, 19-44
 - warnings 19-102
 - when exiting 19-47
- uppercase, converting to 4-191
- URI 13-3 to 13-4
- user data set 19-12
 - changing active 19-58
 - creating 19-58
 - getting data from 19-57
 - naming 19-58
 - putting data into 19-57
 - returning information about 19-64
- user mode 10-5, 10-6
- user-defined function. *See* function
- user-defined subroutine. *See* subroutine, external; subroutine, internal
- username, returning 9-335
- USING statement 4-192 to 4-193
- USING-RANGE statement 4-194 to 4-195
- utility
 - chklock 4-244
 - Synergy Language Profiler 1-38
 - See also* Synergy Language Tools

V

- ^VAL 9-376 to 9-377
 - C interface and 21-5, 21-6
 - contrasted with ^REF 2-37
 - data type 2-11
 - function 7-6 to 7-8
 - passing argument 2-35
 - return type 2-38
 - See also* argument
- value
 - assigning to variable 4-167
 - data type 2-5
 - passing argument by 9-376, 21-5, 21-6
- VARARGS modifier
 - functions 4-72, 4-73
 - methods 4-113, 4-114
 - subroutines 4-182, 4-184
- variable 1-24
 - assigning values to 4-167
 - clearing 4-26
 - common 1-24
 - compiler 9-378
 - declaring 1-11
 - decrementing 4-35 to 4-36
 - defining 1-24
 - European conventions 9-128
 - global 1-24, 2-42
 - incrementing 4-96 to 4-97
 - initializing 4-26
 - length minus trailing blanks 9-350, 9-352
 - loading with contents of common area 9-139
 - local 1-24, 2-40
 - maintaining contents between XCALLs 2-40
 - name 1-21
 - reference 1-29
 - ranged 1-32
 - simple 1-29
 - subscripted 1-29
 - returning the length 9-352
 - shared 1-24
- Variable Usage utility. *See Synergy Language Tools*
- /VARIANT compiler option 9-378
- ^VARIANT data reference operation 9-378
- VeriSign 13-6
- version, Synergy 9-379
- %VERSN routine 9-379
- vertical line 19-44, 19-45

- vertical scrolling 19-12, 19-84
- virtual method 4-113
- VIRTUAL modifier 4-113 to 4-114, 4-136 to 4-137
- visible state of ActiveX object 10-4, 10-5
- Visual Basic, DLLs and 11-10
- VMCMD routine 9-380 to 9-381
- VMMSG routine 9-382 to 9-383
- %VMS routine 9-384
- VOID data type 2-11, 2-38

W

- W_AREA routine 19-16 to 19-25
- W_BRDR routine 19-26 to 19-32
- W_CAPTION routine 19-33
- W_DISP routine 19-34 to 19-46
 - READS and ACCEPT vs. 19-101
 - WRITES and DISPLAY vs. 19-101
- W_EXIT routine 19-3, 19-47, 19-97
- W_FLDS routine 19-48 to 19-58
- %W_INFO function 19-68 to 19-72
- W_INFO subroutine 19-59 to 19-67
- W_INIT routine 19-12, 19-47, 19-73 to 19-74
- W_PROC routine 19-75 to 19-91
- W_RESTORE routine 19-93 to 19-94
- W_RETERR function 19-98
- W_SAVE routine 19-95 to 19-96
- W_UPDT routine 19-3, 19-97, 19-102
- WA_ATTR function 19-16, 19-17
- WA_BOX function 19-16, 19-18
- WA_CHANGE function 19-4, 19-16, 19-18
- WA_COLOR function 19-16, 19-19
- WA_COPY function 19-4, 19-16, 19-20
- WA_CORNER function 19-16, 19-21
- WA_FILL function 19-16, 19-22
- WA_INSERT function 19-16, 19-23
 - hardware scrolling 19-12
 - updating screen 19-3, 19-97
- WA_SCROLL function 19-16, 19-23
 - hardware scrolling 19-12
 - updating screen 19-3, 19-97
- WA_SET function 19-4, 19-16, 19-25
- WAC_DTOP option 19-20
- WAC_LLC option 19-21
- WAC_LRC option 19-21
- WAC_PTOD option 19-20
- WAC_ULC option 19-21
- WAC_URC option 19-21

- WAC_WTOD option 19-20
- WAC_WTOP option 19-20
- WAI_BOTTOM option 19-23
- WAI_TOP option 19-23
- WAIT qualifier 4-242
- %WAIT routine 9-385 to 9-386
- waiting
 - for condition 9-385
 - for data on socket 16-40
 - for I/O statement 4-242
 - for input 19-45, 19-58
 - for message 9-99
- WAS_DOWN option 19-23
- WAS_LEFT option 19-23
- WAS_RIGHT option 19-23
- WAS_UP option 19-23
- WB_ATTR function 19-26, 19-27
- WB_CHAR function 19-26, 19-28
- WB_COLOR function 19-26, 19-30
- WB_DRAGOFF function 19-26, 19-30
- WB_DRAGON function 19-26, 19-30
- WB_NOCELL function 19-26
- WB_OFF function 19-26, 19-30
- WB_ON function 19-26, 19-30
- WB_PARTIAL function 19-26, 19-30
- WB_SBOFF function 19-26, 19-31
- WB_SBON function 19-26, 19-31
- WB_TATTR function 19-26, 19-31
- WB_TCOLOR function 19-26, 19-31
- WB_TITLE function 19-26, 19-32
- WB_TOFF function 19-26, 19-32
- WB_TON function 19-27, 19-32
- WB_TPOS function 19-27, 19-32
- WBT_BEG option 19-32
- WBT_BOTTOM option 19-32
- WBT_CENTER option 19-32
- WBT_END option 19-32
- WBT_LEFT option 19-32
- WBT_RIGHT option 19-32
- WBT_TOP option 19-32
- WC_GET function 19-33
- WC_SET function 19-33
- WD_ACCEPT function 19-34, 19-35
 - updating screen 19-3, 19-97
 - vs. ACCEPT 19-101
- WD_ANSI function 19-34, 19-36
- WD_ATTR function 19-34, 19-37
- WD_BELL function 19-34, 19-37
- WD_CLEAR function 19-34, 19-37
- WD_CLR function 19-34, 19-38
- WD_COLOR function 19-34, 19-38
- WD_FIELD function 19-34, 19-38, 19-43
- WD_GETS function 19-34, 19-39
 - updating screen 19-3, 19-97
 - WD_READS and 19-43
- WD_HERASE function 19-34, 19-40
- WD_HLINE function 19-34, 19-40
- WD_MOVE function 19-34, 19-41
- WD_POS function 19-34, 19-42
- WD_READS function 19-34, 19-42
 - activation characters and 9-12
 - updating screen 19-3, 19-97
 - vs. READ 19-101
- WD_REPEAT function 19-34, 19-43
- WD_UPDT function 19-3, 19-34, 19-44
- WD_VERASE function 19-34, 19-44
- WD_VLINE function 19-34, 19-45
- WD_WAIT function 19-34, 19-45
- WD_WRITES function 19-34, 19-46
- WDC_BOL option 19-38
- WDC_BOW option 19-38
- WDC_EOL option 19-38
- WDC_EOW option 19-38
- WDC_LIN option 19-38
- WDC_WND option 19-38
- weekday, returning 9-387
- WF_ATTR function 19-48, 19-49
- WF_COLOR function 19-48, 19-50
- WF_CREATE function 19-48, 19-50
- WF_FILL function 19-48, 19-51
- WF_GET function 19-48, 19-51
- WF_INPUT function 19-48, 19-52
- WF_PARAM function 19-48, 19-55
- WF_POS function 19-48, 19-55
- WF_PUT function 19-48, 19-51, 19-55
- WF_SET function 19-48, 19-56
- WF_UGET function 19-12, 19-48, 19-57
- WF_UPUT function 19-12, 19-48, 19-57
- WF_USER function 19-12, 19-48, 19-58
- WF_USERID function 19-48, 19-58
- WF_WAIT function 19-48, 19-58
- WFI_GUI option 19-53
- WFI_INSRT option 19-52, 19-54
- WFI_NODEL option 19-52, 19-53 to 19-54

- WFI_PRELOAD option 19-52
- WFI_SEID option 19-53
- WFI_SFULL option 19-53
- WFI_SINVIS option 19-53
- WFI_SNOTPL option 19-53
- WFI_SOCCL option 19-53, 19-53 to 19-54
- WFI_SOVRFL option 19-53, 19-54
- WFI_STMOUT option 19-53
- WFI_SUNDFL option 19-53
- WFI_TFULL option 19-52, 19-54
- WFI_TOCCCL option 19-52, 19-53 to 19-54
- WFI_TOVRFL option 19-54
- WFI_TOVRL option 19-52
- WFI_TUNDFL option 19-53, 19-54
- WFN_ALL option 19-49
- WFN_CURR option 19-49
- WFN_FIRST option 19-49
- WFN_LAST option 19-49
- WFN_NEXT option 19-49
- WFN_PREV option 19-49
- what string 5-13
- Where class 9-427
- WHILE statement 4-196
- whole number, returning 9-168
- WI_ACTIVE function 19-59, 19-60
- WI_AREAS function 19-59, 19-60
- WI_ATTRS function 19-59, 19-61
- WI_BCHR function 19-59, 19-61
- WI_NAME function 19-59, 19-62
- WI_PALET function 19-59, 19-62
- WI_PLACED function 19-59, 19-62
- WI_SYSTEM function 19-59, 19-63
- WI_TITLE function 19-59, 19-64
- WI_USER function 19-12, 19-59, 19-64
- WI_VPLACE function 19-59, 19-64
- WI_WINDOW function 19-59, 19-65
- WI_XFR function 19-59, 19-66
- WID_APP option 19-78, 19-79
- WID_CONTAINER option 19-72, 19-78, 19-79, 19-88, 19-91
- WID_NONE option 19-78
- width
 - display area 19-20
 - processing area 19-18, 19-20, 19-25
 - window 19-77
- WIF_ACTPCOL option 19-69
- WIF_ACTPROW option 19-68
- WIF_BDRFLG option 19-68
- WIF_BMAR option 19-68
- WIF_CAPTURE option 19-68, 19-70, 19-80
- WIF_CHILD option 19-69, 19-71
- WIF_COLPOS option 19-69
- WIF_COLS option 19-68
- WIF_CURFLD option 19-69
- WIF_CURSOR option 19-69, 19-71, 19-80
- WIF_DCOL option 19-69
- WIF_DCOLSIZ option 19-69
- WIF_DRAGBAR option 19-69, 19-71
- WIF_DROW option 19-69
- WIF_DROWSIZ option 19-69
- WIF_FLDS option 19-69
- WIF_HWND option 19-69, 19-70
- WIF_LMAR option 19-68
- WIF_MAXWND option 19-68
- WIF_NEXT option 19-69, 19-71
- WIF_NOMFLG option 19-68
- WIF_NORMALX option 19-69, 19-72
- WIF_NORMALY option 19-69, 19-72
- WIF_OCLFLG option 19-68
- WIF_PARENT option 19-69, 19-71
- WIF_PCOL option 19-69
- WIF_PCOLSIZ option 19-69
- WIF_PLCCOL option 19-68
- WIF_PLCFLG option 19-68
- WIF_PLCROW option 19-68
- WIF_POSX option 19-69, 19-71
- WIF_POSY option 19-69, 19-71
- WIF_PROW option 19-69
- WIF_PROWSIZ option 19-69
- WIF_RMAR option 19-68
- WIF_ROWPOS option 19-69
- WIF_ROWS option 19-68
- WIF_SCOLS option 19-68
- WIF_SROWS option 19-68
- WIF_STATE option 19-69
- WIF_TMAR option 19-68
- WIF_TTCHNL option 19-68
- WIF_TTLFLG option 19-68
- WIF_UFLDS function 19-58
- WIF_UFLDS option 19-69
- WIF_UFLDSIZ function 19-58
- WIF_UFLDSIZ option 19-69
- WIF_UNDER option 19-69, 19-71
- WIF_USERID function 19-58

- WIF_USERID option 19-70
- WIF_USRMEM option 19-70
- WIFH_APP option 19-71
- WIFH_CONTAINER option 19-71
- %WIN_PROC routine 10-30
- %WIN_STOP routine 10-31
- window
 - active 19-60, 19-73
 - area 19-3
 - copying parameters to display area 19-20
 - copying parameters to processing area 19-20
 - returning information about 19-60
 - attribute, returning information about 19-61
 - border 19-27
 - clearing 19-37, 19-38
 - color, returning information about 19-61
 - converting code to use 19-101
 - copying text and attributes 19-76
 - creating 19-77
 - definitions 19-11
 - deleting 19-47, 19-81
 - field 19-48
 - creating 19-50
 - current 19-65
 - manipulating 19-48
 - number of 19-65
 - finding 19-81
 - functions 19-2
 - handle, ActiveX object 10-4
 - ID 19-2
 - input, obtaining 19-35, 19-39, 19-42
 - I/O 19-3
 - margins 19-82
 - moving 19-82, 19-84
 - number of 19-63
 - occluding 19-4
 - overlying 19-4, 19-65, 19-86
 - placed, number of 19-62
 - virtual 19-64
 - placement of 19-65, 19-82, 19-87
 - position
 - moving by length of field 19-35, 19-38
 - moving specified amount 19-41
 - moving to beginning of field 19-55
 - moving to corner 19-21
 - moving to specified position 19-42
 - returning information about 19-65
 - processing 19-75
 - removing 19-89
 - resizing 19-90
 - restoring 19-93
 - returning information about 19-65
 - saving in compressed form 19-95
 - searching for 19-81
 - size 19-65, 19-77
 - status 19-65
 - subroutines 19-13 to 19-97
 - in Windows environments 19-10
 - UI Toolkit and 19-10 to 19-11
 - system
 - chaining between programs 19-74
 - exiting 19-47
 - getting information about 19-63, 19-68 to 19-72
 - initializing 19-47
 - title 19-27, 19-65
 - Windows
 - interface from ActiveX API 10-3, 10-30, 10-31
 - printing API 17-4 to 17-14
 - error handling 17-15, 17-20, 17-27, 17-36
 - examples 17-7
 - processing messages 10-30
 - suspending system processing 10-31
 - windowing subroutines on 19-10
 - windows.def file 19-11
 - winprint.def file 17-24, 17-27, 17-30
 - WIX_AGET option 19-66
 - WIX_APUT option 19-66
 - WIX_DGET option 19-66
 - WIX_DPUT option 19-66
 - WIX_SAGET option 19-66
 - WIX_SDGET option 19-66
 - %WKDAY routine 9-387 to 9-388
 - WNDC environment variable 19-5
 - Workbench 1-3, 10-3, 20-5
 - working storage 2-40
 - WP_CHILD function 19-76
 - WP_CPYBLK function 19-76
 - WP_CREATE function 19-77
 - WP_CURSOR function 19-78
 - WP_DELETE function 19-81
 - WP_FIND function 19-81
 - WP_MARGIN function 19-82
 - WP_MOVE function 19-82
 - WP_OPTION function 19-84

WP_OVERLAY function 19-86
 WP_PALET function 19-87
 WP_PLACE function 19-87
 WP_POSITION function 19-88
 WP_REDRAW function 19-3, 19-89, 19-97
 WP_REMOVE function 19-81, 19-89
 WP_RESIZE function 19-90
 WP_SIZE function 19-91
 WP_STATE function 19-91
 WPC_CAPTURE option 19-78
 WPC_LVCSR option 19-78
 WPC_OFF option 19-78
 WPC_ON option 19-78
 WPC_POS option 19-78
 WPC_SET option 19-78
 WPCS_APPSTARTING option 19-79
 WPCS_ARROW option 19-79
 WPCS_CROSS option 19-79
 WPCS_HANDLE option 19-71, 19-79
 WPCS_HELP option 19-79
 WPCS_IBEAM option 19-79
 WPCS_NO option 19-79
 WPCS_SIZEALL option 19-79
 WPCS_SIZENESW option 19-79
 WPCS_SIZENS option 19-79
 WPCS_SIZENWSE option 19-79
 WPCS_SIZEWE option 19-79
 WPCS_UPARROW option 19-79
 WPCS_WAIT option 19-79
 WPO_COLORMAP option 19-84
 WPO_HIDEBRDR option 19-84
 WPO_JUMP option 19-84
 WPO_KEEPRDR option 19-84
 WPO_NOSCROLL option 19-84
 WPO_SMOOTH option 19-84
 WPO_VSCROLL option 19-84
 WPOC_OFF option 19-84
 WPOC_ON option 19-84
 WPOC_ULINE option 19-84
 %WPR_EXECUTE routine 17-15 to 17-19
 %WPR_GETDEVICE routine 17-20 to 17-26
 %WPR_INFO routine 17-27 to 17-35
 %WPR_PRINT routine 17-36 to 17-41
 %WPR_SETDEVICE routine 17-42 to 17-43
 WPS_HIDDEN option 19-72, 19-91
 WPS_MAXIMIZED option 19-72, 19-91
 WPS_MINIMIZED option 19-72, 19-91

WPS_NORMAL option 19-72, 19-91
 WRITE statement 4-198 to 4-201
 WRITES statement 4-202 to 4-203
 writing
 fixed-length data 4-141 to 4-142
 record 4-198 to 4-201, 4-202
 sequential fixed-length data 4-143
 WS_CLIPCHILDREN window style 10-4

X

X character 3-29
 ^X data reference operation 9-389
 X.509 support 13-2
 %XADDR routine 9-391
 XARGS routine 9-393
 XCALL statement 4-204
 xcall.h file 21-3
 xcallv.h file 21-4
 xclapif.o file 21-27
 xclapif.obj file 21-27
 xclapi.o file 21-27
 xclapi.obj file 21-27
 xfNetLink 1-3, 15-13
 xfODBC 1-3
 xfServer 1-3, 4-154
 xfServerPlus 1-3, 15-2
 catching errors 4-122
 EXEC routine and 9-115
 FLAGS routine and 9-128
 PURGE routine and 9-247
 RUNJB routine and 9-279
 SETLOG routine and 9-304
 SHELL routine and 9-308, 9-380
 SPAWN routine and 9-316
 XML
 API 18-4 to 18-111
 compiler option 18-7
 defines 18-17
 requirements 18-7
 success of operation 18-18
 assembling Synergy data in 18-9, 18-16
 attribute
 adding 18-75
 adding list of attributes to list 18-31
 adding to list 18-30
 assigning handle as value 18-28
 clearing list 18-33

- copying 18-19, 18-34
- counting 18-35
- creating 18-20
- creating list 18-36
- finding 18-38, 18-41
- getting name 18-22
- getting value 18-23, 18-65
- inserting in list 18-39
- manipulating 18-11
- manipulating list 18-11
- releasing 18-21
- releasing list 18-37
- removing 18-71
- removing from attribute list 18-42
- returning 18-60
- returning handle 18-24
- setting name 18-26
- setting value 18-27
- character set 18-44
- declaration 18-47, 18-51
 - adding to document 18-43
 - removing 18-49
- document 18-4
 - adding declaration to 18-43
 - creating 18-45
 - declaration 18-47, 18-51
 - loading into memory 18-5
 - manipulating 18-9
 - releasing 18-46
 - removing declaration from 18-49
 - returning root element 18-48
 - setting root element 18-52
 - well-formed 18-5
 - writing contents to file 18-53
 - writing contents to string 18-54
- element 18-4
 - adding attribute 18-75
 - adding attributes in a list to 18-55
 - adding child to 18-57, 18-58
 - adding list of elements to list 18-82
 - adding to list 18-81
 - assigning handle 18-78
 - attribute value 18-65
 - child 18-61
 - clearing list contents 18-84
 - copying 18-62
 - copying list 18-85
 - counting 18-86
 - creating 18-63
 - creating list 18-87
 - finding 18-89, 18-90, 18-93
 - getting child elements 18-66
 - getting tag name 18-67
 - inserting in list 18-91
 - manipulating 18-10
 - manipulating list 18-12
 - releasing 18-64
 - releasing list 18-88
 - removing attributes from 18-72
 - removing child from 18-73, 18-74
 - removing from element list 18-94
 - returning attributes 18-60
 - returning root 18-48
 - returning text handle 18-69
 - setting root 18-52
 - setting tag name 18-76
 - setting text value 18-77
 - string representation 18-80
 - text value 18-68
- embedding 18-5
- escape codes 9-324
- parser 18-6, 18-7
 - creating 18-97
 - getting last error 18-99
 - releasing 18-98
- parsing 18-101
 - example 18-14
 - string 18-102
- processing data 18-13
- sending and receiving data 13-3
- setting option 18-95
- string
 - appending 18-103, 18-104, 18-105
 - clearing contents 18-106
 - copying 18-107
 - creating 18-108
 - getting memory handle 18-110
 - manipulating 18-12
 - parsing 18-102
 - releasing 18-109
 - size of 18-111
 - writing document to 18-54

- tag name 18-66, 18-67
 - finding element by 18-90
 - setting 18-76
- unescaping escaped characters 9-336
- version 18-43
- %XML_ATTR_COPY routine 18-19
- %XML_ATTR_CREATE routine 18-20
- %XML_ATTR_DELETE routine 18-21
- %XML_ATTR_GETNAME routine 18-22
- %XML_ATTR_GETVALUE routine 18-23
- %XML_ATTR_GETVALUEHANDLE routine 18-24
- %XML_ATTR_SETNAME routine 18-26
- %XML_ATTR_SETVALUE routine 18-27
- %XML_ATTR_SETVALUEHANDLE routine 18-28
- XML_ATTR_TYPE 18-17
- %XML_ATTRLIST_ADD routine 18-30
- %XML_ATTRLIST_ADDLIST routine 18-31
- %XML_ATTRLIST_CLEAR routine 18-33
- %XML_ATTRLIST_COPY routine 18-34
- %XML_ATTRLIST_COUNT routine 18-11, 18-35
- %XML_ATTRLIST_CREATE routine 18-36
- %XML_ATTRLIST_DELETE routine 18-37
- %XML_ATTRLIST_FIND routine 18-38
- %XML_ATTRLIST_INSERT routine 18-39
- %XML_ATTRLIST_ITEM routine 18-11, 18-41
- %XML_ATTRLIST_REMOVE routine 18-42
- XML_ATTRLIST_TYPE 18-17
- %XML_DOC_ADDDECLARATION routine 18-43
- %XML_DOC_CREATE routine 18-9, 18-16, 18-45
- %XML_DOC_DELETE routine 18-9, 18-13, 18-16, 18-46
- %XML_DOC_GETDECLARATION routine 18-47
- %XML_DOC_GETROOT routine 18-9, 18-13, 18-48
- %XML_DOC_REMOVEDDECLARATION routine 18-49
- %XML_DOC_SETCOMMENT routine 18-50
- %XML_DOC_SETDECLARATION routine 18-51
- %XML_DOC_SETROOT routine 18-9, 18-16, 18-52
- %XML_DOC_TOFILE routine 18-9, 18-16, 18-53
- %XML_DOC_TOSTRING routine 18-9, 18-12, 18-16, 18-54
- XML_DOC_TYPE 18-17
- %XML_ELEM_ADDATTRIBUTES routine 18-55
- %XML_ELEM_ADDCHILD routine 18-16, 18-57
- %XML_ELEM_ADDCHILDREN routine 18-58
- %XML_ELEM_ATTRIBUTES routine 18-11, 18-60
- %XML_ELEM_CHILDREN routine 18-10, 18-12, 18-13, 18-61
- %XML_ELEM_COPY routine 18-62
- %XML_ELEM_CREATE routine 18-16, 18-63
- %XML_ELEM_DELETE routine 18-64
- %XML_ELEM_GETATTRIBUTE routine 18-65, 18-10, 18-11, 18-13
- %XML_ELEM_GETELEMENTSBYNAME routine 18-12, 18-13, 18-66, 18-10
- %XML_ELEM_GETNAME routine 18-67
- %XML_ELEM_GETTEXT routine 18-10, 18-13, 18-68
- %XML_ELEM_GETTEXTHANDLE routine 18-69
- %XML_ELEM_REMOVEATTRIBUTE routine 18-71
- %XML_ELEM_REMOVEATTRIBUTES routine 18-72
- %XML_ELEM_REMOVECHILD routine 18-73
- %XML_ELEM_REMOVECHILDREN routine 18-74
- %XML_ELEM_SETATTRIBUTE routine 18-10, 18-11, 18-16, 18-75
- %XML_ELEM_SETNAME routine 18-16, 18-76
- %XML_ELEM_SETTEXT routine 18-10, 18-77
- %XML_ELEM_SETTEXTHANDLE routine 18-78
- %XML_ELEM_TOSTRING routine 18-80
- XML_ELEM_TYPE 18-17
- %XML_ELEMLIST_ADD routine 18-81
- %XML_ELEMLIST_ADDLIST routine 18-82
- %XML_ELEMLIST_CLEAR routine 18-84
- %XML_ELEMLIST_COPY routine 18-85
- %XML_ELEMLIST_COUNT routine 18-86, 18-12
- %XML_ELEMLIST_CREATE routine 18-87
- %XML_ELEMLIST_DELETE routine 18-88
- %XML_ELEMLIST_FIND routine 18-12, 18-89
- %XML_ELEMLIST_FINDTAG routine 18-12, 18-90
- %XML_ELEMLIST_INSERT routine 18-91
- %XML_ELEMLIST_ITEM routine 18-12, 18-93
- %XML_ELEMLIST_REMOVE routine 18-94
- XML_ELEMLIST_TYPE 18-18
- %XML_OPTION routine 18-95
- %XML_PARSER_CREATE routine 18-8, 18-13, 18-97
- %XML_PARSER_DELETE routine 18-8, 18-13, 18-98
- %XML_PARSER_ERROR routine 18-99
- %XML_PARSER_PARSEFILE routine 18-8, 18-13, 18-101
- %XML_PARSER_PARSESTRING routine 18-8, 18-12, 18-102
- XML_PARSER_TYPE 18-18
- %XML_STRING_APPEND routine 18-12, 18-103
- %XML_STRING_APPENDDDYN routine 18-12, 18-104

%XML_STRING_APPENDHANDLE routine 18-12,
 18-105
 %XML_STRING_CLEAR routine 18-106
 %XML_STRING_COPY routine 18-107
 %XML_STRING_CREATE routine 18-12, 18-108
 %XML_STRING_DELETE routine 18-109
 %XML_STRING_GETHANDLE routine 18-12, 18-16,
 18-110
 %XML_STRING_GETSIZE routine 18-12, 18-16,
 18-111
 XML_STRING_TYPE 18-18
 XML_SUCCESS 18-18
 XON/XOFF protocol 9-161
 .XOR. operator 3-12
 XRETURN statement 4-205, 7-2
 XSTAT routine 9-394 to 9-395
 XSUBR routine 9-391, 9-397 to 9-399
 ^XTRNL data reference operation 9-400, 21-5

Z

Z character 3-30
 zero
 leading 1-24
 returning value of 9-120
 zoned numeric representation 9-402
 %ZONED routine 9-402

